Язык программирования C# 6.0 и платформа .NET 4.6

7-е издание

Эндрю Троелсен
Филипп Джепикс

www.williamspublishing.com
Apress®
www.apress.com

Язык программирования C# 6.0 и платформа .NET 4.6

C# 6.0 and the .NET 4.6 Framework

Seventh Edition

Andrew Troelsen
Philip Japikse

Apress®

Язык программирования C# 6.0 и платформа .NET 4.6

7-е издание

Эндрю Троелсен
Филипп Джепикс

Москва • Санкт-Петербург • Киев
2016

ББК 32.973.26-018.2.75
Т70
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского *Ю.Н. Артеменко*
Под редакцией *Ю.Н. Артеменко*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Троелсен, Эндрю, **Джепикс**, Филипп.

Т70 Язык программирования C# 6.0 и платформа .NET 4.6, 7-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2016. — 1440 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-2099-7 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание
Эндрю Троелсен, Филипп Джепикс

Язык программирования C# 6.0 и платформа .NET 4.6 7-е издание

Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Павлютин*

Подписано в печать 25.07.2016. Формат 70×100/16.
Гарнитура Times.
Усл. печ. л. 116,1. Уч.-изд. л. 91,55.
Тираж 500 экз. Заказ № 4783

Отпечатано в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-2099-7 (рус.)
ISBN 978-1-4842-1333-9 (англ.)

Оглавление

Содержание

Об авторах

Эндрю Троелсен обладает более чем 20-летним опытом в индустрии программного обеспечения. На протяжении этого времени он выступал в качестве разработчика, преподавателя, автора, публичного докладчика, а теперь возглавляет команду и является ведущим инженером в компании Thomson Reuters. Он был автором многочисленных книг, посвященных миру Microsoft, в которых раскрывалась разработка для COM на языке C++ с помощью ATL, COM и взаимодействия с .NET, а также разработка на языках Visual Basic и C# с применением платформы .NET. Эндрю Троелсен получил степень магистра в области разработки программного обеспечения (MSSE) в Университете Святого Томаса и работает над получением второй степени магистра по математической лингвистике (CLMS) в Вашингтонском университете.

Филипп Джепикс имеет дело с компьютерами, начиная с середины 1980-х годов, и занимался их построением, объединением в сети и созданием программного обеспечения. В настоящее время он сосредоточен на разработке программного обеспечения, архитектурах уровня предприятия и гибких методологиях. Будучи обладателем звания Microsoft MVP с 2009 года, Филипп является постоянным членом сообщества разработчиков, руководителем группы пользователей Cincinnati .NET (`www.cinnug.org`), основанной Cincinnati Day of Agile (`www.dayofagile.org`), а также контент-директором и соведущим подкаста Hallway Conversations (`www.hallwayconversations.com`). Найти Филиппа Джепикса можно на конференциях в США и Европе, в Твиттере (`@skimedic`) и в блоге по адресу `www.skimedic.com/blog/`. За рамками технологий Филипп является верным мужем и отцом троих детей, заядлым лыжником (20-летний член Национального лыжного патруля) и лодочником.

Благодарности

Эндрю Троелсен. Как всегда, я хочу искренне поблагодарить всю команду издательства Apress. Мне повезло работать с Apress при написании многочисленных книг, начиная с 2001 года. Помимо подготовки высококачественного технического материала персонал великолепен, и без него эта книга не увидела бы свет. Спасибо вам всем!

Я также выражаю благодарность своему соавтору Филиппу Джепиксу. Спасибо, Фил, за напряженную работу по поддержанию той же самой степени доступности текста книги наряду добавлением персонального опыта и мнения. Я уверен, что наша книга и ее читатели непременно выиграют от этого нового партнерства!

Напоследок я благодарю свою жену Мэнди и сына Ссорена за поддержку на протяжении моих последних писательских проектов. Люблю вас очень.

Филипп Джепикс. Я также хочу поблагодарить издательство Apress и всю команду, вовлеченную в работу над данной книгой. Это моя вторая книга, изданная в Apress, и я впечатлен тем уровнем поддержки, который мы получили в процессе написания. Кроме того, я хочу выразить благодарность Эндрю за приглашение поучаствовать в проекте. Данная книга стала главным продуктом с самого начала моей карьеры, связанной с платформой .NET; в моей библиотеке есть экземпляры всех ее изданий! Я очень горжусь этой работой и результатами сотрудничества по выходу настоящего издания на рынок. Я благодарю вас, читатель, и надеюсь, что книга окажется полезной в вашей карьере, как было в моем случае. Наконец, я не сумел бы сделать эту работу без моей семьи и поддержки, которую я получил от них. Без вашего понимания, сколько времени занимает написание и вычитывание, мне никогда не удалось бы завершить работу! Люблю вас всех!

Всему роду Троелсен: Мэри (матери), Уолтеру (отцу), Мэнди (жене) и Сорену (сыну). Нам не хватает тебя, Микко (кот).

Эндрю

Моей семье, Эми (жене), Коннеру (сыну), Логану (сыну) и Скайлер (дочери), спасибо за поддержку и терпение с вашей стороны.

Филипп

Введение

Первое издание этой книги вышло в 2001 году в то же самое время, когда компания Microsoft выпустила сборку Beta 2 платформы .NET 1.0. Работа над первым изданием, безусловно, была сложной задачей, поскольку в процессе написания книги API-интерфейсы и язык C# постоянно менялись. В то время полнота платформы .NET с точки зрения разработчика была вполне управляемой. Инфраструктура Windows Forms являлась единственным API-интерфейсом для построения графических пользовательских интерфейсов настольных приложений в рамках платформы, инфраструктура ASP.NET была сконцентрирована исключительно на модели программирования с веб-формами, а язык C# не содержал ничего лишнего помимо средств объектно-ориентированного программирования.

На протяжении первых шести изданий книги я был единственным автором, ответственным за обновление текста для учета многочисленных изменений в языке C# и новых API-интерфейсов внутри платформы .NET. Эта книга обновлялась в течение последних 14 лет, чтобы принять во внимание модель программирования Language Integrated Query (LINQ), Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), новые модели и ключевые слова, связанные с многопоточностью, новые инструменты для разработки, а также изменения в инфраструктуре программирования веб-приложений (не считая многих других аспектов).

Начиная с седьмого издания, мне стало ясно, что полное обновление книги потребовало бы исключительно долгого времени. Несомненно, моя собственная жизнь стала намного более занятой, чем было в 2001 году (и даже в 2011 году; я подозреваю, что причина в том, что я стал отцом, или, может быть, слишком большое число университетских курсов... трудно сказать). Так или иначе, но когда в Apress мне предложили обновить книгу с целью учета последнего выпуска платформы .NET, мы обдумали целый ряд разных подходов, чтобы получить книгу своевременно. В конечном счете, мы пришли к заключению, что обновление текста без посторонней помощи может занять значительное время. В Apress мне посоветовали встретиться с Филиппом Джепиксом, чтобы посмотреть, получится ли из этого хорошая партия. После нескольких телефонных разговоров, переписки по электронной почте и тщательных размышлений я был счастлив пригласить его присоединиться к проекту. Я рад сообщить, что настоящая книга является результатом совместных усилий меня самого и соавтора Филиппа Джепикса. Позвольте Филиппу представить себя самостоятельно...

Эндрю Троелсен

Займись этим, Филипп!

Когда компания Microsoft выпускала ранние бета-версии .NET, я уже плотно занимался технологией Microsoft. Я перешел с других технологий на Visual Basic и строил клиентские приложения, а также веб-сайты на классическом ASP с использованием VB и MTS. Хотя на то время эти инструменты работали хорошо, я предвидел их скорый конец.

Исследование других стеков технологий привело меня к первому изданию данной книги. Я прочитал ее от корки до корки и увидел не только перспективу .NET, но также важность этой книги в моей библиотеке технологий. Наличие одной книги, раскрывающей язык C# и всю экосистему .NET, было бесценным. Не могу точно сказать, сколько экземпляров настоящей книги продало издательство Apress на протяжении прошедших лет, но своим клиентам и слушателям я рекомендовал приобрести каждое издание в качестве первой книги по C# и .NET. Независимо от того, начинаете вы изучать .NET или просто интересуетесь нововведениями в последней версии платформы, трудно найти книгу лучше этой.

Я уже написал одну книгу для Apress, посвященную Windows 8.1 и C#, так что был знаком с командой и оценил методы, которыми издательство Apress содействует работе и поддерживает своих авторов. Когда мой редактор связался со мной и спросил, не хотел бы я совместно потрудиться над седьмым изданием "книги Троелсена" (так я ее всегда называл), я был взволнован и польщен. Это было по-настоящему любимой работой, и мне очень трудно выразить словами, насколько я счастлив считать себя частицей данной книги. Я надеюсь, что вы получите столько же удовольствия при чтении книги, сколько я получил во время ее написания и работы с Эндрю и потрясающей командой из Apress.

Филипп Джепикс

Авторы и читатели — одна команда

Авторам книг по технологиям приходится писать для очень требовательной группы людей. Вам известно, что построение программных решений с применением любой платформы или языка исключительно сложно и специфично для отдела, компании, клиентской базы и поставленной задачи. Возможно, вы работаете в индустрии электронных публикаций, разрабатываете системы для правительства или местных органов власти либо сотрудничаете с NASA или какой-то военной отраслью. Вместе мы трудимся в нескольких отраслях, включая разработку обучающего программного обеспечения для детей (Oregon Trail/Amazon Trail), разнообразных производственных систем и проектов в медицинской и финансовой сфере. Написанный вами код на месте вашего трудоустройства на 100% будет иметь мало общего с кодом, который создавали мы на протяжении многих лет.

По указанной причине в этой книге мы намеренно решили избегать демонстрации примеров кода, свойственного какой-то конкретной отрасли или направлению программирования. Таким образом, мы объясняем язык C#, объектно-ориентированное программирование, среду CLR и библиотеки базовых классов .NET с использованием примеров, не привязанных к отрасли. Вместо того чтобы заставлять каждый пример наполнять сетку данными, подчитывать фонд заработной платы или делать еще что-нибудь подобное, мы придерживаемся темы, близкой каждому из нас: автомобили (с добавлением умеренного количества геометрических структур и систем расчета заработной платы для сотрудников). И вот тут наступает ваш черед.

Наша работа заключается в как можно лучшем объяснении языка программирования C# и основных аспектов платформы .NET. Мы также будем делать все возможное для того, чтобы снарядить вас инструментами и стратегиями, которые вам необходимы для продолжения обучения после завершения работы с данной книгой.

Ваша работа предусматривает усвоение этой информации и ее применение к решению своих задач программирования. Мы полностью отдаем себе отчет, что ваши проекты, скорее всего, не будут связаны с автомобилями и их дружественными именами, но именно в этом состоит суть прикладных знаний.

Мы уверены, что после освоения тем и концепций, представленных в этой книге, вы сможете успешно строить решения .NET, которые соответствуют вашей конкретной среде программирования.

Краткий обзор книги

Книга логически разделена на восемь частей, каждая из которых содержит несколько связанных друг с другом глав. Ниже приведено краткое содержание частей и глав.

Часть I. Введение в C# и платформу .NET

Часть I этой книги предназначена для ознакомления с природой платформы .NET и различными инструментами разработки (включая межплатформенные IDE-среды), которые используются во время построения приложений .NET.

Глава 1. Философия .NET

Первая глава выступает в качестве основы для всего остального материала. Ее главная цель в том, чтобы представить вам набор строительных блоков .NET, таких как общеязыковая исполняющая среда (Common Language Runtime), общая система типов (Common Type System), общеязыковая спецификация (Common Language Specification) и библиотеки базовых классов. Здесь вы впервые взглянете на язык программирования C# и формат сборок .NET. Кроме того, будет исследована независимая от платформы природа .NET.

Глава 2. Создание приложений на языке C#

Целью этой главы является введение в процесс компиляции файлов исходного кода C# с применением разнообразных средств и приемов. Здесь вы узнаете о роли инструментов разработки Microsoft Express и совершенно бесплатного (но полнофункционального) продукта Visual Studio Community Edition, который используется повсеместно в книге. Мы также кратко коснемся роли IDE-среды Xamarin и покажем, как она делает возможной разработку приложений .NET в средах операционных систем Linux и Mac OS X. Вы также научитесь конфигурировать машину разработки с установленной локальной копией всей важной документации .NET Framework 4.6 SDK.

Часть II. Основы программирования на C#

Темы, представленные в этой части книги, очень важны, потому что они связаны с разработкой программного обеспечения .NET любого типа (например, веб-приложений, настольных приложений с графическим пользовательским интерфейсом, библиотек кода или служб Windows). Здесь вы узнаете о фундаментальных типах данных .NET, научитесь манипулировать текстом и ознакомитесь с ролью модификаторов параметров C# (включая необязательные и именованные аргументы).

Глава 3. Главные конструкции программирования на C#: часть I

В этой главе начинается формальное исследование языка программирования C#. Здесь рассматривается роль метода `Main()` и многочисленные детали, касающиеся внутренних типов данных .NET, среди которых манипулирование текстовыми данными с применением типов `System.String` и `System.Text.StringBuilder`. Кроме того, будут описаны итерационные конструкции и конструкции принятия решений, сужающие и расширяющие операции, а также ключевое слово `unchecked`.

Глава 4. Главные конструкции программирования на C#: часть II

В этой главе завершается исследование ключевых аспектов C#. Будет показано, каким образом конструировать перегруженные методы типов и определять параметры с использованием ключевых слов `out`, `ref` и `params`. Также рассматриваются два средства языка C#: именованные и необязательные параметры. Вы узнаете, как создавать

и манипулировать массивами данных, а также определять типы, допускающие `null` (с операциями `?` и `??`). Вдобавок вы освоите отличия между типами значений (в том числе перечисления и специальные структуры) и ссылочными типами.

Часть III. Объектно-ориентированное программирование на C#

В этой части вы изучите ключевые конструкции языка C#, включая детали объектно-ориентированного программирования. Здесь вы научитесь обрабатывать исключения времени выполнения и взаимодействовать со строго типизированными интерфейсами.

Глава 5. Инкапсуляция

В этой главе начинается рассмотрение концепций объектно-ориентированного программирования (ООП) на языке C#. После представления главных принципов ООП (инкапсуляции, наследования и полиморфизма) будет показано, как строить надежные типы классов с применением конструкторов, свойств, статических членов, констант и полей только для чтения. Глава завершается исследованием частичных определений типов, синтаксиса инициализации объектов и автоматических свойств.

Глава 6. Наследование и полиморфизм

Здесь вы ознакомитесь с оставшимися двумя главными принципами ООП (наследованием и полиморфизмом), которые позволяют создавать семейства связанных типов классов. Вы узнаете о роли виртуальных и абстрактных методов (и абстрактных базовых классов), а также о природе полиморфных интерфейсов. Наконец, в главе будет объясняться роль главного базового класса платформы .NET — `System.Object`.

Глава 7. Структурированная обработка исключений

В этой главе обсуждаются способы обработки аномалий, возникающих во время выполнения, в кодовой базе за счет использования структурированной обработки исключений. Вы узнаете не только о ключевых словах C#, которые дают возможность решать такие задачи (`try`, `catch`, `throw`, `when` и `finally`), но и о разнице между исключениями уровня приложения и уровня системы. Вдобавок в главе будут исследоваться разнообразные инструменты внутри Visual Studio, которые позволяют отлаживать исключения, оставшиеся без внимания.

Глава 8. Работа с интерфейсами

Материал этой главы опирается на ваше понимание объектно-ориентированной разработки и посвящен программированию на основе интерфейсов. Здесь вы узнаете, каким образом определять классы и структуры, поддерживающие несколько линий поведения, обнаруживать эти линии поведения во время выполнения и выборочно скрывать какие-то из них с применением явной реализации интерфейсов. В дополнение к созданию специальных интерфейсов вы научитесь реализовывать стандартные интерфейсы, доступные внутри платформы .NET, и использовать их для построения объектов, которые могут сортироваться, копироваться, перечисляться и сравниваться.

Часть IV. Дополнительные конструкции программирования на C#

В этой части книги вы углубите знания языка C# за счет исследования нескольких более сложных (и важных) концепций. Здесь вы завершите ознакомление с системой типов .NET, изучив интерфейсы и делегаты. Вы также узнаете о роли обобщений, кратко взглянете на язык LINQ (Language Integrated Query) и освоите ряд более сложных средств C# (например, методы расширения, частичные методы и манипулирование указателями).

Глава 9. Коллекции и обобщения

Эта глава посвящена обобщениям. Вы увидите, что обобщенное программирование предлагает способ создания типов и членов типов, которые содержат заполнители, указываемые вызывающим кодом. По существу обобщения значительно улучшают производительность приложений и безопасность в отношении типов. Здесь не только описаны разнообразные обобщенные типы из пространства имен `System.Collections.Generic`, но также показано, каким образом строить собственные обобщенные методы и типы (с ограничениями и без).

Глава 10. Делегаты, события и лямбда-выражения

Целью этой главы является прояснение типа делегата. Выражаясь просто, делегат .NET представляет собой объект, который указывает на определенные методы в приложении. С помощью делегатов можно создавать системы, которые позволяют многочисленным объектам участвовать в двухстороннем взаимодействии. После исследования способов применения делегатов .NET вы ознакомитесь с ключевым словом `event` языка C#, которое упрощает манипулирование низкоуровневыми делегатами в коде. В завершение вы узнаете о роли лямбда-операции C# (=>), а также о связи между делегатами, анонимными методами и лямбда-выражениями.

Глава 11. Расширенные средства языка C#

В этой части главе вы сможете углубить понимание языка C# за счет исследования нескольких расширенных приемов программирования. Здесь вы узнаете, как перегружать операции и создавать специальные процедуры преобразования (явного и неявного) для типов. Вы также научитесь строить и взаимодействовать с индексаторами типов и работать с расширяющими методами, анонимными типами, частичными методами и указателями C#, используя контекст небезопасного кода.

Глава 12. LINQ to Objects

В этой главе начинается исследование языка интегрированных запросов (LINQ). Язык LINQ дает возможность строить строго типизированные выражения запросов, которые могут применяться к многочисленным целевым объектам LINQ для манипулирования данными в самом широком смысле этого слова. Здесь вы изучите API-интерфейс LINQ to Objects, который позволяет применять выражения LINQ к контейнерам данных (например, массивам, коллекциям и специальным типам). Эта информация будет полезна позже при рассмотрении других API-интерфейсов, таких как LINQ to XML, LINQ to DataSet, PLINQ и LINQ to Entities.

Глава 13. Время существования объектов

В финальной главе этой части исследуется управление памятью средой CLR с использованием сборщика мусора .NET. Вы узнаете о роли корневых элементов приложения, поколений объектов и типа `System.GC`. После представления основ будут рассматриваться темы освобождаемых объектов (реализующих интерфейс `IDisposable`) и процесса финализации (с применением метода `System.Object.Finalize()`). В главе также описан класс `Lazy<T>`, позволяющий определять данные, которые не будут размещаться вплоть до поступления запроса со стороны вызывающего кода. Вы увидите, что эта возможность очень полезна, когда нежелательно загромождать кучу объектами, которые в действительности программе не нужны.

Часть V. Программирование с использованием сборок .NET

Эта часть книги посвящена деталям формата сборок .NET. Здесь вы узнаете не только о том, как развертывать и конфигурировать библиотеки кода .NET, но также о внутреннем устройстве двоичного образа .NET. Будет описана роль атрибутов .NET и распознавания информации о типе во время выполнения. Кроме того, объясняется роль исполняющей среды динамического языка (Dynamic Language Runtime — DLR) и ключевого слова `dynamic` языка C#. Наконец, рассматриваются более сложные темы, касающиеся сборок, такие как домены приложений, синтаксис языка CIL и построение сборок в памяти.

Глава 14. Построение и конфигурирование библиотек классов

На самом высоком уровне термин "сборка" применяется для описания двоичного файла `*.dll` или `*.exe`, созданного с помощью компилятора .NET. Однако в действительности понятие сборки намного шире. Здесь будет показано, чем отличаются однофайловые и многофайловые сборки, как создавать и развертывать сборки обеих разновидностей, а также, каким образом конфигурировать закрытые и разделяемые сборки с помощью XML-файлов `*.config` и специальных сборок политик издателя. В ходе этого раскрывается внутренняя структура глобального кеша сборок (Global Assembly Cache — GAC).

Глава 15. Рефлексия типов, позднее связывание и программирование на основе атрибутов

В этой главе продолжается исследование сборок .NET. Здесь будет показано, как обнаруживать типы во время выполнения с использованием пространства имен `System.Reflection`. Посредством типов из этого пространства имен можно строить приложения, способные считывать метаданные сборки на лету. Вы также узнаете, как загружать и создавать типы динамически во время выполнения с применением позднего связывания. Напоследок в главе обсуждается роль атрибутов .NET (стандартных и специальных). Для закрепления материала в главе демонстрируется конструирование расширяемого приложения Windows Forms.

Глава 16. Динамические типы и среда DLR

В версии .NET 4.0 появился новый аспект исполняющей среды .NET, который называется исполняющей средой динамического языка (DLR). Используя DLR и ключевое слово `dynamic` языка C#, можно определять данные, которые в действительности не будут распознаваться вплоть до времени выполнения. Такие средства существенно упрощают решение ряда сложных задач программирования для .NET. В этой главе вы ознакомитесь со сценариями применения динамических данных, включая использование API-интерфейсов рефлексии .NET и взаимодействие с унаследованными библиотеками COM с минимальными усилиями.

Глава 17. Процессы, домены приложений и объектные контексты

Базируясь на хорошем понимании вами сборок, в этой главе подробно раскрывается внутреннее устройство загруженной исполняемой сборки .NET. Целью главы является иллюстрация отношений между процессами, доменами приложений и контекстными границами. Упомянутые темы формируют основу для главы 19, где будет исследоваться конструирование многопоточных приложений.

Глава 18. Язык CIL и роль динамических сборок

Последняя глава в этой части преследует двойную цель. В первой половине главы рассматривается синтаксис и семантика языка CIL, а во второй — роль пространства имен `System.Reflection.Emit`. Типы из этого пространства имен можно применять для построения программного обеспечения, которое способно генерировать сборки .NET в памяти во время выполнения. Формально сборки, которые определяются и выполняются в памяти, носят название динамических сборок.

Часть VI. Введение в библиотеки базовых классов .NET

К этому моменту вы уже должны хорошо ориентироваться в языке C# и в подробностях формата сборок .NET. В данной части книги ваши знания расширяются исследованием нескольких часто используемых служб, которые можно обнаружить внутри библиотек базовых классов .NET, включая создание многопоточных приложений, файловый ввод-вывод и доступ к базам данных посредством ADO.NET. Здесь также раскрывается процесс создания распределенных приложений с применением Windows Communication Foundation (WCF) и API-интерфейс LINQ to XML.

Глава 19. Многопоточное, параллельное и асинхронное программирование

Эта глава посвящена построению многопоточных приложений. В ней демонстрируются приемы, которые можно использовать для написания кода, безопасного к потокам. Глава начинается с краткого напоминания о том, что собой представляет тип делегата .NET, и объяснения внутренней поддержки делегата для асинхронного вызова методов. Затем рассматриваются типы из пространства имен `System.Threading`, а также библиотека параллельных задач (Task Parallel Library — TPL). С применением TPL разработчики .NET могут строить приложения, которые распределяют рабочую нагрузку по всем доступным процессорам в исключительно простой манере. В главе также раскрыта роль API-интерфейса PINQ (Parallel LINQ), который предлагает способ создания запросов LINQ, масштабируемых среди множества процессорных ядер. В завершение главы исследуются разнообразные ключевые слова языка C#, позволяющие интегрировать асинхронные вызовы методов непосредственно в язык.

Глава 20. Файловый ввод-вывод и сериализация объектов

Пространство имен `System.IO` позволяет взаимодействовать со структурой файлов и каталогов машины. В этой главе вы узнаете, как программно создавать (и удалять) систему каталогов. Вы также научитесь перемещать данные между различными потоками (например, файловыми, строковыми и находящимися в памяти). Кроме того, в главе рассматриваются службы сериализации объектов платформы .NET. Сериализация позволяет сохранить состояние объекта (или набора связанных объектов) в потоке для последующего использования. Десериализация представляет собой процесс извлечения объекта из потока в память с целью потребления внутри приложения. После описания основ вы освоите настройку процесса сериализации с применением интерфейса `ISerializable` и набора атрибутов .NET.

Глава 21. ADO.NET, часть I: подключенный уровень

В этой первой из трех глав, посвященных базам данных, представлено введение в API-интерфейс доступа к базам данных платформы .NET, который называется ADO.NET. В частности, здесь рассматривается роль поставщиков данных .NET и взаимодействие с реляционной базой данных с использованием подключенного уровня ADO.NET, который представлен объектами подключения, объектами команд, объектами транзакций и объек-

тами чтения данных. В главе также демонстрируется процесс создания специальной базы данных и первой версии специальной библиотеки доступа к данным (AutoLotDAL.dll).

Глава 22. ADO.NET, часть II: автономный уровень

В этой главе изучение взаимодействия с базами данных продолжается описанием автономного уровня ADO.NET. Вы узнаете о роли типа DataSet и объектов адаптеров данных. Вдобавок вы ознакомитесь с многочисленными инструментами Visual Studio, которые могут значительно упростить создание приложений, управляемых данными. Наряду с этим вы научитесь привязывать объекты DataTable к элементам пользовательского интерфейса и применять запросы LINQ к находящимся в памяти объектам DataSet, используя LINQ to DataSet.

Глава 23. ADO.NET, часть III: Entity Framework

В этой главе исследование ADO.NET завершается рассмотрением роли инфраструктуры Entity Framework (EF). По существу EF предлагает способ написания кода доступа к данным с применением строго типизированных классов, которые напрямую отображаются на бизнес-модель. Здесь вы узнаете о роли класса DbContext инфраструктуры EF, об использовании аннотаций данных для формирования базы данных и о реализации хранилищ для инкапсуляции общего кода, поддержки транзакций, перемещений, параллелизма и перехвата. Кроме того, вы научитесь взаимодействовать с реляционными базами данных с помощью LINQ to Entities. В главе также будет создана финальная версия специальной библиотеки доступа к данным (AutoLotDAL.dll), которая будет применяться в нескольких оставшихся главах книги.

Глава 24. Введение в LINQ to XML

В главе 12 была представлена основная модель программирования LINQ, а именно — LINQ to Objects. Здесь вы углубите свои знания языка LINQ, освоив применение запросов LINQ к документам XML. Первым делом будут описаны сложности с манипулированием данными XML, которые существовали в .NET изначально, когда использовались типы из сборки System.Xml.dll. Затем вы узнаете, как создавать документы XML в памяти, сохранять их на жестком диске и перемещаться по их содержимому с помощью модели программирования LINQ (LINQ to XML).

Глава 25. Введение в Windows Communication Foundation

До этого места в книге все примеры приложений запускались на единственном компьютере. В настоящей главе вы ознакомитесь с API-интерфейсом Windows Communication Foundation (WCF), который позволяет создавать распределенные приложения в симметричной манере независимо от лежащих в их основе низкоуровневых деталей. Здесь будет объясняться конструкция служб, хостов и клиентов WCF. Вы увидите, что службы WCF являются чрезвычайно гибкими, поскольку предоставляют клиентам и хостам возможность использования конфигурационных файлов на основе XML, в которых декларативно указываются адреса, привязки и контракты.

Часть VII. Windows Presentation Foundation

Первоначальный API-интерфейс для построения графических пользовательских интерфейсов настольных приложений, поддерживаемый платформой .NET, назывался Windows Forms. Хотя он по-прежнему полностью доступен, в .NET 3.0 программистам был предложен замечательный API-интерфейс под названием Windows Presentation Foundation (WPF), который быстро стал заменой модели программирования настольных приложений Windows Forms. По существу WPF позволяет строить настольные приложения с векторной графикой, интерактивной анимацией и операциями привязки данных,

используя декларативную грамматику разметки XAML. Более того, архитектура элементов управления WPF предлагает легкий способ радикального изменения внешнего вида и поведения типового элемента управления с помощью правильно оформленной разметки XAML.

Глава 26. Введение в Windows Presentation Foundation и XAML

Инфраструктура WPF позволяет создавать исключительно интерактивные и многофункциональные пользовательские интерфейсы для настольных приложений (и косвенно для веб-приложений). В отличие от Windows Forms инфраструктура WPF интегрирует набор ключевых служб (например, двухмерную и трехмерную графику, анимацию и форматированные документы) в единую унифицированную объектную модель. В этой главе предлагается введение в WPF и расширяемый язык разметки приложений (Extendable Application Markup Language — XAML). Здесь вы узнаете, как создавать приложения WPF без XAML, только с XAML и с комбинацией обоих подходов. В завершение главы рассматривается пример построения специального редактора XAML, который будет использоваться в остальных главах, посвященных WPF.

Глава 27. Программирование с использованием элементов управления WPF

В этой главе будет показано, как работать с элементами управления и диспетчерами компоновки, предлагаемыми WPF. Вы узнаете, каким образом создавать системы меню, окна с разделителями, панели инструментов и строки состояния. Также в главе рассматриваются API-интерфейсы (и связанные с ними элементы управления), входящие в состав WPF, в том числе Documents API, Ink API, команды, маршрутизируемые события, модель привязки данных и свойства зависимости.

Глава 28. Службы визуализации графики WPF

Инфраструктура WPF является API-интерфейсом, интенсивно использующим графику, и с учетом этого WPF предоставляет три подхода к визуализации графических данных: фигуры, рисунки и геометрии и визуальные объекты. В настоящей главе вы ознакомитесь с каждым подходом и изучите несколько важных графических примитивов (например, кисти, перья и трансформации). Кроме того, вы узнаете, как встраивать векторные изображения в графику WPF, а также выполнять операции проверки попадания в отношении графических данных.

Глава 29. Ресурсы, анимация, стили и шаблоны

В этой главе освещены важные (и взаимосвязанные) темы, которые позволят углубить знания API-интерфейса WPF. Первым делом вы изучите роль логических ресурсов. Система логических ресурсов (также называемых объектными ресурсами) предлагает способ именования и ссылки на часто используемые объекты внутри приложения WPF. Затем вы узнаете, каким образом определять, выполнять и управлять анимационной последовательностью. Вы увидите, что применение анимации WPF не ограничивается видеоиграми или мультимедиа-приложениями. И, наконец, вы ознакомитесь с ролью стилей WPF. Подобно веб-странице, использующей CSS или механизм тем ASP.NET, приложение WPF может определять общий вид и поведение для набора элементов управления.

Глава 30. Уведомления, команды, проверка достоверности и MVVM

Эта глава начинается с исследования трех основных возможностей инфраструктуры: уведомлений, команд WPF и проверки достоверности. При рассмотрении уведомлений вы узнаете о наблюдаемых моделях и коллекциях, а также о том, как они поддерживают

данные приложения и пользовательский интерфейс в синхронизированном состоянии. Затем вы научитесь создавать специальные команды для инкапсуляции кода. В разделе, посвященном проверке достоверности, вы ознакомитесь с несколькими механизмами проверки достоверности, которые доступны для применения в приложениях WPF. Глава завершается исследованием шаблона "модель-представление-модель представления" (Model View ViewModel — MVVM) и созданием приложения, демонстрирующего шаблон MVVM в действии.

Часть VIII. ASP.NET

Эта часть посвящена построению веб-приложений с использованием API-интерфейса ASP.NET. Инфраструктура ASP.NET предназначена для моделирования процесса создания настольных пользовательских интерфейсов путем наложения управляемой событиями объектно-ориентированной инфраструктуры поверх стандартного запроса/ответа HTTP. В первых трех главах раскрываются основы веб-программирования и Web Forms, а в финальной главе рассматриваются два новейших элемента в ASP.NET: MVC и Web API.

Глава 31. Введение в ASP.NET Web Forms

В этой главе начинается изучение процесса разработки веб-приложений с помощью ASP.NET. Вы увидите, что сценарии серверной стороны теперь заменены настоящими объектно-ориентированными языками программирования (например, C# и VB.NET). Здесь обсуждается конструкция веб-страницы ASP.NET, внутренняя программная модель и другие ключевые аспекты ASP.NET, такие как выбор веб-сервера и применение файлов `Web.config`.

Глава 32. Веб-элементы управления, мастер-страницы и темы ASP.NET

В то время как предыдущая глава была посвящена созданию объектов `Page` из ASP.NET, в этой главе исследуются элементы управления, которые наполняют внутреннее дерево элементов управления. Здесь вы найдете описание основных веб-элементов управления ASP.NET, включая элементы управления проверкой достоверности, элементы управления навигацией по сайту и разнообразные операции привязки данных. Кроме того, рассматривается роль мастер-страниц и механизма тем ASP.NET, который является альтернативой серверной стороны традиционным таблицам стилей.

Глава 33. Управление состоянием в ASP.NET

Эта глава расширяет ваши знания ASP.NET описанием различных способов управления состоянием в .NET. Подобно классическому ASP в ASP.NET довольно легко создавать cookie-наборы, а также переменные уровня приложения и уровня сеанса. Кроме того, ASP.NET предлагает еще один прием управления состоянием: кеш приложения. После исследования нескольких способов поддержки состояния в ASP.NET вы узнаете о роли базового класса `HttpApplication` и научитесь динамически переключать поведение веб-приложения во время выполнения с использованием файла `Web.config`.

Глава 34. ASP.NET MVC и ASP.NET Web API

В этой главе раскрываются две новейшие связанные друг с другом инфраструктуры ASP.NET: MVC и Web API. Инфраструктура ASP.NET MVC основана на шаблоне "модель-представление-контроллер" (Model View Controller); после ее освоения вы построите приложение MVC. Вы узнаете о формировании шаблонов в Visual Studio, маршрутизации, контроллерах, действиях и представлениях. Затем вы создадите REST-службу Web API для обработки операций создания, чтения, обновления и удаления (CRUD) складских данных (с применением библиотеки `AutoLotDAL.dll`), а напоследок обнови-

те приложение MVC для использования новой службы вместо обращения напрямую к `AutoLotDAL.dll`.

Загружаемые приложения

В дополнение к печатным материалам на веб-сайте издательства для загрузки доступны два дополнительных приложения в формате PDF. В них раскрывается несколько дополнительных API-интерфейсов платформы .NET, которые вы можете счесть удобными в повседневной работе. В частности, вы найдете следующие материалы:

- Приложение А. Программирование с помощью Windows Forms
- Приложение Б. Независимая от платформы разработка приложений .NET с помощью Mono

В первом приложении рассматриваются основы API-интерфейса Windows Forms и предоставляются сведения, необходимые для воссоздания ряда настольных графических пользовательских интерфейсов, которые приводились в начальных главах книги (до описания Windows Presentation Foundation). Второе приложение взято из предшествующего издания этой книги; в нем раскрывается роль платформы Mono более подробно, чем это делалось в главах 1 и 2 настоящей книги. Однако имейте в виду, что экранные снимки в приложении Б были получены в более старой IDE-среде MonoDevelop, которая теперь заменена продуктом Xamarin Studio (рассмотренным в главе 2). Тем не менее, основные примеры кода Mono будут работать ожидаемым образом.

Исходный код примеров

Исходный код всех примеров, рассмотренных в книге, доступен для загрузки на веб-сайте издательства.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

Украины: 03150, Киев, а/я 152

ЧАСТЬ I

Введение в C# и платформу .NET

В этой части

Глава 1. Философия .NET

Глава 2. Создание приложений на языке C#

ГЛАВА 1

Философия .NET

Платформа Microsoft .NET (и связанный с ней язык программирования C#) впервые была представлена приблизительно в 2002 году и быстро стала главной опорой современной индустрии разработки программного обеспечения. Как отмечалось во вводном разделе этой книги, при ее написании преследовались две цели. Первая из них — предоставление читателям глубокого и подробного описания синтаксиса и семантики языка C#. Вторая (не менее важная) цель — иллюстрация применения многочисленных API-интерфейсов .NET, в том числе доступ к базам данных с помощью ADO.NET и Entity Framework (EF), набор технологий LINQ, инфраструктуры WPF и WCF, а также разработка веб-сайтов с использованием ASP.NET. Как говорится, пеший поход длиной 1000 километров начинается с первого шага, который и будет сделан в настоящей главе.

Задача главы заключается в построении концептуальной основы для успешного освоения всего остального материала книги. Здесь вы найдете высокоуровневое рассмотрение нескольких связанных с .NET тем, таких как сборки, общий промежуточный язык (Common Intermediate Language — CIL) и оперативная компиляция (Just-In-Time Compilation — JIT). В дополнение к предварительному обзору ряда ключевых слов C# вы узнаете о взаимоотношениях между разнообразными компонентами платформы .NET, среди которых общеязыковая исполняющая среда (Common Language Runtime — CLR), общая система типов (Common Type System — CTS) и общеязыковая спецификация (Common Language Specification — CLS).

Кроме того, в главе представлен обзор функциональности, поставляемой в библиотеках базовых классов .NET, для обозначения которых иногда применяется аббревиатура BCL (base class library — библиотека базовых классов). Здесь вы кратко ознакомитесь с независимой от языка и платформы природой .NET (это действительно так; платформа .NET вовсе не ограничивается операционной системой Windows). Как должно быть понятно, многие темы будут более детально исследоваться в оставшихся главах книги.

Начальное знакомство с платформой .NET

До того, как компания Microsoft выпустила язык C# и платформу .NET, разработчики программного обеспечения, создававшие приложения для операционных систем семейства Windows, часто использовали модель программирования COM. Технология COM (Component Object Model — модель компонентных объектов) позволяла строить библиотеки кода, которые можно было разделять между несходными языками программирования. Например, программист на C++ мог построить библиотеку COM, которой мог пользоваться разработчик на Visual Basic. Независимая от языка природа COM, безусловно, была удобной; тем не менее, технология COM досаждала своей усложненной инфраструктурой, хрупкой моделью развертывания и возможностью работы только в среде операционной системы Windows.

Несмотря на сложность и ограничения COM, с применением этой архитектуры было успешно создано буквально бесчисленное количество приложений. Однако в наши дни большинство приложений, ориентированных на операционные системы семейства Windows, создаются без использования модели COM. Взамен приложения для настольных компьютеров, веб-сайты, службы операционной системы и библиотеки многократно применяемой логики доступа к данным и бизнес-логики строятся с помощью платформы .NET.

Некоторые основные преимущества платформы .NET

Как уже упоминалось, язык C# и платформа .NET впервые были представлены в 2002 году и предназначались для обеспечения более мощной, гибкой и простой модели программирования по сравнению с COM. В оставшихся главах книги вы увидите, что .NET Framework представляет собой программную платформу для построения приложений, функционирующих под управлением операционных систем семейства Windows, а также многочисленных операционных систем производства не Microsoft, таких как Mac OS X и различные дистрибутивы Unix и Linux. Ниже приведен краткий перечень некоторых ключевых средств, предоставляемых .NET.

- *Возможность взаимодействия с существующим кодом.* Это, несомненно, очень полезно. Существующее программное обеспечение COM можно смешивать (т.е. взаимодействовать) с более новым программным обеспечением .NET и наоборот. С выходом .NET 4.0 и последующих версий возможность взаимодействия дополнительно упростилась благодаря добавлению ключевого слова `dynamic` (рассматривается в главе 16).
- *Поддержка многочисленных языков программирования.* Приложения .NET могут быть созданы с использованием любого числа языков программирования (C#, Visual Basic, F# и т.д.).
- *Общий исполняющий механизм, разделяемый всеми поддерживающими .NET языками.* Одним из аспектов этого механизма является наличие четко определенного набора типов, которые способен понимать каждый поддерживающий .NET язык.
- *Языковая интеграция.* В .NET поддерживается межъязыковое наследование, межъязыковая обработка исключений и межъязыковая отладка кода. Например, можно определить базовый класс в C# и расширить этот тип в Visual Basic.
- *Обширная библиотека базовых классов.* Данная библиотека предоставляет тысячи предварительно определенных типов, которые позволяют строить библиотеки кода, простые терминальные приложения, графические настольные приложения и веб-сайты уровня предприятия.
- *Упрощенная модель развертывания.* В отличие от COM библиотеки .NET не регистрируются в системном реестре. Более того, платформа .NET позволяет нескольким версиям одной и той же сборки `*.dll` благополучно сосуществовать на одном и том же компьютере.

Эти и многие другие темы будут подробно рассматриваться в последующих главах.

Введение в строительные блоки платформы .NET (CLR, CTS и CLS)

Теперь, когда вы узнали кое-что об основных преимуществах, присущих платформе .NET, давайте рассмотрим три ключевых (и взаимосвязанных) компонента, которые делают все это возможным — CLR, CTS и CLS. С точки зрения программиста .NET мо-

жет восприниматься как исполняющая среда и обширная библиотека базовых классов. Уровень исполняющей среды правильно называть *общеязыковой исполняющей средой* (Common Language Runtime) или, сокращенно, *средой CLR*. Главной задачей CLR является автоматическое обнаружение, загрузка и управление объектами .NET. Вдобавок среда CLR заботится о ряде низкоуровневых аспектов, таких как управление памятью, обслуживание приложений, координирование потоков и выполнение базовых проверок, связанных с безопасностью (помимо прочих низкоуровневых деталей).

Еще одним строительным блоком платформы .NET является *общая система типов* (Common Type System) или, сокращенно, *система CTS*. Спецификация CTS полностью описывает все возможные типы данных и все программные конструкции, поддерживаемые исполняющей средой, указывает, каким образом эти сущности могут взаимодействовать друг с другом, и как они представлены в формате метаданных .NET (которые рассматриваются далее в этой главе, а более подробно — в главе 15).

Важно понимать, что отдельно взятый язык, совместимый с .NET, может не поддерживать абсолютно все функциональные средства, которые определены спецификацией CTS. Поэтому существует родственная *общеязыковая спецификация* (Common Language Specification) или, сокращенно, *спецификация CLS*, где описано подмножество общих типов и программных конструкций, которое должны поддерживать все языки программирования для .NET. Таким образом, если вы строите типы .NET, открывающие доступ только к совместимым с CLS средствам, то можете быть уверены в том, что их смогут использовать все языки, поддерживающие .NET. И, наоборот, если вы применяете тип данных или программную конструкцию, которая выходит за границы CLS, то не можете гарантировать, что каждый язык программирования для .NET окажется способным взаимодействовать с вашей библиотекой кода .NET. К счастью, как вы увидите далее в этой главе, компилятору C# довольно просто сообщить о необходимости проверки всего кода на предмет совместимости с CLS.

Роль библиотек базовых классов

В дополнение к спецификациям CLR, CTS и CLS платформа .NET предоставляет библиотеку базовых классов, которая доступна всем языкам программирования .NET. Эта библиотека не только инкапсулирует разнообразные примитивы, такие как потоки, файловый ввод-вывод, системы визуализации графики и механизмы взаимодействия с разнообразными внешними устройствами, но также обеспечивает поддержку для многочисленных служб, требуемых большинством реальных приложений.

Библиотеки базовых классов определяют типы, которые можно использовать для построения программных приложений любого вида. Например, инфраструктуру ASP.NET можно применять для создания веб-сайтов и служб REST, инфраструктуру WCF — для построения распределенных систем, инфраструктуру WPF — для написания настольных приложений с графическим пользовательским интерфейсом и т.д. Библиотеки базовых классов также предлагают типы для взаимодействия с XML-документами, каталогами и файловой системой заданного компьютера, для коммуникаций с реляционными базами данных (через ADO.NET) и т.п. С высокоуровневой точки зрения отношения между CLR, CTS, CLS и библиотеками базовых классов выглядят так, как показано на рис. 1.1.

Что привносит язык C#

Синтаксис языка программирования C# выглядит *очень* похожим на синтаксис языка Java. Однако называть C# клоном Java неправильно. В действительности и C#, и Java являются членами семейства языков программирования, основанных на C (например, C, Objective C, C++), поэтому они разделяют сходный синтаксис.

Рис. 1.1. Отношения между CLR, CTS, CLS и библиотеками базовых классов

Правда заключается в том, что многие синтаксические конструкции C# смоделированы в соответствии с разнообразными аспектами языков Visual Basic (VB) и C++. Например, подобно VB язык C# поддерживает понятия свойств класса (как противоположность традиционным методам извлечения и установки) и необязательных параметров. Подобно C++ язык C# позволяет перегружать операции, а также создавать структуры, перечисления и функции обратного вызова (посредством делегатов).

Более того, по мере проработки материала книги вы очень скоро заметите, что C# поддерживает множество средств, которые традиционно встречаются в различных языках функционального программирования (например, LISP или Haskell), скажем, лямбда-выражения и анонимные типы. Вдобавок, с появлением технологии LINQ (Language Integrated Query — язык интегрированных запросов) язык C# стал поддерживать конструкции, которые делают его довольно-таки уникальным в мире программирования. Но, несмотря на все это, наибольшее влияние на него оказали именно языки, основанные на C.

Из-за того, что C# представляет собой гибрид из нескольких языков, он является таким же синтаксически чистым (если не чище), как и Java, почти настолько же простым, как VB, и практически таким же мощным и гибким, как C++. Ниже приведен неполный список ключевых особенностей языка C#, которые характерны для всех его версий.

- Указатели необязательны! В программах на C# обычно не возникает потребность в прямых манипуляциях указателями (хотя в случае абсолютной необходимости можно опуститься и на этот уровень, как будет показано в главе 11).
- Автоматическое управление памятью посредством сборки мусора. Учитывая это, в C# не поддерживается ключевое слово `delete`.
- Формальные синтаксические конструкции для классов, интерфейсов, структур, перечислений и делегатов.
- Аналогичная языку C++ возможность перегрузки операций для специальных типов без особой сложности (например, обеспечение "возвращения `*this`, чтобы позволить связывать в цепочку" — не ваша забота).
- Поддержка программирования на основе атрибутов. Эта разновидность разработки позволяет аннотировать типы и их члены для дополнительного уточнения их поведения. Например, если пометить метод атрибутом `[Obsolete]`, то при попытке его использования программисты увидят ваше специальное предупреждение.

С выходом версии .NET 2.0 (примерно в 2005 году) язык программирования C# был обновлен для поддержки многочисленных новых функциональных возможностей, наиболее значимые из которых перечислены далее.

- Возможность создания обобщенных типов и обобщенных членов. Применяя обобщения, можно писать очень эффективный и безопасный к типам код, который определяет множество *заполнителей*, указываемых во время взаимодействия с обобщенными элементами.
- Поддержка анонимных методов, которые позволяют предоставлять встраиваемую функцию везде, где требуется тип делегата.
- Возможность определения одиночного типа в нескольких файлах кода (или при необходимости в виде представления в памяти) с использованием ключевого слова `partial`.

В версии .NET 3.5 (вышедшей приблизительно в 2008 году) в язык программирования C# была добавлена дополнительная функциональность, включая следующие средства.

- Поддержка строго типизированных запросов (например, LINQ), применяемых для взаимодействия с разнообразными формами данных. Запросы LINQ вы впервые встретите в главе 12.
- Поддержка анонимных типов, которые позволяют моделировать *устройство* типа, а не его поведение, на лету в коде.
- Возможность расширения функциональности существующего типа (не создавая его подклассы) с использованием расширяющих методов.
- Включение лямбда-операции (=>), которая еще больше упрощает работу с типами делегатов .NET.
- Новый синтаксис инициализации объектов, позволяющий устанавливать значения свойств во время создания объекта.

Версия .NET 4.0 (выпущенная в 2010 году) снова дополнила язык C# рядом средств, которые указаны ниже.

- Поддержка в методах необязательных параметров и именованных аргументов.
- Поддержка динамического поиска членов во время выполнения через ключевое слово `dynamic`. Как будет показано в главе 18, это обеспечивает универсальный подход к вызову членов на лету независимо от инфраструктуры, в которой они реализованы (COM, IronRuby, IronPython или службы рефлексии .NET).
- Работа с обобщенными типами стала намного понятнее, учитывая возможность легкого отображения обобщенных данных на универсальные коллекции `System.Object` через ковариантность и контравариантность.

В выпуске .NET 4.5 язык C# обрел пару новых ключевых слов (`async` и `await`), которые значительно упрощают многопоточное и асинхронное программирование. Если вы работали с предшествующими версиями C#, то можете вспомнить, что вызов методов через вторичные потоки требовал довольно большого объема малопонятного кода и применения разнообразных пространств имен .NET. Учитывая то, что теперь C# поддерживает языковые ключевые слова, которые автоматически устраняют эту сложность, процесс вызова методов асинхронным образом оказывается почти настолько же легким, как их вызов в синхронной манере. Данные темы детально раскрываются в главе 19.

Все это подводит нас к текущей версии языка C# и платформы .NET 4.6, где появилось несколько мелких средств, которые помогают упростить вашу кодовую базу.

По мере чтения книги вы их увидите, однако ниже представлен краткий обзор ряда новых возможностей, появившихся в C#.

- Встраиваемая инициализация для автоматических свойств, а также поддержка автоматических свойств, предназначенных только для чтения.
- Реализация однострочных методов с использованием лямбда-операции C#.
- Поддержка "статического импорта" для предоставления прямого доступа к статическим членам внутри пространства имен.
- `null`-условная операция, которая помогает проверять параметры на предмет `null` в реализации метода.
- Новый синтаксис форматирования строк, называемый *интерполяцией строк*.
- Возможность фильтрации исключений с применением нового ключевого слова `when`.

Сравнение управляемого и неуправляемого кода

Важно отметить, что язык C# может использоваться только для построения программного обеспечения, которое функционирует под управлением исполняющей среды .NET (применять C# для создания COM-сервера или неуправляемого приложения в стиле C/C++ не допускается). Выражаясь официально, для обозначения кода, ориентированного на исполняющую среду .NET, используется термин *управляемый код*. Двоичный модуль, который содержит управляемый код, называется *сборкой* (сборки более подробно рассматриваются далее в этой главе). В противоположность этому код, который не может напрямую обслуживаться исполняющей средой .NET, называется *неуправляемым кодом*.

Ранее упоминалось (и более подробно об этом пойдет речь позже в текущей и следующей главе), что платформа .NET способна функционировать в средах разнообразных операционных систем. Таким образом, вполне вероятно строить приложение C# на машине с применением Visual Studio и запускать его на машине Mac OS X с использованием исполняющей среды Mono .NET. Кроме того, приложение C# можно построить на машине Linux с помощью среды Xamarin Studio (глава 2) и запускать его под управлением Windows, Mac и т.д. Конечно, идея управляемой среды делает возможным построение, развертывание и запуск программ .NET на широком выборе целевых машин.

Другие языки программирования, ориентированные на .NET

Важно помнить, что C# не является единственным языком, который может применяться для построения приложений .NET. Среда Visual Studio изначально снабжает вас пятью управляемыми языками, а именно — C#, Visual Basic, C++/CLI, JavaScript и F#.

На заметку! F# — это язык .NET, основанный на синтаксисе функциональных языков. Наряду с тем, что F# может использоваться как чистый функциональный язык, он также располагает поддержкой конструкций объектно-ориентированного программирования (ООП) и библиотек базовых классов .NET. Дополнительные сведения об этом управляемом языке доступны на его официальной домашней странице по адресу `http://msdn.microsoft.com/fsharp`.

В дополнение к управляемым языкам, предлагаемым Microsoft, существуют компиляторы .NET для языков Smalltalk, Ruby, Python, COBOL и Pascal (и это неполный перечень). Хотя в настоящей книге внимание сосредоточено почти полностью на C#, на

следующей странице Википедии приведен большой список языков программирования, которые ориентированы на платформу .NET:

```
https://ru.wikipedia.org/wiki/Список_.NET-языков
```

Даже если вы интересуетесь главным образом построением программ .NET с применением синтаксиса C#, все равно рекомендуется посетить указанную страницу, т.к. вы наверняка сочтете многие языки для .NET заслуживающими дополнительного внимания (к примеру, LISP.NET).

Жизнь в многоязычном мире

Как только разработчики приходят к осознанию независимой от языка природы .NET, у них возникает множество вопросов. Самый распространенный из них может быть сформулирован так: если все языки .NET компилируются в управляемый код, то почему нам необходимо более одного языка/компилятора?

Отвечать на этот вопрос можно по-разному. Прежде всего, программисты бывают *очень* привередливы, когда дело касается выбора языка программирования. Некоторые предпочитают языки с многочисленными точками с запятой и фигурными скобками, но с минимальным количеством ключевых слов. Другим нравятся языки, предлагающие более *читабельные* синтаксические конструкции (такие как Visual Basic). Кто-то после перехода на платформу .NET желает задействовать имеющийся опыт работы на мэйнфреймах (выбрав компилятор COBOL.NET).

А теперь ответьте честно: если бы в Microsoft предложили единственный "официальный" язык .NET, выведенный на базе семейства BASIC, то все ли программисты были бы рады такому выбору? А если бы "официальный" язык .NET основывался на синтаксисе Fortran, то сколько людей в мире просто бы проигнорировали платформу .NET? Из-за того, что исполняющая среда .NET демонстрирует меньшую зависимость от языка, используемого для построения блока управляемого кода, программисты .NET могут сохранять свои синтаксические предпочтения и разделять скомпилированные сборки между коллегами, отделами и внешними организациями (безотносительно к тому, какой язык .NET выбрали для работы другие).

Еще одним превосходным побочным эффектом интеграции разнообразных языков .NET в единственное унифицированное программное решение является тот простой факт, что каждый язык программирования обладает своими сильными и слабыми сторонами. Например, некоторые языки программирования предлагают эффективную встроенную поддержку для реализации сложных математических вычислений. Другие языки лучше приспособлены для финансовых или логических вычислений, взаимодействия с мэйнфреймами и т.п. Когда преимущества конкретного языка программирования объединяются с преимуществами платформы .NET, то выигрывают все.

Конечно, в реальности довольно велики шансы на то, что большую часть времени вы будете уделять построению программного обеспечения, применяя предпочитаемый язык .NET. Однако, освоив синтаксис одного языка .NET, легко изучить другой. Это также очень выгодно, особенно для консультантов по разработке программного обеспечения. Например, если вашим предпочитаемым языком является C#, но вы находитесь на клиентской площадке, где все настроено на Visual Basic, то все равно сможете использовать функциональность .NET Framework и понимать общую структуру кодовой базы с минимальными усилиями.

Обзор сборок .NET

Независимо от того, какой язык .NET выбран для программирования, важно понимать, что хотя двоичные модули .NET имеют такое же файловое расширение, как

и неуправляемые двоичные компоненты Windows (`*.dll` или `*.exe`), внутренне они устроены совершенно по-другому. В частности, двоичные модули .NET содержат не специфические, а независимые от платформы инструкции на промежуточном языке (*Intermediate Language — IL*) и метаданные типов. На рис. 1.2 показано, как все это выглядит схематически.

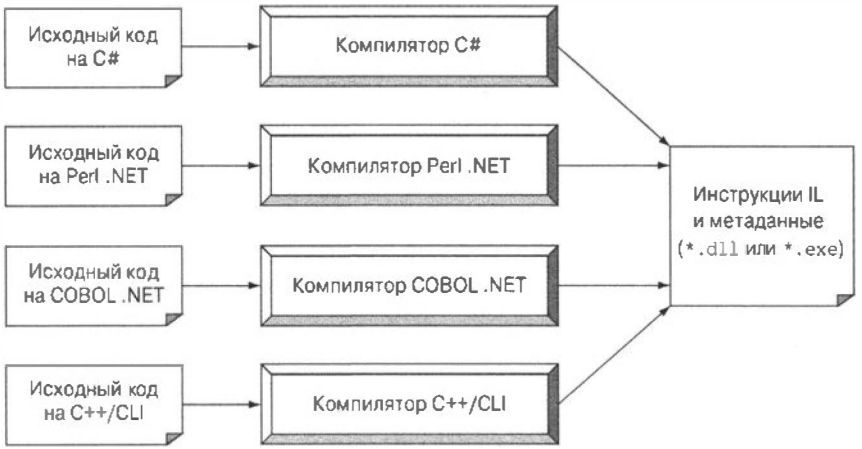


Рис. 1.2. Все компиляторы .NET генерируют инструкции IL и метаданные

На заметку! Относительно аббревиатуры IL следует сделать одно замечание. Данной аббревиатурой также обозначается промежуточный язык Microsoft (Microsoft Intermediate Language — MSIL) или общий промежуточный язык (Common Intermediate Language — CIL). Таким образом, при чтении литературы по .NET не забывайте о том, что IL, MSIL и CIL описывают в точности одну и ту же концепцию. В настоящей книге при ссылке на низкоуровневый набор инструкций будет применяться аббревиатура CIL.

Когда файл `*.dll` или `*.exe` был создан с использованием компилятора .NET, результирующий большой двоичный объект называется *сборкой*. Все многочисленные детали, касающиеся сборок .NET, подробно рассматриваются в главе 14. Тем не менее, для упрощения текущего обсуждения необходимо понимать некоторые основные свойства этого нового файлового формата.

Как упоминалось ранее, сборка содержит код CIL, который концептуально похож на байт-код Java тем, что не компилируется в специфичные для платформы инструкции до тех пор, пока это не станет абсолютно необходимым. Обычно "абсолютная необходимость" наступает тогда, когда на блок инструкций CIL (такой как реализация метода) производится ссылка с целью его применения исполняющей средой .NET.

В дополнение к инструкциям CIL сборки также содержат *метаданные*, которые детально описывают характеристики каждого "типа" внутри двоичного модуля. Например, если имеется класс по имени `SportsCar` , то метаданные типа представляют такие детали, как базовый класс `SportsCar`, реализуемые `SportsCar` интерфейсы (если есть) и полные описания всех членов, поддерживаемых `SportsCar`. Метаданные .NET всегда присутствуют внутри сборки и автоматически генерируются компилятором языка .NET.

Наконец, помимо инструкций CIL и метаданных типов сами сборки также описываются с помощью метаданных, которые официально называются манифестом. Манифест содержит информацию о текущей версии сборки, сведения о культуре (используемые для локализации строковых и графических ресурсов) и список ссылок на все внешние

сборки, которые требуются для правильного функционирования. Разнообразные инструменты, которые можно применять для исследования типов, метаданных и манифестов сборок, рассматриваются в нескольких последующих главах.

Роль языка CIL

Теперь давайте исследуем код CIL, метаданные типов и манифест сборки более детально. CIL — это язык, который находится выше любого набора инструкций, специфичных для конкретной платформы. Например, приведенный ниже код C# моделирует простой калькулятор. Не углубляясь пока в подробности синтаксиса, обратите внимание на формат метода `Add()` в классе `Calc`.

```
// Calc.cs
using System;
namespace CalculatorExample
{
  // Этот класс содержит точку входа приложения.
  class Program
  {
    static void Main()
    {
      Calc c = new Calc();
      int ans = c.Add(10, 84);
      Console.WriteLine("10 + 84 is {0}.", ans);
      // Ожидать нажатия пользователем клавиши <Enter> перед завершением работы.
      Console.ReadLine();
    }
  }
  // Калькулятор C#.
  class Calc
  {
    public int Add(int x, int y)
    { return x + y; }
  }
}
```

В результате компиляции этого файла кода с помощью компилятора C# (`csc.exe`) получается однофайловая сборка `*.exe`, которая содержит манифест, инструкции CIL и метаданные, описывающие каждый аспект классов `Calc` и `Program`.

На заметку! В главе 2 будет показано, как использовать графические IDE-среды (такие как Visual Studio Community Edition) для компиляции файлов кода.

Например, открыв полученную сборку в утилите `ildasm.exe` (которая рассматривается далее в этой главе), вы обнаружите, что метод `Add()` был представлен в CIL следующим образом:

```
.method public hidebysig instance int32 Add(int32 x,
   int32 y) cil managed
{
  // Code size 9 (0x9)
  // Размер кода 9 (0x9)
  .maxstack 2
  .locals init (int32 V_0)
  IL_0000: nop
  IL_0001: ldarg.1
```

```
  IL_0002: ldarg.2
  IL_0003: add
  IL_0004: stloc.0
  IL_0005: br.s IL_0007
  IL_0007: ldloc.0
  IL_0008: ret
} // end of method Calc::Add
  // конец метода Calc::Add
```

Не стоит беспокоиться, если итоговый код CIL этого метода выглядит непонятным — в главе 18 будут описаны базовые аспекты языка программирования CIL. Важно понимать, что компилятор C# выпускает CIL-код, а не инструкции, специфичные для платформы. Теперь вспомните, что это справедливо для всех компиляторов .NET. В целях иллюстрации создадим то же самое приложение на языке Visual Basic вместо C#:

```
' Calc.vb
Imports System

Namespace CalculatorExample
  ' "Модуль" VB - это класс, который
  ' содержит только статические члены.
  Module Program
    Sub Main()
      Dim c As New Calc
      Dim ans As Integer = c.Add(10, 84)
      Console.WriteLine("10 + 84 is {0}.", ans)
      Console.ReadLine()
    End Sub
  End Module

  Class Calc
    Public Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
      Return x + y
    End Function
  End Class
End Namespace
```

Просмотрев код CIL метода `Add()`, можно найти похожие инструкции (слегка скорректированные компилятором Visual Basic, `vbc.exe`):

```
.method public instance int32 Add(int32 x,
  int32 y) cil managed
{
  // Code size 8 (0x8)
  // Размер кода 8 (0x8)
  .maxstack 2
  .locals init (int32 V_0)
  IL_0000: ldarg.1
  IL_0001: ldarg.2
  IL_0002: add.ovf
  IL_0003: stloc.0
  IL_0004: br.s IL_0006
  IL_0006: ldloc.0
  IL_0007: ret
} // end of method Calc::Add
  // конец метода Calc::Add
```

Исходный код. Файлы с кодом `Calc.cs` и `Calc.vb` находятся в подкаталоге `Chapter_1`.

Преимущества языка CIL

В этот момент вас может заинтересовать, какую выгоду приносит компиляция исходного кода в CIL, а не напрямую в специфичный набор инструкций. Одним из преимуществ является языковая интеграция. Как вы уже видели, все компиляторы .NET генерируют практически идентичные инструкции CIL. Следовательно, все языки способны взаимодействовать в рамках четко определенной "двоичной арены".

Более того, учитывая независимость от платформы языка CIL, сама инфраструктура .NET Framework не зависит от платформы и обеспечивает те же самые преимущества, к которым так привыкли разработчики на Java (например, единую кодовую базу, функционирующую в средах разных операционных систем). В действительности для языка C# предусмотрен международный стандарт, и уже существует крупное подмножество платформы .NET и реализации для многих операционных систем, отличающихся от Windows (более подробно об этом речь пойдет в конце главы).

Компиляция кода CIL в инструкции, специфичные для платформы

Поскольку сборки содержат инструкции CIL, а не инструкции, специфичные для платформы, перед применением код CIL должен компилироваться на лету. Компонентом, который транслирует код CIL в содержательные инструкции центрального процессора (ЦП), является оперативный (just-in-time — JIT) компилятор (иногда его называют *Jitter*). Для каждого целевого ЦП в исполняющей среде .NET имеется JIT-компилятор, оптимизированный под лежащую в основе платформу.

Например, если строится приложение .NET, предназначенное для развертывания на карманном устройстве (таком как мобильное устройство Windows), то соответствующий JIT-компилятор будет оснащен возможностями запуска в среде с ограниченным объемом памяти. С другой стороны, если сборка развертывается на внутреннем сервере компании (где память редко представляет собой проблему), то JIT-компилятор будет оптимизирован для функционирования в среде с большим объемом памяти. Таким образом, разработчики могут писать единственный блок кода, который способен эффективно транслироваться JIT-компилятором и выполняться на машинах с разной архитектурой.

Более того, при трансляции инструкций CIL в соответствующий машинный код JIT-компилятор будет кешировать результаты в памяти в манере, подходящей для целевой операционной системы. В этом случае, если производится вызов метода по имени `PrintDocument()`, то инструкции CIL компилируются в специфичные для платформы инструкции при первом вызове и остаются в памяти для более позднего использования. Благодаря этому при вызове метода `PrintDocument()` в следующий раз повторная компиляция инструкций CIL не понадобится.

На заметку! Можно также выполнять "предварительную JIT-компиляцию" сборки во время установки приложения с помощью инструмента командной строки `ngen.exe`, который поставляется вместе с .NET Framework SDK. Это позволяет улучшить показатели времени запуска для графически насыщенных приложений.

Роль метаданных типов .NET

В дополнение к инструкциям CIL сборка .NET содержит полные и точные метаданные, которые описывают каждый определенный в двоичном модуле тип (например, класс, структуру, перечисление), а также члены каждого типа (например, свойства, методы, события). К счастью, за выпуск актуальных метаданных типов всегда отвечает компилятор, а не программист. Из-за того, что метаданные .NET настолько основательны, сборки являются целиком самоописательными сущностями.

Чтобы проиллюстрировать формат метаданных типов .NET, давайте взглянем на метаданные, которые были сгенерированы для исследуемого ранее метода `Add()` класса `Calc`, написанного на C# (метаданные для версии Visual Basic метода `Add()` аналогичны; дополнительные сведения о работе с утилитой `ildasm` будут приведены чуть позже):

```
TypeDef #2 (02000003)
-------------------------------------------------------
  TypDefName: CalculatorExample.Calc (02000003)
  Flags     : [NotPublic] [AutoLayout] [Class]
  [AnsiClass] [BeforeFieldInit] (00100001)
  Extends   : 01000001 [TypeRef] System.Object
  Method #1 (06000003)
-------------------------------------------------------
  MethodName: Add (06000003)
  Flags     : [Public] [HideBySig] [ReuseSlot] (00000086)
  RVA       : 0x00002090
  ImplFlags : [IL] [Managed] (00000000)
  CallCnvntn: [DEFAULT]
  hasThis
  ReturnType: I4
    2 Arguments
    Argument #1:  I4
    Argument #2:  I4
    2 Parameters
    (1) ParamToken : (08000001) Name : x flags: [none] (00000000)
    (2) ParamToken : (08000002) Name : y flags: [none] (00000000)
```

Метаданные применяются многочисленными аспектами исполняющей среды .NET, а также разнообразными инструментами разработки. Скажем, средство IntelliSense, предоставляемое такими инструментами, как Visual Studio, стало возможным благодаря чтению метаданных сборки во время проектирования. Метаданные также используются различными утилитами для просмотра объектов, инструментами отладки и самим компилятором C#. Бесспорно, метаданные являются принципиальной основой многочисленных технологий .NET, включая Windows Communication Foundation (WCF), рефлексию, позднее связывание и сериализацию объектов. Роль метаданных .NET будет раскрыта в главе 15.

Роль манифеста сборки

Последний, но не менее важный момент: вспомните, что сборка .NET также содержит метаданные, которые описывают ее саму (формально называемые *манифестом*). Кроме прочего манифест документирует все внешние сборки, которые требуются текущей сборке для ее корректного функционирования, номер версии сборки, информацию об авторских правах и т.д. Подобно метаданным типов, за генерацию манифеста сборки всегда отвечает компилятор. Ниже представлены некоторые существенные детали манифеста, сгенерированного при компиляции показанного ранее в главе файла кода `Calc.cs` (предполагается, что вы проинструктировали компилятор о назначении сборке имени `Calc.exe`):

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
```

```
.assembly Calc
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Calc.exe
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 0x00000200
.corflags 0x00000001
```

Выражаясь кратко, этот манифест документирует набор внешних сборок, требуемых для `Calc.exe` (в директиве `.assembly extern`), а также разнообразные характеристики самой сборки (к примеру, номер версии и имя модуля). Полезность данных манифеста будет более подробно исследоваться в главе 14.

Понятие общей системы типов (CTS)

Сборка может содержать любое количество различающихся типов. В мире .NET *тип* — это просто общий термин, применяемый для ссылки на член из набора (класс, интерфейс, структура, перечисление, делегат). При построении решений на любом языке .NET, скорее всего, придется взаимодействовать со многими такими типами. Например, в сборке может быть определен класс, реализующий некоторое число интерфейсов. Возможно, метод одного из интерфейсов принимает перечисление в качестве входного параметра и возвращает вызывающему компоненту структуру.

Вспомните, что CTS является формальной спецификацией, которая документирует, каким образом типы должны быть определены, чтобы они могли обслуживаться средой CLR. Внутренние детали CTS обычно интересуют только тех, кто занимается построением инструментов и/или компиляторов, предназначенных для платформы .NET. Однако для всех программистов .NET важно знать о том, как работать с пятью типами, определенными в CTS, на выбранных ими языках. Ниже приведен краткий обзор.

Типы классов CTS

Каждый язык .NET поддерживает, по меньшей мере, понятие *типа класса*, которое является краеугольным камнем ООП. Класс может состоять из любого количества членов (таких как конструкторы, свойства, методы и события) и элементов данных (полей). В языке C# классы объявляются с использованием ключевого слова `class`, примерно так:

```
// Тип класса C# с одним методом.
class Calc
{
  public int Add(int x, int y)
  {
    return x + y;
  }
}
```

Формальное знакомство с построением типов классов в C# начнется в главе 5, а пока в табл. 1.1 приведен перечень характеристик, свойственных типам классов.

Таблица 1.1. Характеристики классов CTS

Характеристика класса	Практический смысл
Является ли класс запечатанным?	Запечатанные классы не могут выступать в качестве базовых для других классов
Реализует ли класс какие-то интерфейсы?	Интерфейс — это коллекция абстрактных членов, которая предоставляет контракт между объектом и пользователем объекта. Система CTS позволяет классу реализовывать любое количество интерфейсов
Является ли класс абстрактным или конкретным?	Абстрактные классы не допускают прямого создания экземпляров и предназначены для определения общего поведения производных типов. Экземпляры конкретных классов могут создаваться напрямую
Какова видимость класса?	Каждый класс должен конфигурироваться с ключевым словом видимости, таким как `public` или `internal`. По существу оно управляет тем, может ли класс использоваться во внешних сборках или же только внутри определяющей его сборки

Типы интерфейсов CTS

Интерфейсы — это всего лишь именованные коллекции определений абстрактных членов, которые могут поддерживаться (т.е. быть реализованными) в заданном классе или структуре. В языке C# типы интерфейсов определяются с применением ключевого слова `interface`. По соглашению имена всех интерфейсов .NET начинаются с прописной буквы `I`, как показано в следующем примере:

```
// Тип интерфейса C# обычно объявляется как
// public, чтобы позволить типам из других
// сборок реализовывать его поведение.
public interface IDraw
{
  void Draw();
}
```

Сами по себе интерфейсы приносят немного пользы. Тем не менее, когда класс или структура реализует выбранный интерфейс уникальным образом, появляется возможность получать доступ к предоставленной функциональности, используя ссылку на этот интерфейс в полиморфной манере. Программирование на основе интерфейсов подробно рассматривается в главе 8.

Типы структур CTS

Концепция структуры также формализована в CTS. Если вы имели дело с языком C, то вас наверняка обрадует, что эти определяемые пользователем типы (user-defined type — UDT) сохранились в мире .NET (хотя их внутренне поведение несколько изменилось). Попросту говоря, *структуру* можно считать легковесным типом класса, который имеет семантику, основанную на значении. Тонкости структур более подробно исследуются в главе 4. Обычно структуры лучше всего подходят для моделирования геометрических и математических данных и создаются в языке C# с применением ключевого слова `struct`, например:

```
// Тип структуры C#.
struct Point
{
```

```
  // Структуры могут содержать поля.
  public int xPos, yPos;

  // Структуры могут содержать параметризованные конструкторы.
  public Point(int x, int y)
  { xPos = x; yPos = y;}

  // Структуры могут определять методы.
  public void PrintPosition()
  {
    Console.WriteLine("({0}, {1})", xPos, yPos);
  }
}
```

Типы перечислений CTS

Перечисления — это удобная программная конструкция, которая позволяет группировать пары "имя-значение". Например, предположим, что требуется создать игровое приложение, в котором игроку бы позволялось выбирать персонажа из трех категорий: Wizard (маг), Fighter (воин) или Thief (вор). Вместо отслеживания простых числовых значений, представляющих каждую категорию, можно было бы создать строго типизированное перечисление, используя ключевое слово `enum`:

```
// Тип перечисления C#.
enum CharacterType
{
  Wizard = 100,
  Fighter = 200,
  Thief = 300
}
```

По умолчанию для хранения каждого элемента выделяется блок памяти, соответствующий 32-битному целому, однако при необходимости (скажем, при программировании для устройств с малым объемом памяти наподобие мобильных устройств) область хранения можно изменить. Кроме того, спецификация CTS требует, чтобы перечислимые типы были производными от общего базового класса `System.Enum`. Как будет показано в главе 4, в этом базовом классе определено несколько интересных членов, которые позволяют извлекать, манипулировать и преобразовывать лежащие в основе пары "имя-значение" программным образом.

Типы делегатов CTS

Делегаты являются .NET-эквивалентом безопасных к типам указателей на функции в стиле C. Основная разница в том, что делегат .NET представляет собой *класс*, производный от `System.MulticastDelegate`, а не простой указатель на низкоуровневый адрес в памяти. В языке C# делегаты объявляются с помощью ключевого слова `delegate`:

```
// Этот тип делегата C# может "указывать" на любой метод,
// возвращающий тип int и принимающий два значения int.
delegate int BinaryOp(int x, int y);
```

Делегаты критически важны, когда необходимо обеспечить объект возможностью перенаправления вызова другому объекту, и они формируют основу архитектуры событий .NET. Как будет показано в главах 10 и 19, делегаты обладают внутренней поддержкой группового вызова (т.е. перенаправления запроса множеству получателей) и асинхронного вызова методов (т.е. вызова методов во вторичном потоке).

Члены типов CTS

Теперь, когда было представлено краткое описание каждого типа, формализованного в CTS, следует осознать тот факт, что большинство этих типов располагает любым количеством *членов*. Формально член типа ограничен набором {конструктор, финализатор, статический конструктор, вложенный тип, операция, метод, свойство, индексатор, поле, поле только для чтения, константа, событие}.

В спецификации CTS определены разнообразные *характеристики*, которые могут быть ассоциированы с заданным членом. Например, каждый член может иметь характеристику видимости (например, открытый, закрытый или защищенный). Некоторые члены могут быть объявлены как абстрактные (чтобы обеспечить полиморфное поведение в производных типах) или как виртуальные (чтобы определить заготовленную, но допускающую переопределение реализацию). Вдобавок большинство членов могут быть сконфигурированы как статические (связанные с уровнем класса) или члены экземпляра (связанные с уровнем объекта). Создание членов типов будет описано в нескольких последующих главах.

На заметку! Как будет показано в главе 9, язык C# также поддерживает создание обобщенных типов и обобщенных членов.

Встроенные типы данных CTS

Финальный аспект спецификации CTS, который следует знать на текущий момент, заключается в том, что она устанавливает четко определенный набор фундаментальных типов данных. Хотя в каждом отдельном языке для объявления фундаментального типа данных обычно имеется уникальное ключевое слово, ключевые слова всех языков .NET в конечном итоге распознаются как один и тот же тип CTS, определенный в сборке по имени `mscorlib.dll`. В табл. 1.2 показано, каким образом основные типы данных CTS выражаются в различных языках .NET.

Таблица 1.2. Встроенные типы данных CTS

Тип данных CTS	Ключевое слово VB	Ключевое слово C#	Ключевое слово C++/CLI
`System.Byte`	`Byte`	`byte`	`unsigned char`
`System.SByte`	`SByte`	`sbyte`	`signed char`
`System.Int16`	`Short`	`short`	`short`
`System.Int32`	`Integer`	`int`	`int` или `long`
`System.Int64`	`Long`	`long`	`__int64`
`System.UInt16`	`UShort`	`ushort`	`unsigned short`
`System.UInt32`	`UInteger`	`uint`	`unsigned int` или `unsigned long`
`System.UInt64`	`ULong`	`ulong`	`unsigned __int64`
`System.Single`	`Single`	`float`	`float`
`System.Double`	`Double`	`double`	`double`
`System.Object`	`Object`	`object`	`object^`
`System.Char`	`Char`	`char`	`wchar_t`
`System.String`	`String`	`string`	`String^`
`System.Decimal`	`Decimal`	`decimal`	`Decimal`
`System.Boolean`	`Boolean`	`bool`	`bool`

Учитывая, что уникальные ключевые слова в управляемом языке являются просто сокращенными обозначениями для реальных типов в пространстве имен System, больше не нужно беспокоиться об условиях переполнения/потери значимости для числовых данных или о том, как внутренне представляются строки и булевские значения в разных языках. Взгляните на следующие фрагменты кода, в которых определяются 32-битные целочисленные переменные в C# и Visual Basic с применением ключевых слов языка, а также формального типа данных CTS:

```
// Определение целочисленных переменных в C#.
int i = 0;
System.Int32 j = 0;

' Определение целочисленных переменных в VB.
Dim i As Integer = 0
Dim j As System.Int32 = 0
```

Понятие общеязыковой спецификации (CLS)

Как вы уже знаете, разные языки программирования выражают одни и те же программные конструкции с помощью уникальных и специфичных для конкретного языка терминов. Например, в C# конкатенация строк обозначается с использованием операции "плюс" (+), а в VB для этого обычно применяется амперсанд (&). Даже если два разных языка выражают одну и ту же программную идиому (скажем, функцию, не возвращающую значение), то высока вероятность того, что синтаксис на первый взгляд будет выглядеть не сильно отличающимся:

```
// Ничего не возвращающий метод C#.
public void MyMethod()
{
  // Некоторый код...
}
' Ничего не возвращающий метод VB.
Public Sub MyMethod()
  ' Некоторый код...
End Sub
```

Вы уже видели ранее, что такие небольшие синтаксические вариации для исполняющей среды .NET несущественны, учитывая, что соответствующие компиляторы (в этом случае — csc.exe и vbc.exe) выпускают сходный набор инструкций CIL. Тем не менее, языки могут также отличаться в отношении общего уровня функциональности. Например, язык .NET может иметь или не иметь ключевое слово для представления данных без знака и поддерживать или не поддерживать типы указателей. При таких возможных вариациях было бы идеально располагать опорными требованиями, которым бы удовлетворяли все языки, ориентированные на .NET.

Спецификация CLS — это набор правил, подробно описывающих минимальное и полное множество характеристик, которые отдельный компилятор .NET должен поддерживать, чтобы генерировать код, обслуживаемый средой CLR и в то же время доступный в унифицированной манере всем языкам, которые ориентированы на платформу .NET. Во многих отношениях CLS можно рассматривать как *подмножество* полной функциональности, определенной в CTS.

В конечном итоге CLS является набором правил, которых должны придерживаться создатели компиляторов, если они намерены обеспечить гладкое функционирование своих продуктов в мире .NET. Каждое правило имеет простое название (например, "Правило номер 6"), и каждое правило описывает воздействие на тех, кто строит компиляторы, и на тех, кто (каким-либо образом) взаимодействует с ними.

Самым важным в CLS является правило номер 1.

- *Правило номер 1.* Правила CLS применяются только к тем частям типа, которые видны извне определяющей сборки.

Из этого правила можно сделать корректный вывод о том, что остальные правила CLS не применяются к логике, используемой для построения внутренних рабочих деталей типа .NET. Единственными аспектами типа, которые должны быть согласованы с CLS, являются сами определения членов (т.е. соглашения об именовании, параметры и возвращаемые типы). В рамках логики реализации члена может применяться любое количество приемов, не соответствующих CLS, т.к. для внешнего мира это не играет никакой роли.

В целях иллюстрации ниже представлен метод `Add()` в C#, который не совместим с CLS, поскольку его параметры и возвращаемое значение используют данные без знака (что не является требованием CLS):

```
class Calc
{
  // Открытые для доступа данные без знака не совместимы с CLS!
  public ulong Add(ulong x, ulong y)
  {
    return x + y;
  }
}
```

Тем не менее, если просто работать с данными без знака внутри метода, как в следующем примере:

```
class Calc
{
  public int Add(int x, int y)
  {
    // Поскольку эта переменная ulong используется только
    // внутренне, совместимость с CLS сохраняется.
    ulong temp = 0;
    ...
    return x + y;
  }
}
```

то правила CLS по-прежнему соблюдены и все языки .NET смогут обращаться к такому методу `Add()`.

Разумеется, помимо "Правила номер 1" в спецификации CLS определено множество других правил. Например, в CLS описано, каким образом заданный язык должен представлять текстовые строки, как внутренне представлять перечисления (базовый тип, применяемый для хранения их значений), каким образом определять статические члены и т.д. К счастью, для того, чтобы стать умелым разработчиком .NET, запоминать все эти правила вовсе не обязательно. В общем и целом глубоко разбираться в спецификациях CTS и CLS обычно должны только создатели инструментов и компиляторов.

Обеспечение совместимости с CLS

Как вы увидите при чтении книги, в языке C# определено несколько программных конструкций, несовместимых с CLS. Однако хорошая новость заключается в том, что компилятор C# можно инструктировать о необходимости проверки кода на предмет совместимости с CLS, используя единственный атрибут .NET:

```
// Сообщить компилятору C# о том, что он должен осуществлять
// проверку на совместимость с CLS.
[assembly: CLSCompliant(true)]
```

Детали программирования на основе атрибутов более подробно рассматриваются в главе 15. А пока следует просто запомнить, что атрибут `[CLSCompliant]` заставляет компилятор C# проверять каждую строку кода на соответствие правилам CLS. В случае обнаружения любых нарушений спецификации CLS компилятор сообщит об ошибке и выдаст описание проблемного кода.

Понятие общеязыковой исполняющей среды (CLR)

Помимо спецификаций CTS и CLS для получения общей картины на данный момент осталось рассмотреть еще одну аббревиатуру — CLR. С точки зрения программирования термин *исполняющая среда* можно понимать как коллекцию служб, которые требуются для выполнения скомпилированной единицы кода. Например, когда разработчики на Java развертывают программное обеспечение на новом компьютере, им необходимо удостовериться в том, что на компьютере установлена виртуальная машина Java (Java Virtual Machine — JVM), которая обеспечит выполнение их программного обеспечения.

Платформа .NET предлагает еще одну исполняющую среду. Основное отличие исполняющей среды .NET от упомянутых выше сред заключается в том, что исполняющая среда .NET обеспечивает единый четко определенный уровень выполнения, который разделяется *всеми* языками и платформами, ориентированными на .NET.

Главный механизм CLR физически представлен библиотекой по имени `mscoree.dll` (также известной как общий механизм выполнения исполняемого кода объектов (Common Object Runtime Execution Engine)). Когда на сборку производится ссылка для ее применения, библиотека `mscoree.dll` загружается автоматически и в свою очередь загружает требуемую сборку в память. Исполняющая среда отвечает за решение нескольких задач. Прежде всего, она является агентом, который занимается определением местоположения сборки и нахождением запрошенного типа в двоичном модуле за счет чтения содержащихся в нем метаданных. Затем среда CLR размещает тип в памяти, компилирует ассоциированный код CIL в специфичные для платформы инструкции, производит все необходимые проверки безопасности и после этого выполняет нужный код.

В дополнение к загрузке специальных сборок и созданию специальных типов среда CLR будет также взаимодействовать с типами, содержащимися в библиотеках базовых классов .NET, когда это требуется. Хотя полная библиотека базовых классов разделена на ряд обособленных сборок, главной сборкой считается `mscorlib.dll`, которая содержит большое количество основных типов, инкапсулирующих широкий спектр распространенных задач программирования, а также основные типы данных, используемые во всех языках .NET. При построении решений .NET доступ к этой сборке предоставляется автоматически.

На рис. 1.3 показан высокоуровневый рабочий поток, возникающий между исходным кодом (в котором применяются типы из библиотеки базовых классов), заданным компилятором .NET и исполняющей средой .NET.

Различия между сборками, пространствами имен и типами

Любой из нас понимает важность библиотек кода. Главное назначение библиотек платформы — предоставлять разработчикам четко определенный набор готового кода, который можно задействовать в создаваемых приложениях.

Рис. 1.3. Сборка `mscoree.dll` в действии

Однако C# не поставляется с какой-то специфичной для языка библиотекой кода. Вместо этого разработчики на C# используют нейтральные к языкам библиотеки .NET. Для поддержания всех типов внутри библиотек базовых классов в организованном виде в рамках платформы .NET широко применяется концепция *пространств имен*.

Пространство имен — это группа семантически родственных типов, которые содержатся в одной или нескольких связанных друг с другом сборках. Например, пространство имен `System.IO` содержит типы, относящиеся к файловому вводу-выводу, пространство имен `System.Data` — типы для работы с базами данных и т.д. Важно понимать, что в одной сборке (такой как `mscorlib.dll`) может содержаться любое количество пространств имен, каждое из которых может иметь любое число типов.

Для прояснения на рис. 1.4 приведен экранный снимок окна браузера объектов Visual Studio (доступного через меню View (Вид)). Этот инструмент позволяет просматривать сборки, на которые имеются ссылки в текущем проекте, пространства имен внутри отдельной сборки, типы в конкретном пространстве имени и члены специфи-

ческого типа. Обратите внимание, что сборка `mscorlib.dll` содержит много разных пространств имен (вроде `System.IO`), каждое с собственными семантически связанными типами (например, `BinaryReader`).

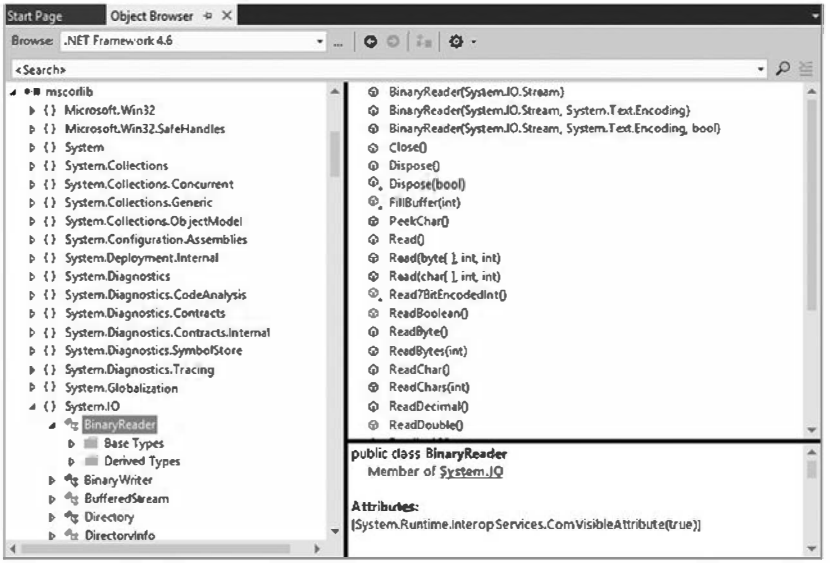


Рис. 1.4. Одиночная сборка может иметь любое количество пространств имен, а пространство имен может содержать любое число типов

Основное отличие между таким подходом и специфичной для языка библиотекой заключается в том, что любой язык, ориентированный на исполняющую среду .NET, использует *те же самые* пространства имен и *те же самые* типы. Например, ниже показан код вездесущего приложения "Hello World" на языках C#, VB и C++/CLI:

```
// Приложение "Hello World" на языке C#.
using System;

public class MyApp
{
  static void Main()
  {
    Console.WriteLine("Hi from C#");
  }
}
' Приложение "Hello World" на языке VB.
Imports System
Public Module MyApp
  Sub Main()
    Console.WriteLine("Hi from VB")
  End Sub
End Module
// Приложение "Hello World" на языке C++/CLI
#include "stdafx.h"
using namespace System;
int main(array<System::String ^> ^args)
{
  Console::WriteLine(L"Hi from C++/CLI");
  return 0;
}
```

Обратите внимание, что в каждом языке применяется класс `Console`, определенный в пространстве имен `System`. Помимо очевидных синтаксических различий эти три приложения выглядят довольно похожими как физически, так и логически.

Понятно, что после освоения выбранного языка программирования для .NET вашей следующей целью как разработчика в .NET будет освоение изобилия типов, определенных в многочисленных пространствах имен .NET. Наиболее фундаментальное пространство имен, с которого нужно начать, называется `System`. Это пространство имен предлагает основной набор типов, которые вам как разработчику в .NET придется задействовать неоднократно. В действительности без добавления, по крайней мере, ссылки на пространство имен `System` построить какое-либо функциональное приложение C# невозможно, т.к. в `System` определены основные типы данных (например, `System.Int32` и `System.String`). В табл. 1.3 приведены краткие описания некоторых (конечно же, не всех) пространств имен .NET, сгруппированных по функциональности.

Таблица 1.3. Избранные пространства имен .NET

Пространство имен .NET	Описание
`System`	Внутри пространства имен `System` содержится множество полезных типов, предназначенных для работы с внутренними данными, математическими вычислениями, генерацией случайных чисел, переменными среды и сборкой мусора, а также ряд распространенных исключений и атрибутов
`System.Collections` `System.Collections.Generic`	В этих пространствах имен определен набор контейнерных типов, а также базовые типы и интерфейсы, которые позволяют строить настраиваемые коллекции

`System.Data`
`System.Data.Common`
`System.Data.EntityClient`
`System.Data.SqlClient`
	Эти пространства имен используются для взаимодействия с базами данных через ADO.NET

`System.IO`
`System.IO.Compression`
`System.IO.Ports`
	В этих пространствах имен определены многочисленные типы, предназначенные для работы с файловым вводом-выводом, сжатием данных и портами

`System.Reflection` `System.Reflection.Emit`	В этих пространствах имен определены типы, которые поддерживают обнаружение типов во время выполнения, а также динамическое создание типов
`System.Runtime.InteropServices`	Это пространство имен предоставляет средства, позволяющие типам .NET взаимодействовать с неуправляемым кодом (например, DLL-библиотеками на основе C и серверами COM) и наоборот
`System.Drawing` `System.Windows.Forms`	В этих пространствах имен определены типы, применяемые при построении настольных приложений с использованием исходного инструментального набора .NET для создания пользовательских интерфейсов (Windows Forms)

`System.Windows`
`System.Windows.Controls`
`System.Windows.Shapes`
	Пространство имен `System.Windows` является корневым для нескольких пространств имен, которые представляют инструментальный набор для построения пользовательских интерфейсов Windows Presentation Foundation (WPF)

Окончание табл. 1.3

Пространство имен .NET	Описание

`System.Linq`
`System.Xml.Linq`
`System.Data.DataSetExtensions`
	В этих пространствах имен определены типы, применяемые при программировании с использованием API-интерфейса LINQ

`System.Web`	Это одно из многих пространств имен, которые позволяют строить веб-приложения ASP.NET
`System.Web.Http`	Это одно из многих пространств имен, которые позволяют строить веб-службы REST
`System.ServiceModel`	Это одно из многих пространств имен, применяемых для построения распределенных приложений с использованием API-интерфейса Windows Communication Foundation (WCF)
`System.Workflow.Runtime` `System.Workflow.Activities`	Это два из множества пространств имен, определяющих типы, которые применяются при построении приложений, поддерживающих рабочие потоки, с использованием API-интерфейса Windows Workflow Foundation (WF)
`System.Threading` `System.Threading.Tasks`	В этих пространствах имен определены многочисленные типы для построения многопоточных приложений, которые могут распределять рабочую нагрузку по нескольким ЦП
`System.Security`	Безопасность является неотъемлемым аспектом мира .NET. В пространствах имен, связанных с безопасностью, содержится множество типов, которые позволяют работать с разрешениями, криптографией и т.д.
`System.Xml`	В пространстве имен, относящемся к XML, определены многочисленные типы, применяемые для взаимодействия с XML-данными

Роль корневого пространства имен `Microsoft`

Во время изучения списка, представленного в табл. 1.3, вы наверняка заметили, что `System` является корневым пространством имен для большинства вложенных пространств имен (таких как `System.IO`, `System.Data` и т.д.). Однако оказывается, что помимо `System` в библиотеке базовых классов определено несколько корневых пространств имен наивысшего уровня, наиболее полезное из которых называется `Microsoft`.

Любое пространство имен, вложенное внутрь `Microsoft` (скажем, `Microsoft.CSharp`, `Microsoft.ManagementConsole`, `Microsoft.Win32`), содержит типы, которые используются для взаимодействия со службами, уникальными для операционной системы Windows. Учитывая это, вы не должны предполагать, что эти типы моли бы успешно применяться в других операционных системах, поддерживающих .NET, таких как Mac OS X. По большей части в настоящей книге мы не будем углубляться в детали пространств имен, для которых `Microsoft` является корневым, поэтому обращайтесь в документацию .NET Framework 4.6 SDK, если вам интересно.

На заметку! В главе 2 будет показано, как работать с документацией .NET Framework 4.6 SDK, в которой содержатся подробные описания всех пространств имен, типов и членов внутри библиотек базовых классов.

Доступ к пространству имен программным образом

Полезно снова повторить, что пространство имен — это всего лишь удобный способ логической организации связанных типов, содействующий их пониманию. Давайте еще раз обратимся к пространству имен `System`. С точки зрения разработчика можно предположить, что конструкция `System.Console` представляет класс по имени `Console`, который содержится внутри пространства имен под названием `System`. Однако с точки зрения исполняющей среды .NET это не так. Исполняющая среда видит только одиночный класс по имени `System.Console`.

В языке C# ключевое слово `using` упрощает процесс ссылки на типы, определенные в отдельном пространстве имен. Рассмотрим, каким образом оно работает. Пусть требуется построить графическое настольное приложение с использованием API-интерфейса WPF. Хотя освоение типов в каждом пространстве имен предполагает изучение и экспериментирование, ниже приведен ряд возможных кандидатов на ссылку из такого приложения:

```
// Некоторые возможные пространства имен,
// применяемые при построении приложения WPF.
using System;                   // Общие типы из библиотек базовых классов.
using System.Windows.Shapes;    // Типы для графической визуализации.
using System.Windows.Controls; // Типы виджетов графического
                                // пользовательского интерфейса Windows Forms.
using System.Data;              // Общие типы, связанные с данными.
using System.Data.SqlClient;    // Типы доступа к данным MS SQL Server.
```

После указания нескольких необходимых пространств имен (и установки ссылки на сборки, где они определены) можно свободно создавать экземпляры типов, которые в них содержатся. Например, если нужно создать экземпляр класса `Button` (определенного в пространстве имен `System.Windows.Controls`), то можно написать следующий код:

```
// Явно перечислить пространства имен, используемые в этом файле.
using System;
using System.Windows.Controls;

class MyGUIBuilder
{
  public void BuildUI()
  {
    // Создать элемент управления типа кнопки.
    Button btnOK = new Button();
    ...
  }
}
```

Благодаря импортированию в файле кода пространства имен `System.Windows.Controls` компилятор имеет возможность распознать класс `Button` как член этого пространства имен. Если не импортировать пространство имен `System.Windows.Controls`, то компилятор сообщит об ошибке. Тем не менее, можно также объявлять переменные с применением *полностью заданных имен*:

```
// Пространство имен System.Windows.Controls не указано!
using System;

class MyGUIBuilder
{
  public void BuildUI()
  {
```

```
    // Использование полностью заданного имени.
    System.Windows.Controls.Button btnOK =
      new System.Windows.Controls.Button();
    ...
  }
}
```

Хотя определение типа с использованием полностью заданного имени позволяет делать код более читабельным, трудно не согласиться с тем, что применение ключевого слова `using` в C# значительно сокращает объем набора на клавиатуре. В настоящей книге полностью заданные имена в основном использоваться не будут (разве что для устранения установленной неоднозначности), а предпочтение отдается упрощенному подходу с применением ключевого слова `using`.

Однако всегда помните о том, что ключевое слово `using` — это просто сокращенный способ указать полностью заданное имя типа. Любой из подходов дает в результате тот же самый код CIL (учитывая, что в коде CIL всегда используются полностью заданные имена) и не влияет ни на производительность, ни на размер сборки.

Ссылка на внешние сборки

В дополнение к указанию пространства имен через ключевое слово `using` языка C# компилятору C# также необходимо сообщить имя сборки, содержащей действительную реализацию на CIL типа, на который производится ссылка. Как упоминалось ранее, многие основные пространства имен .NET определены внутри сборки `mscorlib.dll`. Однако, к примеру, класс `System.Drawing.Bitmap` содержится в отдельной сборке по имени `System.Drawing.dll`. Подавляющее большинство сборок .NET Framework размещено в специальном каталоге, который называется *глобальным кешем сборок* (global assembly cache — GAC). На машине Windows по умолчанию GAC может находиться внутри каталога `C:\Windows\Assembly\GAC` (рис. 1.5).

В зависимости от инструмента разработки, применяемого для построения приложений .NET, вы будете иметь различные пути информирования компилятора о том, какие сборки необходимо включать в цикл компиляции. Все это подробно описано в главе 2, поэтому здесь детали опущены.

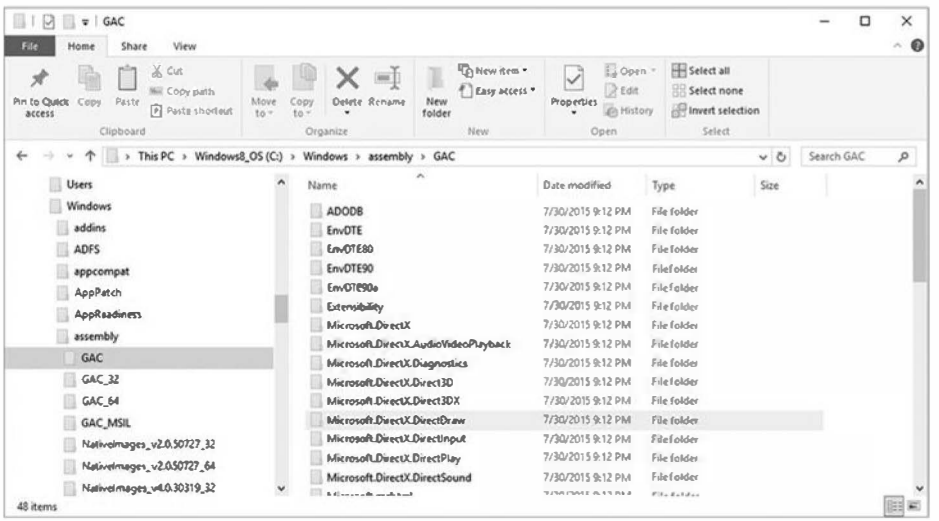


Рис. 1.5. Многие библиотеки .NET размещены в GAC

На заметку! Как будет показано в главе 14, в среде Windows есть несколько местоположений, где могут быть установлены библиотеки платформы; тем не менее, обычно это изолировано от разработчика. На машинах с операционными системами, отличными от Windows (такими как Mac OS X или Linux), местоположение GAC зависит от дистрибутива .NET.

Исследование сборки с помощью `ildasm.exe`

Если вас начинает беспокоить мысль о необходимости освоения всех пространств имен .NET, то просто вспомните о том, что уникальность пространству имен придает факт наличия в нем типов, которые каким-то образом *семантически связаны*. Следовательно, если в качестве пользовательского интерфейса достаточно простого консольного режима, то можно вообще не думать о пространствах имен для построения интерфейсов настольных и веб-приложений. Если вы создаете приложение рисования, то пространства имен для работы с базами данных, скорее всего, не понадобятся. Как и в случае любого нового набора готового кода, изучение должно проходить постепенно.

Утилита `ildasm.exe` (Intermediate Language Disassembler — дизассемблер промежуточного языка), которая поставляется в составе .NET Framework, позволяет загрузить любую сборку .NET и изучить ее содержимое, включая ассоциированный с ней манифест, код CIL и метаданные типов. Этот инструмент позволяет программистам более подробно разобраться, как их код C# отображается на код CIL, и в конечном итоге помогает понять внутреннюю механику функционирования платформы .NET. Хотя использование `ildasm.exe` вовсе не обязательно для того, чтобы стать опытным программистом .NET, настоятельно рекомендуется время от времени применять этот инструмент, чтобы лучше понимать, каким образом написанный код C# укладывается в концепции исполняющей среды.

На заметку! Запустить утилиту `ildasm.exe` довольно легко, открыв окно командной строки Visual Studio, введя в нем `ildasm` и нажав клавишу <Enter>.

После запуска утилиты `ildasm.exe` выберите пункт меню File⇨Open (Файл⇨Открыть) и перейдите к сборке, которую желаете исследовать. В целях иллюстрации на рис. 1.6 показано окно утилиты `ildasm.exe` со сборкой `Calc.exe`, сгенерированной на основе файла `Calc.cs`, содержимое которого приводилось ранее в главе. Утилита `ildasm.exe` представляет структуру любой сборки в знакомом древовидном формате.

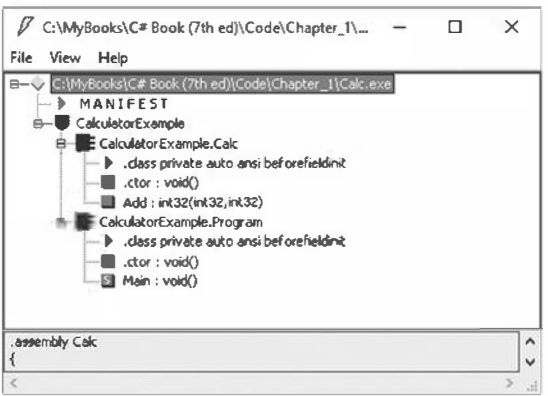


Рис. 1.6. Утилита ildasm.exe позволяет видеть содержащиеся внутри сборки .NET код CIL, манифест и метаданные типов

Просмотр кода CIL

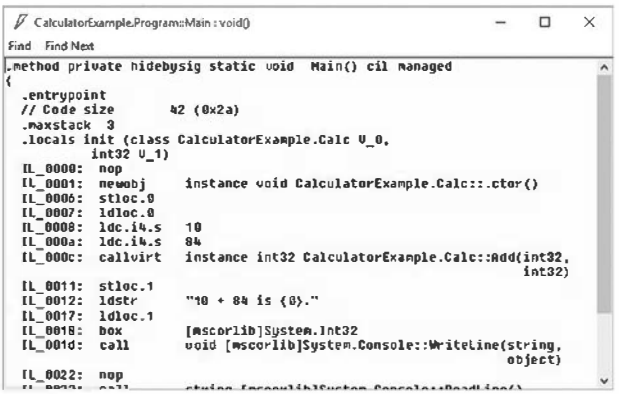


Рис. 1.7. Просмотр кода CIL для метода

В дополнение к отображению пространств имен, типов и членов, содержащихся в загруженной сборке, утилита `ildasm.exe` также позволяет просматривать инструкции CIL для заданного члена. Например, двойной щелчок на методе `Main()` класса `Program` приводит к открытию отдельного окна с кодом CIL для этого метода (рис. 1.7).

Просмотр метаданных типов

Для просмотра метаданных типов, содержащихся в загруженной в текущий момент сборке, необходимо нажать комбинацию клавиш <Ctrl+M>. На рис. 1.8 показаны метаданные для метода `Calc.Add()`.

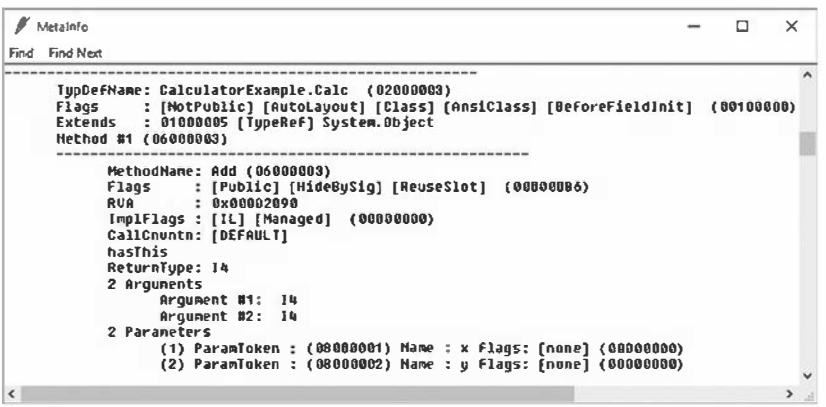


Рис. 1.8. Просмотр метаданных типов с помощью `ildasm.exe`

Просмотр метаданных сборки (манифеста)

Наконец, чтобы просмотреть содержимое манифеста сборки (рис. 1.9), нужно просто дважды щелкнуть на значке MANIFEST (Манифест) в главном окне утилиты `ildasm`. Разумеется, утилита `ildasm.exe` обладает более широким набором возможностей, чем было продемонстрировано здесь, и они еще будут рассматриваться в подходящих ситуациях.

Независимая от платформы природа .NET

Теперь хотелось бы сказать несколько слов о независимой от платформы природе .NET. К удивлению многих разработчиков, сборки .NET могут строиться и выполняться в средах операционных систем, отличных от Microsoft, в числе которых Mac OS X, различные дистрибутивы Linux, Solaris, а также iOS и Android на мобильных устройствах.

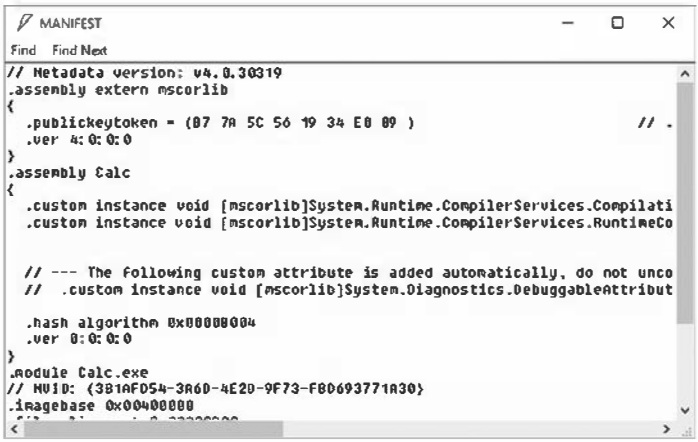


Рис. 1.9. Просмотр данных манифеста с помощью `ildasm.exe`

Чтобы понять, как такое возможно, необходимо рассмотреть еще одну аббревиатуру из мира .NET — CLI (Common Language Infrastructure — общеязыковая инфраструктура).

Когда компания Microsoft выпустила язык программирования C# и платформу .NET, был также разработан набор официальных документов, которые описывали синтаксис и семантику языков C# и CIL, формат сборок .NET, основные пространства имен .NET и механику действия исполняющей среды .NET. Эти документы были поданы в организацию Ecma International (`www.ecma-international.org`) и утверждены в качестве официальных международных стандартов. Наибольший интерес среди них представляют следующие спецификации:

- *ECMA-334: C# Language Specification* (спецификация языка C#);
- *ECMA-335: Common Language Infrastructure (CLI)* (спецификация общеязыковой инфраструктуры (CLI)).

Важность этих документов становится очевидной, как только вы осознаете тот факт, что они открывают третьим сторонам возможность создания дистрибутивов платформы .NET для любого числа операционных систем и/или процессоров. Из этих двух спецификаций ECMA-335 является более насыщенной, поэтому она разбита на разделы, которые описаны в табл. 1.4.

Имейте в виду, что в разделе IV ("Профили и библиотеки") описан лишь минимальный набор пространств имен, которые представляют основные службы, ожидаемые от дистрибутива CLI (например, коллекции, консольный ввод-вывод, файловый ввод-вывод, многопоточная обработка, рефлексия, доступ в сеть, основные средства защиты, манипулирование XML-данными). В CLI не определены пространства имен, которые упрощают разработку веб-приложений (ASP.NET), реализацию доступа к базам данных (ADO.NET) или создание настольных приложений с графическим пользовательским интерфейсом (Windows Presentation Foundation либо Windows Forms).

Однако хорошая новость состоит в том, что альтернативный дистрибутив .NET (названный Mono) расширяет библиотеки CLI совместимыми с Microsoft эквивалентами реализаций ASP.NET, реализаций ADO.NET и реализаций разнообразных инфраструктур для построения настольных приложений с графическим пользовательским интерфейсом, чтобы предложить полнофункциональные платформы разработки приложений производственного уровня. На сегодняшний день помимо специфичной для Windows платформы .NET от Microsoft существуют две крупных реализации, представленные в табл. 1.5.

Таблица 1.4. Разделы спецификации CLI

Раздел ECMA-335	Практический смысл
Раздел I. Концепции и архитектура	Описывает общую архитектуру CLI, в том числе правила CTS и CLS и механику функционирования исполняющей среды .NET
Раздел II. Определение и семантика метаданных	Описывает детали метаданных и формат сборок .NET
Раздел III. Набор инструкций CIL	Описывает синтаксис и семантику кода CIL
Раздел IV. Профили и библиотеки	Предоставляет высокоуровневый обзор минимальных и полных библиотек классов, которые должны поддерживаться дистрибутивом .NET
Раздел V. Формат обмена информацией отладки	Описывает стандартный способ обмена информацией отладки между создателями и потребителями CLI
Раздел VI. Приложения	Предоставляет набор разрозненных деталей, таких как руководящие указания по проектированию библиотек классов и детали реализации компилятора CIL

Таблица 1.5. Дистрибутивы .NET с открытым кодом

Дистрибутив	Описание
Проект Mono	Проект Mono — это дистрибутив CLI с открытым кодом, который ориентирован на разнообразные версии Linux (например, SuSe, Fedora), Mac OS X, устройства iOS (iPad, iPhone), устройства Android и (сюрприз) Windows
.NET Core 5	В дополнение к ориентированной на Windows реализации .NET Framework в Microsoft также поддерживается межплатформенная версия .NET, которая сосредоточена на конструировании библиотек кода и корпоративных веб-приложений

Проект Mono

Проект Mono окажется великолепным выбором, если вы планируете строить программное обеспечение .NET, которое способно выполняться под управлением разнообразных операционных систем. Вдобавок ко всем основным пространствам имен .NET проект Mono предоставляет дополнительные библиотеки, делая возможным создание настольного программного обеспечения с графическим пользовательским интерфейсом, веб-приложений ASP.NET и программного обеспечения для мобильных устройств (iPad, iPhone и Android). Загрузить дистрибутив Mono можно по следующему URL:

```
www.mono-project.com/
```

Изначально проект Mono состоит из ряда инструментов командной строки, с которыми ассоциированы библиотеки кода. Тем не менее, как вы увидите в главе 2, вместе с Mono обычно используется готовая графическая IDE-среда под названием Xamarin Studio. На самом деле проекты Microsoft Visual Studio могут быть загружены в проекты Xamarin и наоборот. Более подробную информацию вы можете почерпнуть из главы 2, но имеет смысл также посетить указанный ниже веб-сайт Xamarin:

```
http://xamarin.com/
```

На заметку! В приложении Б, доступном в загружаемых примерах кода, предлагается обзор платформы Mono.

Microsoft .NET Core

Другой крупный межплатформенный дистрибутив .NET поступает от самой компании Microsoft Corporation. В начале 2014 года компанией Microsoft была анонсирована версия с открытым кодом полномасштабной (специфичной для Windows) платформы .NET 4.6 Framework, получившая название .NET Core. Дистрибутив .NET Core не является полным дубликатом .NET 4.6 Framework. Вместо этого .NET Core фокусируется на строительстве веб-приложений ASP.NET, которые могут запускаться в средах Linux, Mac OS X и Windows. Таким образом, по существу вы можете считать .NET Core подмножеством полновесной платформы .NET Framework. На веб-сайте .NET Blog в MSDN есть хорошая статья, в которой приводится сравнение и противопоставление полной платформы .NET Framework и .NET Core. Ниже показана прямая ссылка (если она изменится, то просто выполните поиск в Интернете по ключевой фразе .NET Core is Open Source):

```
http://blogs.msdn.com/b/dotnet/archive/2014/11/12/
  net-core-is-open-source.aspx
```

К счастью, в состав .NET Core включены все средства C# и несколько основных библиотек. По этой причине большая часть настоящей книги будет напрямую применима и к данному дистрибутиву. Однако вспомните, что дистрибутив .NET Core ориентирован на построение веб-приложений и не предоставляет реализаций API-интерфейсов создания графических пользовательских интерфейсов для настольных приложений (таких как WPF или Windows Forms). Если вам необходимо разрабатывать межплатформенные настольные приложения с графическим пользовательским интерфейсом, то предпочтение следует отдать проекту Mono. Полезно также отметить, что компания Microsoft выпустила бесплатный, легковесный и межплатформенный редактор кода для помощи в разработке с использованием .NET Core. Этот редактор назван просто — Visual Studio Code. Несмотря на то что он определенно не обладает полным набором средств, как у продукта Microsoft Visual Studio или Xamarin Studio, редактор Visual Studio Code представляет собой удобный инструмент для редактирования кода C# в межплатформенной манере. Этот редактор в книге не рассматривается, но при желании дополнительные сведения о нем можно получить на следующем веб-сайте:

```
https://code.visualstudio.com/
```

Резюме

Задачей настоящей главы было формирование концептуальной основы, требуемой для освоения остального материала книги. Сначала исследовались ограничения и сложности, присущие технологиям, которые предшествовали платформе .NET, после чего было в общих чертах показано, как .NET и C# пытаются упростить существующее положение дел. По существу платформа .NET сводится к механизму исполняющей среды (`mscoree.dll`) и библиотекам базовых классов (`mscorlib.dll` и связанные с ней библиотеки). Общеязыковая исполняющая среда (CLR) способна обслуживать любые двоичные модули .NET (называемые сборками), которые следуют правилам управляемого кода. Как вы увидели, сборки содержат инструкции CIL (в дополнение к метаданным типов и манифестам сборок), которые с помощью JIT-компилятора транслируются в инструкции, специфичные для платформы. Кроме того, вы ознакомились с ролью общеязыковой спецификации (CLS) и общей системы типов (CTS). После этого вы узнали об инструменте для просмотра объектов по имени `ildasm.exe`.

В следующей главе будет предложен обзор распространенных IDE-сред, которые можно применять при построении программных проектов на языке C#. Вас наверняка обрадует тот факт, что в книге будут использоваться полностью бесплатные (и богатые возможностями) IDE-среды, поэтому вы начнете изучение мира .NET без каких-либо финансовых затрат.

ГЛАВА 2

Создание приложений на языке C#

Как программист на C#, вы можете выбрать подходящий инструмент среди многочисленных средств для построения приложений .NET. Выбор инструмента (или инструментов) будет осуществляться главным образом на основе трех факторов: сопутствующие финансовые затраты, операционная система (ОС), которую вы используете для разработки программного обеспечения, и вычислительные платформы, на которые оно должно ориентироваться. Цель этой главы — предоставить обзор наиболее распространенных интегрированных сред разработки (integrated development environment — IDE), которые поддерживают язык C#. Вы должны иметь в виду, что в настоящей главе *не* будут описаны мельчайшие подробности каждой IDE-среды; она только предоставит достаточный объем информации, чтобы вы сумели успешно установить среду для программирования, и даст вам основу, от которой можно двигаться дальше.

В первой части этой главы будет исследоваться набор IDE-сред от Microsoft, которые дают возможность разрабатывать приложения .NET в среде ОС Windows (7, 8.*x* и 10). ак вы увидите, некоторые из IDE-сред могут применяться для построения только приложений, ориентированных на Windows, в то время как другие поддерживают создание приложений C# для альтернативных ОС и устройств (подобных Mac OS X, Linux или Android). Во второй части главы будут рассматриваться IDE-среды, которые могут *выполняться* под управлением ОС, отличной от Windows. Они дают разработчикам возможность строить программы на C#, используя компьютеры Apple, а также дистрибутивы Linux.

На заметку! В данной главе будет предложен обзор довольно большого числа IDE-сред. Тем не менее, во всей книге предполагается, что вы применяете (полностью бесплатную) IDE-среду Visual Studio Community. Если вы хотите строить свои приложения в среде другой ОС (Mac OS X или Linux), то эта глава укажет правильное направление, но окна выбранной вами IDE-среды будут отличаться от тех, которые изображены на экранных снимках, приводимых в тексте.

Построение приложений C# в среде Windows

Изучая материал этой главы, вы увидите, что для построения приложений C# выбирать подходящий вариант можно из множества IDE-сред; некоторые IDE-среды предлагаются Microsoft, а другие — независимыми поставщиками (причем многие среды поставляются с открытым исходным кодом). В настоящее время, вопреки тому, что вы могли подумать, многие IDE-среды производства Microsoft являются совершенно бесплатными.

Таким образом, если ваш основной интерес заключается в построении программного обеспечения .NET для ОС Windows (7, 8.*x* или 10), то вы обнаружите следующие главные варианты:

- Visual Studio Express
- Visual Studio Community
- Visual Studio Professional (или выше)

Хотя перечисленные IDE-среды предоставляют похожую функциональность, они отличаются в основном количеством средств корпоративного уровня и числом поддерживаемых типов проектов. Например, в редакциях Express среды Visual Studio отсутствует ряд усовершенствованных инструментов интеграции с базами данных и специализированные шаблоны проектов для альтернативных языков .NET (таких как F# и Python), которые включены в редакцию Visual Studio 2015 Professional. Редакция Visual Studio Community поддерживает те же самые типы проектов, что и Visual Studio 2015 Professional, но по-прежнему лишена средств, которые были бы полезными в корпоративной среде разработки (вроде полной интеграции с продуктом Team Foundation Server). К счастью, каждая IDE-среда поставляется с развитыми редакторами кода, основными визуальными конструкторами баз данных, встроенными визуальными отладчиками, визуальными конструкторами графических пользовательских интерфейсов для настольных и веб-приложений и т.д. Давайте начнем с исследования роли IDE-сред семейства Express.

Семейство IDE-сред Visual Studio Express

Семейство IDE-сред Visual Studio Express является совершенно бесплатным. В ранних версиях платформы .NET редакции Express разделялись по поддерживаемым ими языкам .NET; например, в свое время компания Microsoft предлагала инструменты под названиями C# Express, VB Express, Web Developer Express и C++ Express. Однако в последнее время семейство Express было перераспределено на основе исключительно типов приложений, которые планируется создавать (веб-приложения, настольные приложения и т.д.). В частности, в состав семейства Visual Studio Express входят перечисленные ниже члены.

- *Express for Windows Desktop* (IDE-среда Express для рабочего стола Windows). Поддерживает разработку консольных приложений и настольных приложений с графическим пользовательским интерфейсом (Windows Forms и Windows Presentation Foundation) для Windows. Воспринимает языки C#, VB и C++.
- *Express for Windows 10* (IDE-среда Express для Windows 10). Поддерживает разработку основного программного обеспечения, ориентированного на "универсальный" тип приложений Windows 10, которые могут запускаться на многочисленных устройствах Microsoft (Windows, Xbox, Windows Mobile, HoloLens и т.п.). Воспринимает языки C#, VB и C++.
- *Express for Web (IDE-среда Express для веб).* Поддерживает разработку веб-приложений ASP.NET, облачных приложений Azure и приложений Microsoft Silverlight. Воспринимает языки C#, VB и C++.
- *Team Foundation Server 2015 Express.* Эта версия семейства Express сосредоточена на предоставлении графического пользовательского интерфейса для поддержки множества версий кода, создания и обработки утверждений и задач, а также обеспечения возможности сотрудничества для команд разработчиков программного обеспечения. С данной редакцией поставляются ограниченные инструменты разработки, поэтому подробно она рассматриваться не будет.

На заметку! Продукты Express можно загрузить по ссылке `https://www.visualstudio.com/products/visual-studio-express-vs` (или просто выполните поиск в Интернете по ключевой фразе *Visual Studio Express*).

Инструменты Express удобны для тех, кто только начинает свой путь как разработчика приложений .NET. Причина в том, что они предоставляют все важные средства, которые обычно ожидается найти (визуальные конструкторы графического пользовательского интерфейса, отладчики, редакторы кода с богатой функциональностью и т.п.), но не перегружают десятками вспомогательных или сложных средств, вызывающих нежелательную путаницу. Кроме того, инструменты Express могут идеально подойти тем, для кого программирование является хобби, или же тем, кто предпочитает использовать "минимальную, но полную" IDE-среду.

При желании можете загрузить IDE-среды Express for Windows Desktop и Express for Web и успешно работать с ними по мере чтения книги. Грубо говоря, материалы глав 2–30 можно усвоить, применяя версию Express for Windows Desktop, т.к. они сконцентрированы на консольных приложениях, приложениях WPF и временами приложениях Windows Forms. В остальных главах книги, в которых раскрывается разработка веб-приложений (глава 31 и далее), используется версия Express for Web.

Имейте в виду, что IDE-среды Express, Visual Studio 2015 Community и Visual Studio 2015 Professional разделяют общий набор основных функциональных возможностей. Таким образом, между ними легко перемещаться и чувствовать себя вполне комфортно, выполняя базовые работы. С учетом этого давайте чуть более подробно рассмотрим некоторые IDE-среды Express (как вы помните, все они совершенно бесплатны).

Краткий обзор IDE-среды Express for Windows Desktop

Эта версия семейства Express позволяет строить настольные приложения, которые выполняются напрямую под управлением ОС Windows (7, 8.x или 10). Чтобы получить о ней представление, мы посвятим некоторое время построению простого приложения C# с применением Express for Windows Desktop, не забывая о том, что проиллюстрированные здесь аспекты относятся ко всем IDE-средам Microsoft.

Диалоговое окно New Project и редактор кода C#

Предполагая, что IDE-среда Express for Windows Desktop была успешно загружена и установлена, выберите пункт меню File⇨New Project (Файл⇨Создать проект). Как видно в открывшемся диалоговом окне New Project (Новый проект), показанном на рис. 2.1, эта IDE-среда предлагает поддержку для консольных приложений, приложений WPF/Windows Forms и ряда низкоуровневых типов проектов C++. Для начала создадим новый проект Console Application (Консольное приложение) на C# по имени `SimpleCSharpConsoleApp`.

После создания проекта вы увидите содержимое начального файла кода C# (по имени `Program.cs`), который открывается в редакторе кода. Добавьте в метод `Main()` следующий код C#. Набирая код, вы заметите, что во время применения операции точки активизируется средство IntelliSense.

```
static void Main(string[] args)
{
  // Настройка консольного пользовательского интерфейса.
  Console.Title = "My Rocking App";
  Console.ForegroundColor = ConsoleColor.Yellow;
  Console.BackgroundColor = ConsoleColor.Blue;
  Console.WriteLine("************************************");
  Console.WriteLine("***** Welcome to My Rocking App *****");
```

```
  Console.WriteLine("*********************************");
  Console.BackgroundColor = ConsoleColor.Black;
  // Ожидание нажатия клавиши <Enter>.
  Console.ReadLine();
}
```

Здесь используется класс `Console`, определенный в пространстве имен `System`. Поскольку пространство имен `System` было автоматически включено посредством оператора `using` в начале файла, указывать это пространство имен перед именем класса не обязательно (например, `System.Console.WriteLine()`). Данная программа не делает ничего особо интересного; тем не менее, обратите внимание на последний вызов `Console.ReadLine()`. Он просто обеспечивает поведение, при котором пользователь должен нажать клавишу <Enter>, чтобы завершить приложение. Если этого не сделать, то программа исчезнет почти мгновенно при проведении ее отладки!

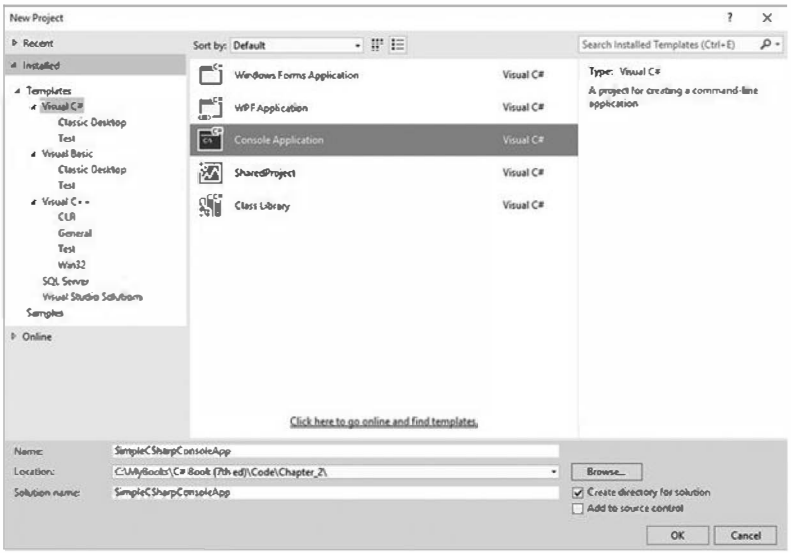


Рис. 2.1. Диалоговое окно New Project

Запуск и отладка проекта

Теперь для запуска программы и просмотра вывода можете просто нажать комбинацию клавиш <Ctrl+F5> (или выбрать соответствующий пункт в меню Debug (Отладка)). После этого на экране появится окно консоли Windows с вашим специальным (раскрашенным) сообщением. Вы должны понимать, что когда "запускаете" свою программу, то обходите интегрированный отладчик.

Если написанный код нужно отладить (что определенно станет важным при построении более крупных программ), то первым делом понадобится поместить точки останова на операторы кода, которые необходимо исследовать. Хотя в рассматриваемом примере не особенно много кода, установите точку останова, щелкнув на крайней слева полосе серого цвета в окне редактора кода (обратите внимание, что точки останова помечаются значком в виде красного кружка (рис. 2.2)).

Если теперь нажать клавишу <F5> (или воспользоваться меню Debug), то программа будет прекращать работу на каждой точке останова. Как и можно было ожидать, у вас есть возможность взаимодействовать с отладчиком с помощью разнообразных кнопок панели инструментов и пунктов меню IDE-среды. После оценки вами всех точек останова приложение в конечном итоге завершится, когда закончится метод `Main()`.

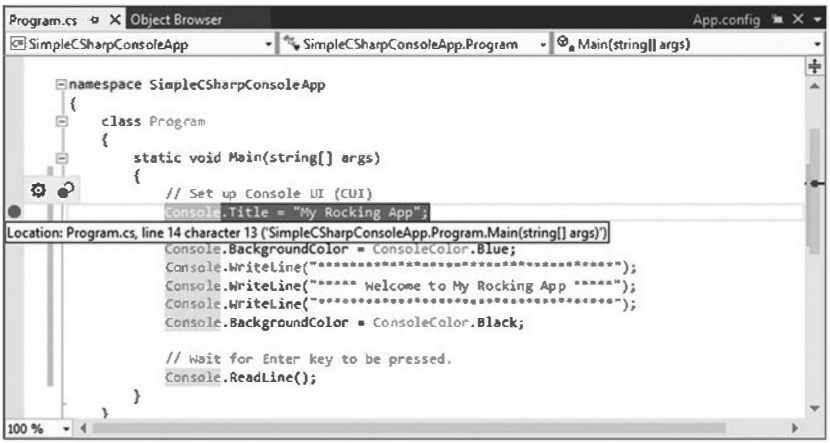


Рис. 2.2. Установка точки останова

На заметку! Предлагаемые Microsoft среды IDE обладают современными отладчиками, и в последующих главах вы изучите разные приёмы работы с ними. Пока что нужно знать, что при нахождении в сеансе отладки в меню Debug появляется большое количество полезных пунктов. Выделите время на ознакомление с ними.

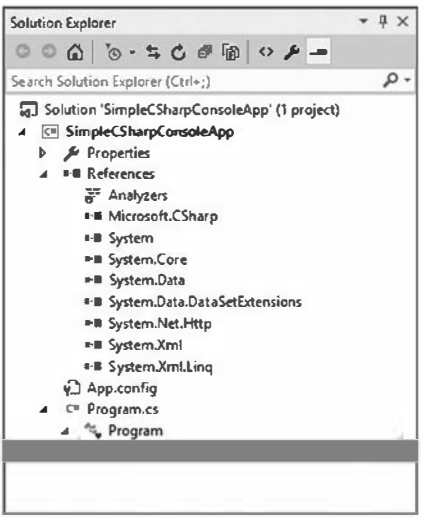


Рис. 2.3. Окно Solution Explorer

Окно Solution Explorer

Взглянув на правую часть IDE-среды, вы заметите окно под названием Solution Explorer (Проводник решений), в котором отображено несколько важных элементов. Первым делом обратите внимание, что IDE-среда создала решение с единственным проектом (рис. 2.3). Поначалу это может сбивать с толку, т.к. решение и проект имеют одно и то же имя (SimpleCSharpConsoleApp). Идея здесь заключается в том, что "решение" может содержать множество проектов, работающих совместно. Например, в состав решения могут входить три библиотеки классов, одно приложение WPF и одна веб-служба WCF. В начальных главах этой книги будет всегда применяться одиночный проект; однако, когда мы займемся построением более сложных примеров, будет показано, каким образом добавлять новые проекты в первоначальное пространство решения.

На заметку! Учтите, что в случае выбора решения в самом верху окна Solution Explorer система меню IDE-среды будет отображать набор пунктов, который отличается от ситуации, когда выбран проект. Если вы когда-нибудь обнаружите, что определенный пункт меню исчез, то внимательно проверьте, не выбрали ли случайно ошибочный узел.

Вы также заметите узел References (Ссылки). Его можно использовать, когда приложение нуждается в ссылке на дополнительные библиотеки .NET помимо тех, которые по умолчанию включаются для данного типа проекта.

Поскольку был создан проект Console Application на языке C#, вы увидите несколько автоматически добавленных библиотек, таких как `System.dll`, `System.Core.dll`, `System.Data.dll` и т.д. (обратите внимание, что элементы, перечисленные в узле References, не отображают файловое расширение `.dll`). Вскоре будет показано, как добавлять библиотеки к проекту.

На заметку! Вспомните из главы 1, что все проекты .NET имеют доступ к фундаментальной библиотеке по имени `mscorlib.dll`. Эта библиотека настолько необходима, что она даже не перечислена явно в окне Solution Explorer.

Окно Object Browser

В результате двойного щелчка на любой библиотеке в узле References откроется интегрированное окно Object Browser (Браузер объектов); его также можно открыть с помощью меню View (Вид). С применением данного инструмента можно просматривать разнообразные пространства имен в сборке, типы в пространстве имен и члены каждого типа. На рис. 2.4 показано окно Object Browser, которое отображает ряд пространств имен в постоянно присутствующей сборке `mscorlib.dll`.

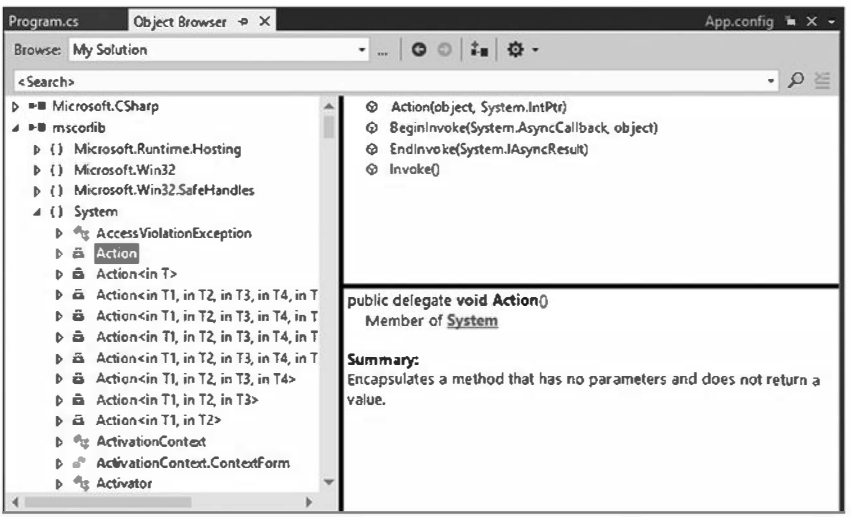


Рис. 2.4. Окно Object Browser

Этот инструмент может быть удобен, когда необходимо увидеть внутреннюю организацию какой-то библиотеки .NET, а также получить краткое описание заданного элемента. Кроме того, обратите внимание на поле <Search> (<Поиск>) в верхней части окна. Оно может быть полезно, если вы знаете имя типа, который нужно использовать, но не имеете понятия, где он находится. В качестве связанного замечания имейте в виду, что по умолчанию средство поиска будет искать только в библиотеках, задействованных в решении (чтобы производить поиск в рамках всей платформы .NET Framework, понадобится изменить выбранный элемент в раскрывающемся списке Browse (Просмотр)).

Ссылка на дополнительные сборки

Продолжая исследование, давайте добавим сборку (также известную как библиотека кода), которая не включается в проект Console Application автоматически. Для этого щелкните правой кнопкой мыши на узле References в окне Solution Explorer и выберите в контекстном меню пункт Add Reference (Добавить ссылку); или же выбе-

рите пункт меню Project⇨Add Reference (Проект⇨Добавить ссылку). В открывшемся диалоговом окне Add Reference (Добавление ссылки) отыщите библиотеку по имени `System.Windows.Forms.dll` (опять-таки, файловые расширения здесь не показаны) и отметьте флажок возле нее (рис. 2.5).

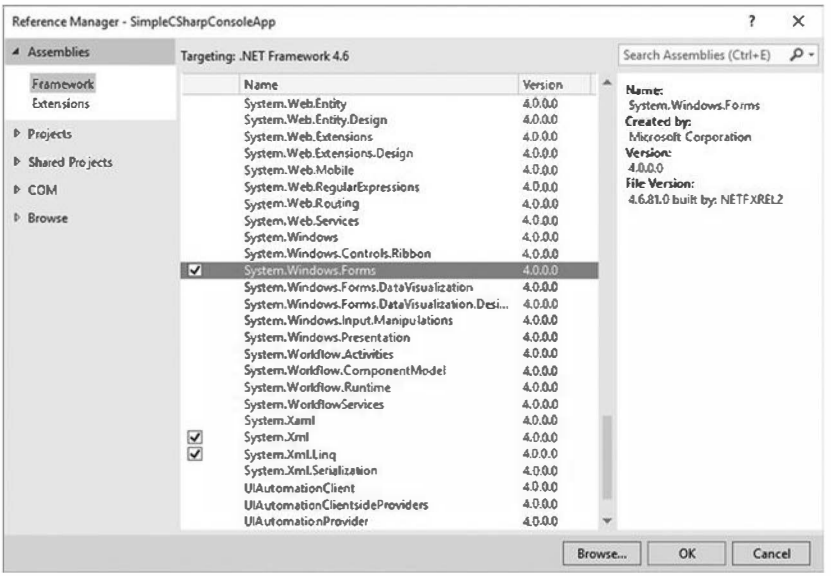


Рис. 2.5. Диалоговое окно Add Reference

После щелчка на кнопке OK новая библиотека добавляется к нашему набору ссылок (вы увидите ее в списке под узлом References). Тем не менее, как объяснялось в главе 1, добавление ссылки на библиотеку — это лишь первый шаг. Чтобы применять типы в файле кода C#, понадобится указать оператор `using`. Добавьте следующую строку к директивам `using` в своем файле коде:

```
using System.Windows.Forms;
```

Затем поместите приведенную ниже строку кода сразу после вызова `Console.ReadLine()` в методе `Main()`:

```
MessageBox.Show("All done!");
```

Запустив или начав отладку программы еще раз, вы обнаружите, что перед завершением работы программы появляется простое диалоговое окно сообщения.

Просмотр свойств проекта

Далее обратите внимание на узел Properties (Свойства) в окне Solution Explorer. Двойной щелчок на нем приводит к открытию сложно устроенного редактора конфигурации проекта. Например, на рис. 2.6 видно, что можно изменять версию платформы .NET Framework, на которую ориентируется решение.

Различные аспекты окна свойств проекта будут обсуждаться далее в книге. Здесь можно устанавливать разнообразные настройки безопасности, назначать сборке строгое имя (глава 14), развертывать приложение, вставлять ресурсы приложения и конфигурировать события, происходящие до и после построения.

На этом краткий обзор IDE-среды Express for Windows Desktop завершен. На самом деле в данном инструменте доступно намного больше средств, чем было показано к настоящему моменту.

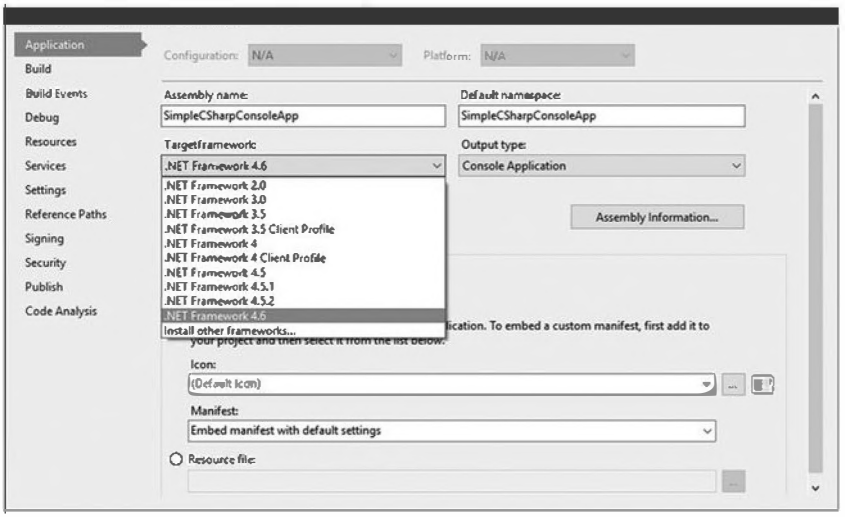


Рис. 2.6. Окно свойств проекта

Вспомните, что редакция Visual Studio Community имеет такой же графический пользовательский интерфейс, как и Express for Windows Desktop. В последующих главах книги будут представлены другие возможности, применимые к любой из двух IDE-сред, но лучше выделить время и уже сейчас поупражняться с пунктами меню, диалоговыми окнами и настройками свойств.

Исходный код. Проект `SimpleCSharpConsoleApp` находится в подкаталоге `Chapter_2`.

Краткий обзор IDE-среды Express for Web

Если вы хотите строить веб-приложения на платформе .NET с помощью инструментального набора Express, то можете загрузить и установить бесплатную IDE-среду Express for Web. В финальных главах этой книги (главы 31–34) будет рассмотрено несколько важных деталей относительно конструирования веб-приложений под управлением платформы .NET, а пока просто взгляните на рис. 2.7, где показаны типы проектов, которые можно создавать в данной IDE-среде через пункт меню File⇨New Project (Файл⇨Создать проект).

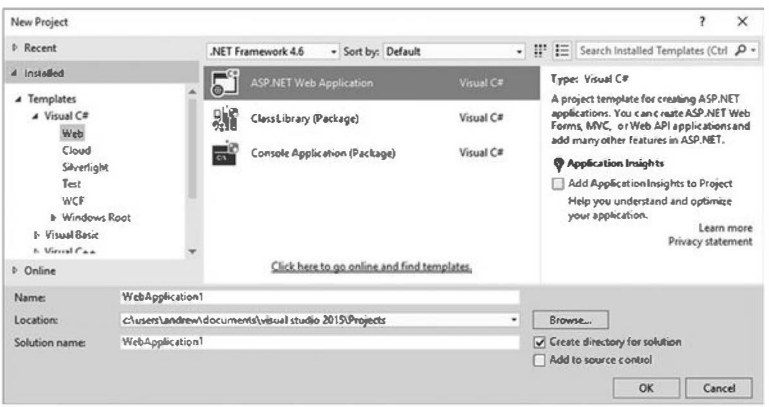


Рис. 2.7. Диалоговое окно New Project в IDE-среде Express for Web

Как видите, IDE-среда Express for Web позволяет создавать веб-приложения ASP.NET и графические пользовательские веб-интерфейсы Silverlight, а также располагает поддержкой облачных служб Microsoft Azure. Вдобавок эта IDE-среда предоставляет шаблон WCF, который дает возможность строить распределенные решения, ориентированные на службы. Мы исследуем ASP.NET в финальных главах книги.

Итак, краткий обзор ряда членов семейства IDE-сред Express завершен. Как уже упоминалось, вы обнаружите, что эти инструменты предлагают "вполне достаточную" функциональность для кодирования. Далее мы переходим к изучению роли Visual Studio Community.

IDE-среда Visual Studio Community

Каждый инструмент Express ограничен в том, что позволяет строить программное обеспечение .NET, которое будет выполняться только под управлением ОС Windows (7, 8.x или 10). Однако, как утверждалось в главе 1, платформа .NET запускается в средах разнообразных ОС и аппаратных устройств. Таким образом, если вам необходимо создать программу .NET, которая способна выполняться под управлением (к примеру) Android или продукта Apple, то инструменты семейства Express не особенно помогут. К счастью, Microsoft предлагает еще одну полностью бесплатную IDE-среду, которая предоставляет возможность строить намного более широкий спектр типов проектов, используя большое число языков .NET — IDE-среду Visual Studio Community.

На заметку! Продукт Visual Studio Community доступен для загрузки по адресу `https://www.visualstudio.com/products/visual-studio-community-vs`.

Первое, о чем следует знать — продукт Visual Studio Community снабжает единой средой для построения настольных *и* веб-приложений (а также "универсальных" приложений .NET). Соответственно, в отличие от семейства Express, нет нужды в загрузке множества продуктов.

Кроме того, эта IDE-среда обеспечивает поддержку ряда дополнительных языков программирования (F#, Python и JavaScript) и типов проектов. Вы найдете не только более специализированные типы проектов для ОС Windows OS, но также типы проектов, которые ориентированы на платформы, отличные от Microsoft. Ниже приведены самые примечательные примеры:

- проекты, ориентированные на приложения Windows Phone и Windows 8.x;
- проекты, ориентированные на устройства Android;
- проекты, ориентированные на устройства с семейством ОС iOS (iPad, iPhone и Apple Watch);
- проекты, ориентированные на низкоуровневые API-интерфейсы C++, такие как MFC и ATL;
- несколько проектов, которые ориентированы на построение видеоигр для разнообразных устройств;
- проекты, позволяющие расширять Visual Studio Community (а также Visual Studio Professional) новой функциональностью посредством расширяемых подключаемых модулей;
- проекты для построения специальных сценариев PowerShell.

Чтобы получить лучшее представление обо всех языках и типах проектов, поддерживаемых в Visual Studio Community, взгляните на рис. 2.8, где приведено диалоговое окно New Project этой IDE-среды.

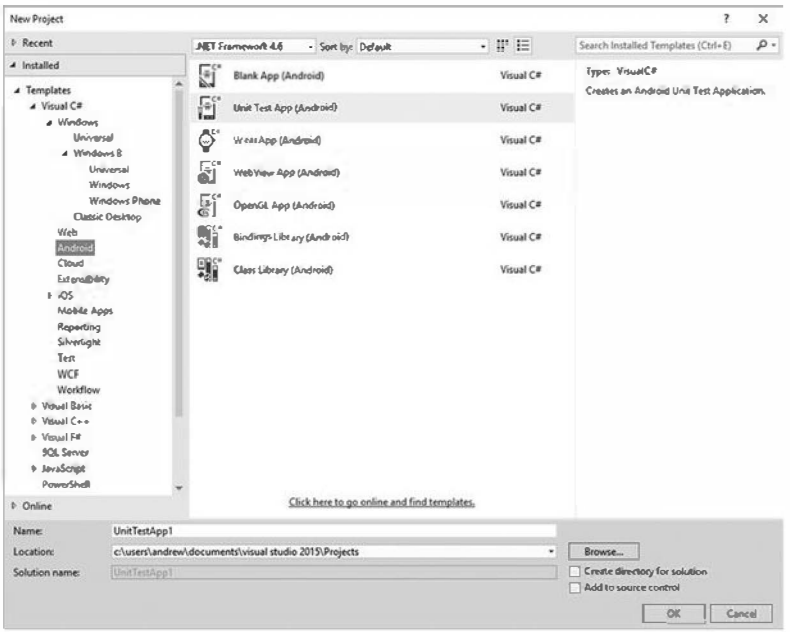


Рис. 2.8. Диалоговое окно New Project в IDE-среде

Визуальный конструктор типов

Среда Visual Studio Community также дает возможность конструировать классы и другие типы (вроде интерфейсов или делегатов) в визуальной манере (в IDE-средах семейства Express она отсутствует). Утилита Class Designer (Конструктор классов) позволяет просматривать и модифицировать отношения между типами (классами, интерфейсами, структурами, перечислениями и делегатами) в проекте. С помощью данного инструмента можно визуально добавлять (или удалять) члены типа с отражением этих изменений в соответствующем файле кода C#. Кроме того, изменения, вносимые в такой файл кода C#, отражаются в диаграмме классов.

Предполагая, что продукт Visual Studio Community был успешно установлен, создайте новый проект типа Console Application на C# по имени `VisualTypeDesignerApp`. Для доступа к средствам визуального конструктора типов сначала понадобится вставить новый файл диаграммы классов. Чтобы сделать это, выберите пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент) и в открывшемся окне найдите элемент Class Diagram (Диаграмма классов), как показано на рис. 2.9.

Изначально поверхность визуального конструктора будет пустой; тем не менее, вы можете перетаскивать на нее файлы из окна Solution Explorer. Например, после перетаскивания на поверхность конструктора файла `Program.cs` вы увидите визуальное представление класса `Program`. Щелкая на значке с изображением стрелки для заданного типа, можно отображать или скрывать его члены (рис. 2.10).

На заметку! Используя панель инструментов утилиты Class Designer, можно настраивать параметры отображения поверхности визуального конструктора.

Утилита Class Designer работает в сочетании с двумя другими средствами Visual Studio — окном Class Details (Детали класса), которое открывается через меню View⇨Other Windows (Вид⇨Другие окна), и панелью инструментов Class Designer, отображаемой выбором пункта меню View⇨Toolbox (Вид⇨Панель инструментов).

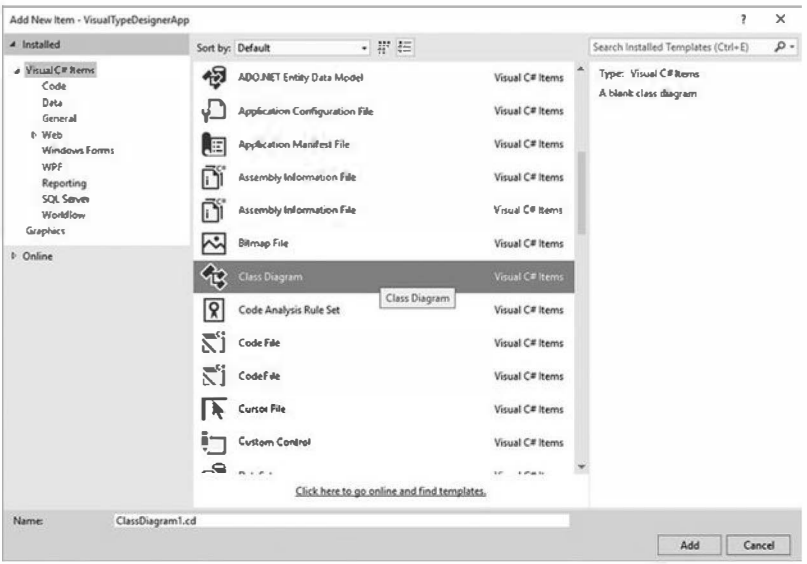


Рис. 2.9. Вставка файла диаграммы классов в текущий проект

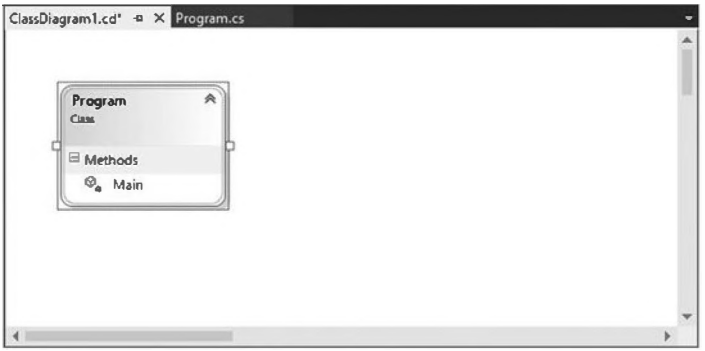


Рис. 2.10. Просмотр диаграммы классов

Окно Class Details не только показывает подробные сведения о текущем выбранном элементе диаграммы, но также позволяет модифицировать существующие члены и вставлять новые члены на лету (рис. 2.11).

Панель инструментов Class Designer, которая также может быть активизирована с применением меню View, позволяет вставлять в проект новые типы (и создавать между ними отношения) визуальным образом (рис. 2.12). (Чтобы видеть эту панель инструментов, должно быть активным окно диаграммы классов.) По мере выполнения таких действий IDE-среда создает на заднем плане новые определения типов на C#.

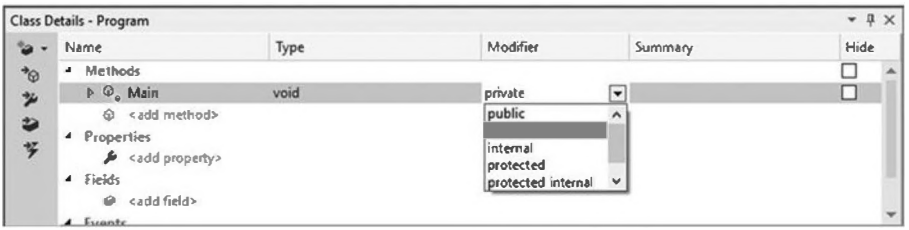


Рис. 2.11. Окно Class Details

В качестве примера перетащите новый элемент Class (Класс) из панели инструментов Class Designer в окно Class Designer. В открывшемся диалоговом окне назначьте ему имя Car. В результате будет создан новый файл C# по имени Car.cs и затем автоматически добавлен к проекту. Теперь, используя окно Class Details, добавьте открытое поле типа string с именем PetName (рис. 2.13).

После этого вы увидите, что определение класса Car на C# было соответствующим образом обновлено (за исключением приведенного ниже комментария):

```
public class Car
{
  // Использовать открытые данные обычно
  // не рекомендуется, но здесь это упрощает
пример.
  public string petName;
}
```

Снова активизируйте утилиту Class Designer, перетащите на поверхность визуального конструктора еще один новый элемент Class и назначьте ему имя SportsCar. Далее выберите значок Inheritance (Наследование) в панели инструментов Class Designer и щелкните в верхней части значка SportsCar. Затем щелкните в верхней части значка класса Car. Если все было сделано правильно, то класс SportsCar станет производным от класса Car (рис. 2.14).

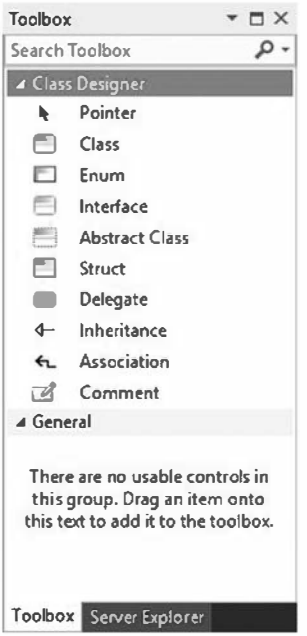


Рис. 2.12. Панель инструментов Class Designer

На заметку! Концепция наследования будет полностью раскрыта в главе 6.

Чтобы завершить данный пример, обновите сгенерированный класс SportsCar, добавив открытый метод по имени GetPetName() со следующим кодом:

```
public class SportsCar : Car
{
  public string GetPetName()
  {
    petName = "Fred";
    return petName;
  }
}
```

Как и можно было ожидать, визуальный конструктор типов является одним из многочисленных средств Visual Studio Community. Ранее уже упоминалось, что в этом издании книги предполагается применение Visual Studio Community в качестве избранной IDE-среды. В последующих главах вы изучите намного больше возможностей этого инструмента.

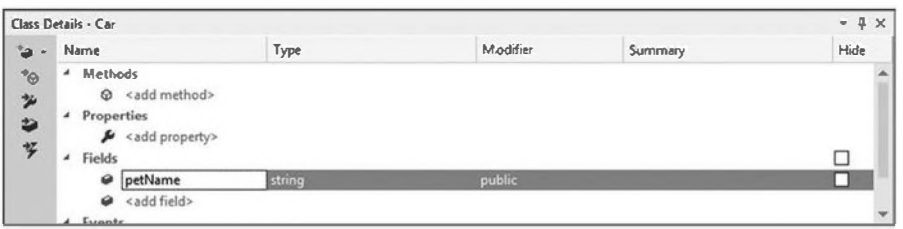


Рис. 2.13. Добавление поля с помощью окна Class Details

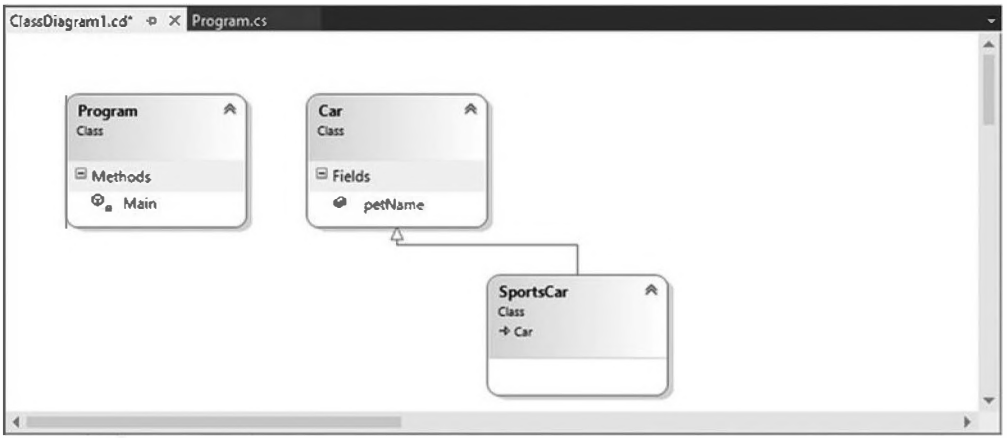


Рис. 2.14. Визуальное наследование от существующего класса

Исходный код. Проект `VisualTypeDesignerApp` находится в подкаталоге `Chapter_2`.

IDE-среда Visual Studio 2015 Professional

В завершение наших исследований IDE-сред, которые функционируют исключительно в среде ОС Windows, давайте кратко рассмотрим Visual Studio 2015 Professional. Если в настоящее время вы занимаете должность инженера по разработке программного обеспечения, то велики шансы того, что компания приобрела копию этой IDE-среды, которая и является вашим избранным инструментом. Продукт Visual Studio 2015 Professional обладает всеми теми же средствами, что и Visual Studio Community (аналогичными типами проектов, набором языков и доступными визуальными конструкторами). Вдобавок эта IDE-среда имеет несколько средств, направленных на организацию совместной корпоративной разработки. К примеру, благодаря Visual Studio Professional вы получаете следующие возможности:

- интеграция с Team Foundation Server (TFS) для управления досками Agile и Kanban;
- инструменты для создания и управления историями, задачами и эпопеями;
- интеграция с SharePoint и дискуссионными группами разработчиков;
- инструменты для управления спринтами.

В действительности погружение в детали жизненного цикла разработки программного обеспечения выходит за рамки настоящей книги. В связи с этим продукт Visual Studio 2015 Professional больше обсуждаться не будет. Если вы выбрали данную IDE-среду, то все прекрасно. Вспомните, что за пределами таких ориентированных на командную разработку инструментов функциональность Visual Studio Community и Visual Studio Professional идентична.

На заметку! По адресу `https://www.visualstudio.com/products/compare-visual-studio-2015-products-vs` можно ознакомиться с результатами сравнения IDE-сред Visual Studio Community и Visual Studio Professional.

Система документации .NET Framework

Последним аспектом Visual Studio, с которым вы *должны* уметь работать с самого начала, является полностью интегрированная справочная система. Документация .NET Framework представляет собой исключительно хороший, понятный и насыщенный полезной информацией источник. Из-за огромного количества предопределенных типов .NET (их тысячи) необходимо уделить время исследованию предлагаемой документации, иначе вряд ли вас ожидает особый успех на поприще разработки приложений .NET.

При наличии подключения к Интернету просматривать документацию .NET Framework можно в онлайновом режиме по следующему адресу:

```
http://msdn.microsoft.com/library
```

Оказавшись на главной странице веб-сайта, найдите раздел Development Tools and Languages (Средства и языки разработки) и щелкните на ссылке .NET Framework class library (Библиотека классов .NET Framework). После загрузки страницы щелкните на подузле .NET Framework class library узла интересующей версии платформы (предполагается версия 4.6). Далее можете использовать древовидное представление для навигации, чтобы просматривать пространства имен, типы и члены, относящиеся к платформе. На рис. 2.15 показан пример просмотра типов в пространстве имен `System`.

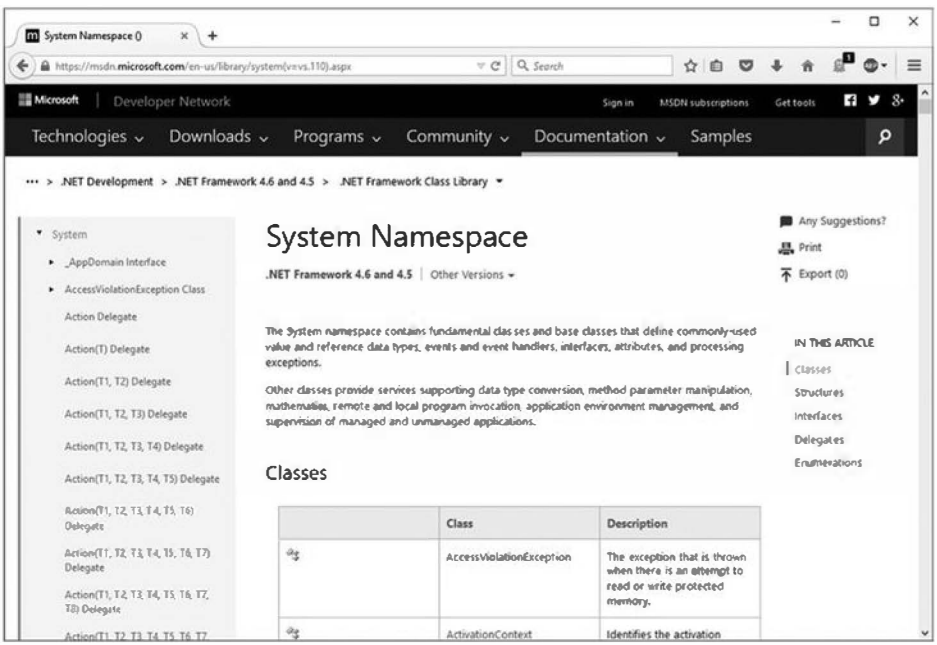


Рис. 2.15. Просмотр документации .NET Framework в онлайновом режиме

На заметку! Не будет удивительным, если в Microsoft когда-нибудь изменят местоположение онлайновой документации по библиотеке классов .NET Framework. В таком случае поиск в Интернете по ключевой фразе *.NET Framework Class Library documentation* поможет выяснить ее текущее местоположение.

В дополнение к онлайновой документации Visual Studio предоставляет возможность установки той же самой справочной системы локально на компьютере (что может быть удобно, когда активное подключение к Интернету отсутствует). Если вы хотите выпол-

нить локальную установку справочной системы после установки Visual Studio Community, то выберите пункт меню Help⇨Add and Remove Help Content (Справка⇨Добавить или удалить содержимое справочной системы). Затем можно выбрать любую справочную документацию для установки ее локально (если места на диске достаточно, то рекомендуется добавить всю возможную документацию). На рис. 2.16 можно видеть пример.

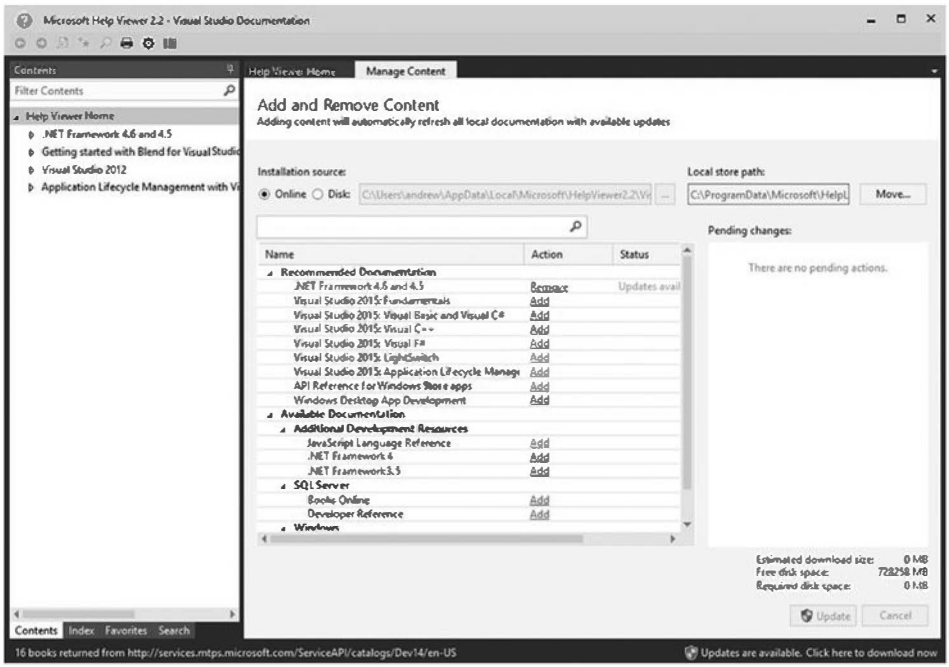


Рис. 2.16. Установка локальной справочной системы

После установки документации вы можете запустить приложение просмотра справочной системы (а также настроить Visual Studio на отображение справки (локальной или онлайновой)) с применением меню Help (Справка). Кроме того, еще более простой способ взаимодействия с документацией предусматривает выделение ключевого слова C#, имени типа либо имени члена в редакторе кода Visual Studio и нажатие клавиши <F1>. Это действие автоматически открывает окно с документацией для выбранного элемента. Например, выделите ключевое слово `string` внутри определения класса `Car`. После нажатия <F1> вы увидите страницу справки для типа `string`.

Еще одним полезным аспектом документации является вкладка Search (Поиск). Здесь вы можете ввести имя любого пространства имен, типа или члена и перейти к соответствующему месту в документации. Скажем, выполнив поиск документации для пространства имен `System.Reflection`, вы получите возможность изучить детали этого пространства имен, исследовать содержащиеся в нем типы, просмотреть примеры кода и т.д.

На заметку! Хотя это может выглядеть надоедливым повторением, нельзя еще раз не напомнить о том, насколько важно научиться пользоваться документацией .NET Framework. Ни одна книга, какой бы объемной она ни была, не способна охватить все аспекты платформы .NET. Непременно выделите время на освоение работы со справочной системой — впоследствии это окупится сторицей.

Построение приложений .NET за пределами среды Windows

Члены семейства Microsoft Visual Studio отличаются высокой мощностью и совершенством. Действительно, если ваш главный (или даже единственный) интерес заключается в построении программ .NET на машине Windows и их выполнении также на машине Windows, то Visual Studio, скорее всего, будет именно той IDE-средой, которая понадобится. Однако вспомните из главы 1, что платформа .NET может запускаться под управлением множества ОС. Учитывая это, давайте рассмотрим основную межплатформенную IDE-среду.

Роль Xamarin Studio

В главе 1 упоминалось о роли проекта Mono. Как вы должны помнить, он представляет собой межплатформенную реализацию .NET, поставляемую с многочисленными инструментами командной строки для построения программного обеспечения. Хотя полномасштабное приложение .NET можно было бы создавать с применением простого текстового редактора и компилятора C# командной строки, такой подход быстро станет довольно обременительным!

Продукт Xamarin Studio представляет собой бесплатную IDE-среду для .NET, которая выполняется под управлением ОС Windows, Mac OS X и Linux. Она похожа на Visual Studio Community тем, что поддерживает множество языков программирования (в том числе C#) и предлагает редакторы кода с развитой функциональностью, равно как визуальные отладчики и конструкторы графического пользовательского интерфейса. Если вы намереваетесь строить программное обеспечение .NET на машине, отличной от Microsoft Windows, то эта IDE-среда наверняка будет вашим избранным инструментом. Продукт Xamarin Studio доступен для загрузки по следующему URL:

```
http://xamarin.com/
```

На заметку! Вполне нормально устанавливать Xamarin Studio на машине, где также имеется Visual Studio Community. Тем не менее, для успешной установки Xamarin Studio обязательно завершите работу IDE-среды Visual Studio.

После установки Xamarin Studio можно создавать новое решение через пункт меню File⇨New⇨Solution (Файл⇨Создать⇨Решение). В открывшемся окне на выбор доступны разнообразные шаблоны и языки программирования. На рис. 2.17 демонстрируется выбор хорошо знакомого типа Console Application на C#.

Создав новое решение, вы должны отметить, что основные компоненты этой IDE-среды достаточно узнаваемы, учитывая все проведенные до сих пор исследования IDE-сред Visual Studio. На рис. 2.18 хорошо видно, что средство IntelliSense и навигация по проектам живут и здравствуют.

Поскольку IDE-среда Xamarin Studio способна выполняться под управлением ОС, отличных от Microsoft Windows, не должен вызывать удивления тот факт, что если она используется на машине Mac OS X или Linux, то имеет дело с исполняющей средой Mono и набором инструментов Mono. Однако Xamarin Studio может также нормально работать на машине Windows. В результате вы имеете возможность компилировать свой код C# либо для платформы Microsoft .NET, либо для платформы Mono (конечно, при условии, что платформа Mono была установлена). Для выбора целевой платформы применяйте пункт меню Tools⇨Options (Сервис⇨Параметры) и затем в открывшемся окне параметров указывайте нужный вариант на вкладке .NET Runtimes (Исполняющие среды .NET), как демонстрируется на рис. 2.19.

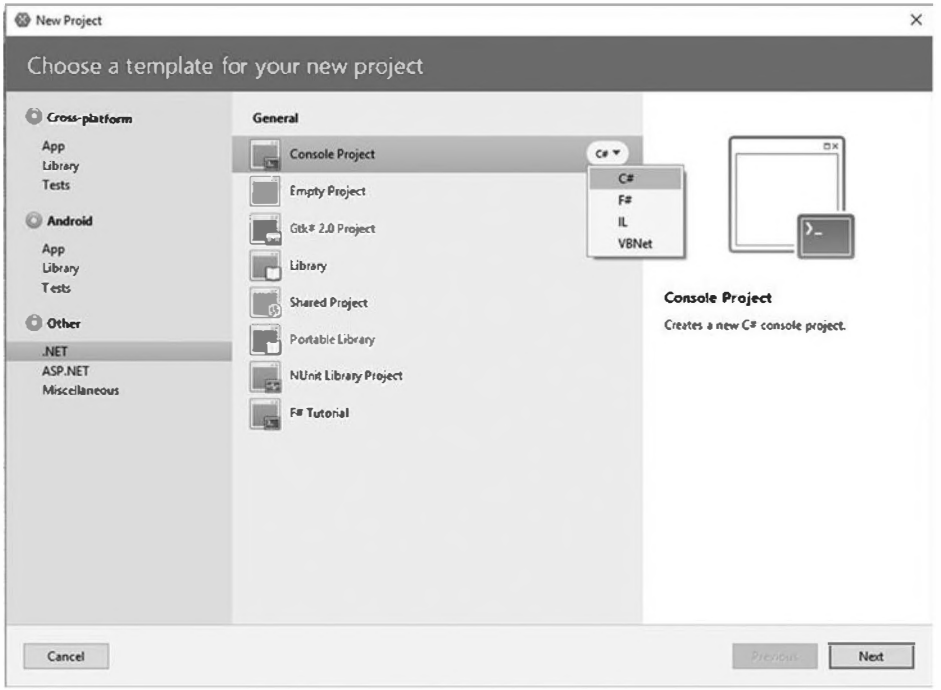


Рис. 2.17. Создание нового решения в Xamarin Studio

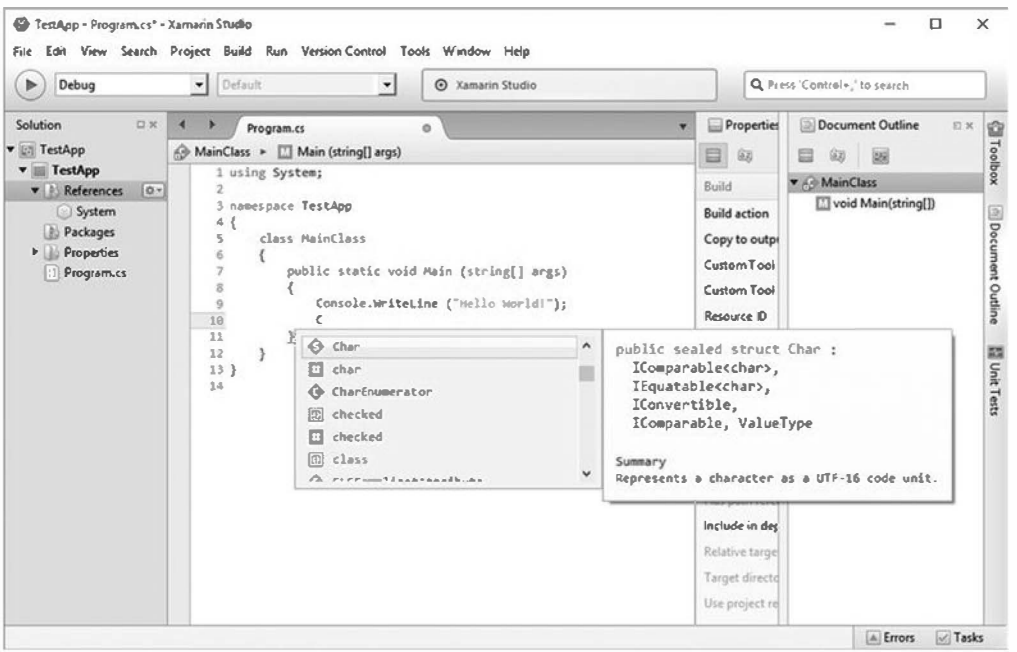


Рис. 2.18. Редактор кода в Xamarin Studio

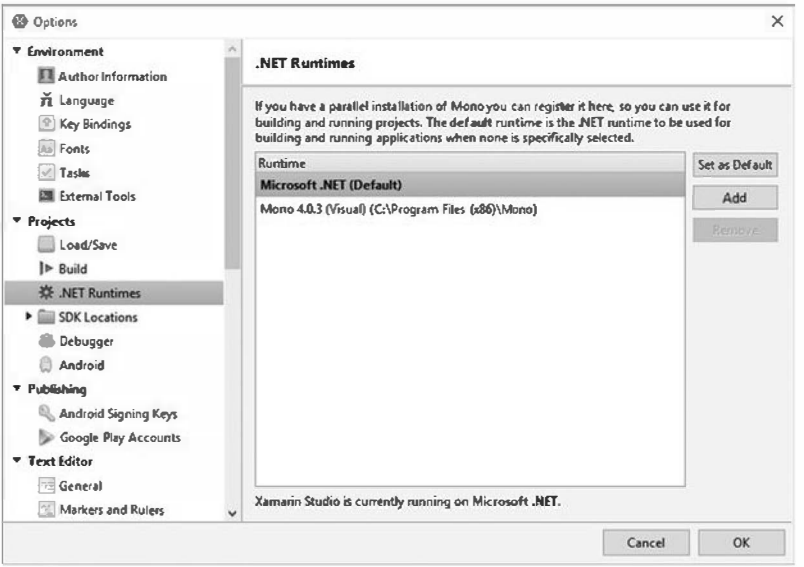


Рис. 2.19. Выбор исполняющей среды .NET в Xamarin Studio

Несмотря на то что IDE-среда Xamarin Studio в этой книге не используется, полезно знать, что большинство создаваемых проектов кода будут прекрасно функционировать в системах, отличных от Microsoft. Таким образом, если вы хотите прорабатывать материал книги в среде Mac OS X или Linux, то сможете делать это вполне гладко. Тем не менее, имейте в виду, что платформа Mono не поддерживает API-интерфейсы Windows Presentation Foundation (WPF), поэтому строить приложения WPF в среде, отличной от Microsoft, не получится. Однако Mono предлагает поддержку большей части API-интерфейсов, рассматриваемых в настоящей книге (у вас не возникнет никаких проблем при изучении глав 3–24 и большинства глав, посвященных веб-разработке).

На заметку! Более подробные сведения о том, какие аспекты Microsoft .NET полностью поддерживаются платформой Mono, можно найти в ее документации (`www.mono-project.com/docs/`).

Резюме

Как вы могли заметить, в вашем распоряжении появилось много новых игрушек! Целью этой главы было проведение краткого экскурса в основные средства, которые программист на языке C# может задействовать во время разработки. В главе указывалось, что если вас интересует только построение приложений .NET на машине разработки с ОС Windows, то лучшим выбором следует считать загрузку и установку Visual Studio Community Edition. Вдобавок также упоминалось о том, что в данном издании книги будет применяться именно эта конкретная IDE-среда, которая постоянно улучшается. Соответственно, предстоящие экранные снимки, пункты меню и визуальные конструкторы рассчитаны на то, что вы используете Visual Studio Community.

Если вы хотите создавать приложения .NET с применением платформы Mono или строить программное обеспечение на машине, отличной от Windows, то лучшим выбором окажется Xamarin Studio. Хотя эта IDE-среда не является идентичной Visual Studio Community, при проработке большей части материала книги с участием данного инструмента особые проблемы возникать не должны. Имея все это в виду, в главе 3 мы приступим к исследованию языка программирования C#.

ЧАСТЬ II

Основы программирования на C#

В этой части

ГЛАВА 3

Главные конструкции программирования на C#: часть I

В этой главе начинается формальное изучение языка программирования C# с представления набора отдельных тем, которые необходимо знать для освоения платформы .NET Framework. В первую очередь мы разберемся, каким образом строить *объект приложения*, и выясним структуру точки входа исполняемой программы — метода `Main()`. Затем мы исследуем фундаментальные типы данных C# (и их эквиваленты в пространстве имен `System`), в том числе классы `System.String` и `System.Text.StringBuilder`.

После ознакомления с деталями фундаментальных типов данных .NET мы рассмотрим несколько приемов преобразования типов данных, включая сужающие и расширяющие операции, а также использование ключевых слов `checked` и `unchecked`.

Кроме того, в главе будет описана роль ключевого слова `var` языка C#, которое позволяет *неявно* определять локальную переменную. Как будет показано далее в книге, неявная типизация чрезвычайно удобна (а порой и обязательна) при работе с набором технологий LINQ. Глава завершается кратким обзором ключевых слов и операций C#, которые дают возможность управлять последовательностью выполняемых в приложении действий с применением разнообразных конструкций циклов и принятия решений.

Структура простой программы C#

Язык C# требует, чтобы вся логика программы содержалась внутри определения типа (вспомните из главы 1, что *тип* — это общий термин, относящийся к любому члену из множества {класс, интерфейс, структура, перечисление, делегат}). В отличие от многих других языков создавать глобальные функции или глобальные элементы данных в языке C# невозможно. Вместо этого все данные-члены и все методы должны находиться внутри определения типа. Первым делом создадим новый проект консольного приложения по имени `SimpleCSharpApp`. Код в первоначальном файле `Program.cs` не особо примечателен:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace SimpleCSharpApp
{
  class Program
  {
    static void Main(string[] args)
    {
    }
  }
}
```

Теперь модифицируем метод `Main()` класса `Program` следующим образом:

```
class Program
{
  static void Main(string[] args)
  {
    // Вывести пользователю простое сообщение.
    Console.WriteLine("***** My First C# App *****");
    Console.WriteLine("Hello World!");
    Console.WriteLine();

    // Ожидать нажатия клавиши <Enter>, прежде чем завершить работу.
    Console.ReadLine();
  }
}
```

На заметку! Язык программирования C# чувствителен к регистру. Следовательно, `Main` — не то же самое, что и `main`, а `Readline` — не то же самое, что и `ReadLine`. Запомните, что все ключевые слова C# вводятся в нижнем регистре (например, `public`, `lock`, `class`, `dynamic`), в то время как названия пространств имен, типов и членов (по соглашению) начинаются с заглавной буквы и имеют заглавные буквы в любых содержащихся внутри словах (скажем, `Console.WriteLine`, `System.Windows.MessageBox` и `System.Data.SqlClient`). Как правило, каждый раз, когда вы получаете от компилятора сообщение об ошибке, касающееся неопределенных символов, то должны в первую очередь проверить правильность написания и регистр.

В предыдущем коде имеется определение типа класса, который поддерживает единственный метод по имени `Main()`. По умолчанию среда Visual Studio назначает классу, определяющему метод `Main()`, имя `Program`; однако при желании его можно изменить. Каждое исполняемое приложение C# (консольная программа, настольная программа Windows или Windows-служба) должно содержать класс, определяющий метод `Main()`, который используется для обозначения точки входа в приложение.

Говоря формально, класс, который определяет метод `Main()`, называется *объектом приложения*. Хотя в одном исполняемом приложении допускается иметь несколько объектов приложений (что может быть удобно при модульном тестировании), вы должны проинформировать компилятор о том, какой из методов `Main()` должен применяться в качестве точки входа. Это делается с помощью опции `/main` компилятора командной строки или посредством раскрывающегося списка Startup Object (Объект запуска) на вкладке Application (Приложение) окна свойств проекта Visual Studio (см. главу 2).

Обратите внимание, что сигнатура метода `Main()` снабжена ключевым словом `static`, которое подробно объясняется в главе 5. На данный момент достаточно знать, что статические члены имеют область видимости уровня класса (а не уровня объекта), поэтому могут вызываться без предварительного создания нового экземпляра класса.

Кроме наличия ключевого слова `static` данный метод `Main()` принимает единственный параметр, который представляет собой массив строк (`string[] args`). Несмотря на то что этот массив здесь никак не обрабатывается, параметр `args` может содержать

любое количество входных аргументов командной строки (доступ к ним описан ниже). И, наконец, метод `Main()` в примере был определен с возвращаемым значением `void`; это значит, что мы не задаем явно возвращаемое значение, используя ключевое слово `return`, перед выходом из области видимости метода.

Внутри метода `Main()` содержится логика класса `Program`. Мы работаем с классом `Console`, который определен в пространстве имен `System`. В состав его членов входит статический метод `WriteLine()`, который отправляет текстовую строку и символ возврата каретки в стандартный вывод. Кроме того, мы производим вызов метода `Console.ReadLine()`, чтобы окно командной строки, открываемое IDE-средой Visual Studio, оставалось видимым на протяжении сеанса отладки до тех пор, пока не будет нажата клавиша <Enter>. (Если вы не добавите такую строку кода, то приложение завершится прямо во время сеанса отладки, и вы не сможете просмотреть результирующий вывод!) Класс `System.Console` более подробно рассматривается далее в главе.

Вариации метода `Main()`

По умолчанию Visual Studio будет генерировать метод `Main()` с возвращаемым значением `void` и одним входным параметром в виде массива `string`. Тем не менее, это не единственно возможная форма метода `Main()`. Точку входа в приложение допускается создавать с использованием любой из приведенных далее сигнатур (предполагая, что они содержатся внутри определения класса или структуры C#):

```
// Возвращаемый тип int, массив строк в качестве параметра.
static int Main(string[] args)
{
  // Должен возвращать значение перед выходом!
  return 0;
}
// Нет возвращаемого типа, нет параметров.
static void Main()
{
}
// Возвращаемый тип int, нет параметров.
static int Main()
{
  // Должен возвращать значение перед выходом!
  return 0;
}
```

На заметку! Метод `Main()` может быть также определен как открытый в противоположность закрытому, что подразумевается, если не указан конкретный модификатор доступа. Среда Visual Studio определяет метод `Main()` как неявно закрытый.

Очевидно, что выбор способа создания метода `Main()` зависит от ответов на два вопроса. Во-первых, нужно ли возвращать значение системе, когда метод `Main()` заканчивается и работа программы завершается? Если да, то необходимо возвращать тип данных `int`, а не `void`. Во-вторых, требуется ли обрабатывать любые предоставляемые пользователем параметры командной строки? Если да, то они будут сохранены в массиве `string`. Давайте рассмотрим все варианты более подробно.

Указание кода ошибки приложения

Хотя в подавляющем большинстве случаев методы `Main()` будут иметь `void` в качестве возвращаемого значения, возможность возврата `int` из `Main()` сохраняет согласо-

ванность C# с другими языками, основанными на C. По соглашению возврат значения 0 указывает на то, что программа завершилась успешно, тогда как любое другое значение (вроде -1) представляет условие ошибки (имейте в виду, что значение 0 возвращается автоматически даже в случае, если метод Main() прототипирован для возвращения void).

В операционной системе Windows возвращаемое приложением значение сохраняется в переменной среды по имени %ERRORLEVEL%. Если создается приложение, которое программно запускает другой исполняемый файл (тема, рассматриваемая в главе 18), то получить значение %ERRORLEVEL% можно с применением статического свойства System.Diagnostics.Process.ExitCode.

Поскольку возвращаемое значение передается системе в момент завершения работы приложения, вполне очевидно, что получить и отобразить финальный код ошибки во время выполнения приложения невозможно. Однако мы покажем, как просмотреть код ошибки по завершении программы, изменив метод Main() следующим образом:

```
// Обратите внимание, что теперь возвращается int, а не void.
static int Main(string[] args)
{
  // Вывести сообщение и ожидать нажатия клавиши <Enter>.
  Console.WriteLine("***** My First C# App *****");
  Console.WriteLine("Hello World!");
  Console.WriteLine();
  Console.ReadLine();
  // Возвратить произвольный код ошибки.
  return -1;
}
```

Теперь давайте захватим возвращаемое значение метода Main() с помощью пакетного файла. Используя проводник Windows, перейдите в папку, содержащую ваше скомпилированное приложение (например, C:\SimpleCSharpApp\bin\Debug). Добавьте в папку Debug новый текстовый файл (по имени SimpleCSharpApp.bat), который содержит приведенные далее инструкции (если раньше вам не приходилось создавать файлы .bat, то можете не беспокоиться о внутренних нюансах — это всего лишь тест):

```
@echo off

rem Пакетный файл для приложения SimpleCSharpApp.exe,
rem в котором захватывается возвращаемое им значение.

SimpleCSharpApp
@if "%ERRORLEVEL%" == "0" goto success

:fail
  rem Приложение потерпело неудачу.
  echo This application has failed!
  echo return value = %ERRORLEVEL%
  goto end
:success
  rem Приложение успешно завершено.
  echo This application has succeeded!
  echo return value = %ERRORLEVEL%
  goto end
:end
rem Работа сделана.
echo All Done.
```

В этот момент откройте окно командной строки и перейдите в папку, содержащую исполняемый файл и новый файл *.bat. Запустите пакетный файл, набрав его имя и нажав <Enter>. Вы должны увидеть показанный ниже вывод, учитывая, что метод

Main() возвращает -1. Если бы он возвращал 0, то вы увидели бы в окне консоли сообщение This application has succeeded!.

```
***** My First C# App *****
Hello World!

This application has failed!
return value = -1
All Done.
```

В подавляющем большинстве приложений C# (если только не во всех) в качестве возвращаемого значения метода Main() будет применяться void, что, как известно, подразумевает неявное возвращение кода ошибки, равного 0. Поэтому все методы Main() в книге (кроме текущего примера) будут в действительности возвращать void (и последующие проекты определенно не будут нуждаться в использовании пакетных файлов для перехвата кодов возврата).

Обработка аргументов командной строки

Теперь, когда вы лучше понимаете, что собой представляет возвращаемое значение метода Main(), давайте посмотрим на входной массив данных string. Предположим, что нам необходимо модифицировать приложение для обработки любых возможных параметров командной строки. Один из способов сделать это предусматривает применение цикла for языка C#. (Все итерационные конструкции C# более подробно рассматриваются в конце главы.)

```
static int Main(string[] args)
{
  ...
  // Обработать любые входные аргументы.
  for(int i = 0; i < args.Length; i++)
    Console.WriteLine("Arg: {0}", args[i]);
  Console.ReadLine();
  return -1;
}
```

Здесь с использованием свойства Length класса System.Array производится проверка, содержит ли массив string какие-то элементы. Как будет показано в главе 4, все массивы C# на самом деле являются псевдонимом класса System.Array и потому разделяют общий набор членов. По мере прохода в цикле по всем элементам массива их значения выводятся на консоль. Предоставить аргументы в командной строке сравнительно просто:

```
C:\SimpleCSharpApp\bin\Debug>SimpleCSharpApp.exe /arg1 -arg2

***** My First C# App *****
Hello World!
Arg: /arg1
Arg: -arg2
```

В качестве альтернативы стандартному циклу for для реализации прохода по входному массиву данных string можно также применять ключевое слово foreach. Вот пример использования foreach (особенности конструкций циклов обсуждаются далее в этой главе):

```
// Обратите внимание, что в случае применения foreach
// отпадает необходимость в проверке размера массива.
static int Main(string[] args)
{
  ...
```

```
  // Обработать любые входные аргументы, используя foreach.
  foreach(string arg in args)
    Console.WriteLine("Arg: {0}", arg);

  Console.ReadLine();
  return -1;
}
```

Наконец, доступ к аргументам командной строки можно также получать с помощью статического метода `GetCommandLineArgs()` типа `System.Environment`. Этот метод возвращает массив элементов `string`. Первый элемент содержит имя самого приложения, а остальные — индивидуальные аргументы командной строки. Обратите внимание, что при таком подходе больше не обязательно определять метод `Main()` как принимающий массив `string` во входном параметре, хотя никакого вреда от этого не будет.

```
static int Main(string[] args)
{
  ...
  // Получить аргументы с использованием System.Environment.
  string[] theArgs = Environment.GetCommandLineArgs();
  foreach(string arg in theArgs)
    Console.WriteLine("Arg: {0}", arg);

  Console.ReadLine();
  return -1;
}
```

Разумеется, именно на вас возлагается решение о том, на какие аргументы командной строки должна реагировать программа (если они вообще будут предусмотрены), и как они должны быть сформатированы (например, с префиксом - или /). В показанном выше коде мы просто передаем последовательность аргументов, которые выводятся прямо в окно командной строки. Однако предположим, что создается новое игровое приложение, запрограммированное на обработку параметра вида `-godmode`. Когда пользователь запускает это приложение с таким флагом, в его отношении можно было бы предпринять соответствующие действия.

Указание аргументов командной строки в Visual Studio

В реальности конечный пользователь при запуске программы имеет возможность предоставлять аргументы командной строки. Тем не менее, указывать допустимые флаги командной строки также может требоваться во время разработки в целях тестирования программы. Чтобы сделать это в Visual Studio, дважды щелкните на значке Properties (Свойства) в проводнике решений. Затем в открывшемся окне свойств перейдите на вкладку Debug (Отладка) и в текстовом поле Command line arguments (Аргументы командной строки) введите желаемые аргументы (рис. 3.1).

Указанные аргументы командной строки будут автоматически передаваться методу `Main()` во время отладки или запуска приложения внутри IDE-среды Visual Studio.

Интересное отступление от темы: некоторые дополнительные члены класса `System.Environment`

Помимо метода `GetCommandLineArgs()` класс `Environment` открывает доступ к ряду других чрезвычайно полезных методов. В частности, с помощью разнообразных статических членов этот класс позволяет получать детальные сведения, касающиеся операционной системы, под управлением которой в текущий момент функционирует приложение .NET.

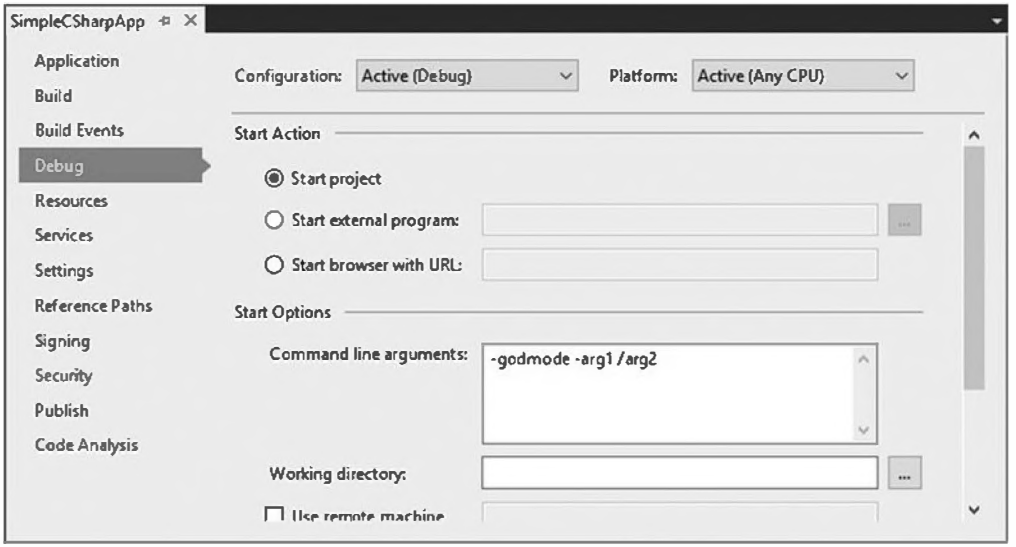


Рис. 3.1. Указание аргументов командной строки в Visual Studio

Чтобы проиллюстрировать полезность класса `System.Environment`, **изменим код метода** `Main()`, **добавив вызов вспомогательного метода по имени** `ShowEnvironmentDetails()`:

```
static int Main(string[] args)
{
  ...
  // Вспомогательный метод внутри класса Program.
  ShowEnvironmentDetails();
  Console.ReadLine();
  return -1;
}
```

Теперь реализуем этот метод внутри класса `Program` **для обращения в нем к разным членам типа** `Environment`:

```
static void ShowEnvironmentDetails()
{
  // Вывести информацию о дисковых устройствах
  // данной машины и другие интересные детали.
  foreach (string drive in Environment.GetLogicalDrives())
    Console.WriteLine("Drive: {0}", drive);      // Логические устройства
  Console.WriteLine("OS: {0}", Environment.OSVersion);  // Версия
                                                  // операционной системы
  Console.WriteLine("Number of processors: {0}",
    Environment.ProcessorCount);                  // Количество процессоров
  Console.WriteLine(".NET Version: {0}",
    Environment.Version);                         // Версия платформы .NET
}
```

Ниже показан возможный вывод, полученный в результате тестового запуска данного метода. Конечно, если на вкладке Debug в Visual Studio не были указаны аргументы командной строки, то они и не будут отображены в окне консоли.

```
***** My First C# App *****

Hello World!

Arg: -godmode
Arg: -arg1
Arg: /arg2

Drive: C:\
Drive: D:\
OS: Microsoft Windows NT 6.2.9200.0
Number of processors: 8
.NET Version: 4.0.30319.42000
```

В типе `Environment` определены и другие члены помимо задействованных в предыдущем примере. В табл. 3.1 описаны некоторые интересные дополнительные свойства; полные сведения о них можно найти в документации .NET Framework 4.6 SDK.

Таблица 3.1. Избранные свойства типа `System.Environment`

Свойство	Описание
`ExitCode`	Получает или устанавливает код возврата для приложения
`Is64BitOperatingSystem`	Возвращает булевское значение, которое представляет признак наличия на текущей машине 64-разрядной операционной системы
`MachineName`	Возвращает имя текущей машины
`NewLine`	Возвращает символ новой строки для текущей среды
`SystemDirectory`	Возвращает полный путь к каталогу системы
`UserName`	Возвращает имя пользователя, запустившего данное приложение
`Version`	Возвращает объект `Version`, который представляет версию платформы .NET

Исходный код. Проект **`SimpleCSharpApp`** доступен в подкаталоге **`Chapter_3`**.

Класс `System.Console`

Почти во всех примерах приложений, создаваемых на протяжении начальных глав книги, будет интенсивно применяться класс `System.Console`. Справедливо заметить, что консольный пользовательский интерфейс может выглядеть не настолько привлекательно, как графический пользовательский интерфейс либо интерфейс веб-приложения. Однако ограничение первоначальных примеров консольными программами позволяет сосредоточиться на синтаксисе C# и ключевых аспектах платформы .NET, не отвлекаясь на сложности, которыми сопровождается построение настольных графических пользовательских интерфейсов или веб-сайтов.

Класс `Console` инкапсулирует средства манипулирования потоками ввода, вывода и ошибок для консольных приложений. В табл. 3.2 перечислены некоторые (но определенно не все) интересные его члены. Как видите, в классе `Console` имеется ряд членов, которые оживляют простые приложения командной строки, позволяя, например, изменять цвета фона и переднего плана и выдавать звуковые сигналы (еще и различной частоты).

Таблица 3.2. Избранные члены класса `System.Console`

Член	Описание
`Beep()`	Этот метод заставляет консоль выдать звуковой сигнал указанной частоты и длительности
`BackgroundColor` `ForegroundColor`	Эти свойства устанавливают цвета фона и переднего плана для текущего вывода. Им может быть присвоен любой член перечисления `ConsoleColor`
`BufferHeight` `BufferWidth`	Эти свойства управляют высотой и шириной буферной области консоли
`Title`	Это свойство получает или устанавливает заголовок текущей консоли

`WindowHeight`
`WindowWidth`
`WindowTop`
`WindowLeft`
	Эти свойства управляют размерами консоли по отношению к установленному буферу

`Clear()`	Этот метод очищает установленный буфер и область отображения консоли

Базовый ввод-вывод с помощью класса Console

Дополнительно к членам, описанным в табл. 3.2, класс `Console` определяет набор методов для захвата ввода и вывода; все они являются статическими и потому вызываются с префиксом в виде имени класса (`Console`). Как вы уже видели, метод `WriteLine()` помещает в поток вывода строку текста (включая символ возврата каретки). Метод `Write()` помещает в поток вывода текст без символа возврата каретки. Метод `ReadLine()` позволяет получить информацию из потока ввода вплоть до нажатия клавиши <Enter>. Метод `Read()` используется для захвата одиночного символа из потока ввода.

Чтобы продемонстрировать реализацию базового ввода-вывода с применением класса `Console`, создадим новый проект консольного приложения по имени `BasicConsoleIO` и модифицируем метод `Main()` для вызова вспомогательного метода `GetUserData()`:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Basic Console I/O *****");
    GetUserData();
    Console.ReadLine();
  }
  private static void GetUserData()
  {
  }
}
```

На заметку! Среда Visual Studio поддерживает несколько "фрагментов кода", которые после своей активизации вставляют код. Фрагмент кода `cw` очень полезен в начальных главах книги, т.к. он автоматически разворачивается в метод `Console.WriteLine()`. Чтобы удостовериться в этом, введите `cw` где-нибудь внутри метода `Main()` и два раза нажмите клавишу <Tab> (к сожалению, фрагмент кода для метода `Console.ReadLine()` отсутствует). Чтобы просмотреть все фрагменты кода, щелкните правой кнопкой мыши на файле кода C# и в открывшемся контекстном меню выберите пункт Insert Snippet (Вставить фрагмент кода).

Теперь реализуем метод GetUserData() внутри класса Program, поместив в него логику, которая приглашает пользователя ввести некоторые сведения и затем дублирует их в стандартный поток вывода. Скажем, мы могли бы запросить у пользователя его имя и возраст (который для простоты будет трактоваться как текстовое значение, а не привычное числовое):

```
static void GetUserData()
{
  // Получить информацию об имени и возрасте.
  Console.Write("Please enter your name: ");  // Предложить ввести имя
  string userName = Console.ReadLine();
  Console.Write("Please enter your age: ");   // Предложить ввести возраст
  string userAge = Console.ReadLine();

  // Просто ради забавы изменить цвет переднего плана.
  ConsoleColor prevColor = Console.ForegroundColor;
  Console.ForegroundColor = ConsoleColor.Yellow;

  // Вывести полученную информацию на консоль.
  Console.WriteLine("Hello {0}! You are {1} years old.",
  userName, userAge);

  // Восстановить предыдущий цвет переднего плана.
  Console.ForegroundColor = prevColor;
}
```

После запуска этого приложения входные данные вполне предсказуемо будут выводиться в окно консоли (с использованием указанного специального цвета).

Форматирование консольного вывода

В ходе изучения первых нескольких глав вы могли заметить, что внутри различных строковых литералов часто встречались такие конструкции, как {0} и {1}. Платформа .NET поддерживает стиль форматирования строк, который немного напоминает стиль, применяемый в операторе printf() языка C. Попросту говоря, когда вы определяете строковый литерал, содержащий сегменты данных, значения которых остаются неизвестными до этапа выполнения, то имеете возможность указывать заполнитель, используя синтаксис с фигурными скобками. Во время выполнения все заполнители замещаются значениями, передаваемыми методу Console.WriteLine().

Первый параметр метода WriteLine() представляет строковый литерал, который содержит заполнители, определяемые с помощью {0}, {1}, {2} и т.д. Запомните, что порядковые числа заполнителей в фигурных скобках всегда начинаются с 0. Остальные параметры WriteLine() — это просто значения, подлежащие вставке вместо соответствующих заполнителей.

На заметку! Если уникально нумерованных заполнителей больше, чем заполняющих аргументов, то во время выполнения будет сгенерировано исключение, связанное с форматом. Однако если количество заполняющих аргументов превышает число заполнителей, то лишние аргументы игнорируются.

Отдельный заполнитель допускается повторять внутри заданной строки. Например, если вы битломан и хотите построить строку "9, Number 9, Number 9", то могли бы написать такой код:

```
// Джон говорит...
Console.WriteLine("{0}, Number {0}, Number {0}", 9);
```

Также вы должны знать о возможности помещения каждого заполнителя в любую позицию внутри строкового литерала. К тому же вовсе не обязательно, чтобы заполнители следовали в возрастающем порядке своих номеров, например:

```
// Выводит: 20, 10, 30
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
```

Форматирование числовых данных

Если для числовых данных требуется более сложное форматирование, то каждый заполнитель может дополнительно содержать разнообразные символы форматирования, наиболее распространенные из которых описаны в табл. 3.3.

Таблица 3.3. Символы для форматирования числовых данных в .NET

Символ форматирования	Описание
C или c	Используется для форматирования денежных значений. По умолчанию значение предваряется символом локальной валюты (например, знаком доллара ($) для культуры US English)
D или d	Используется для форматирования десятичных чисел. В этом флаге можно также указывать минимальное количество цифр для представления значения
E или e	Используется для экспоненциального представления. Регистр этого флага указывает, в каком регистре должна представляться экспоненциальная константа — в верхнем (E) или в нижнем (e)
F или f	Используется для форматирования с фиксированной точкой. В этом флаге можно также указывать минимальное количество цифр для представления значения
G или g	Обозначает общий (*general*) формат. Этот флаг может использоваться для представления чисел в формате с фиксированной точкой или экспоненциальном формате
N или n	Используется для базового числового форматирования (с запятыми)
X или x	Используется для шестнадцатеричного форматирования. В случае символа X в верхнем регистре шестнадцатеричное представление будет содержать символы верхнего регистра

Эти символы форматирования добавляются к заполнителям в виде суффиксов после двоеточия (например, `{0:C}`, `{1:d}`, `{2:X}`). В целях иллюстрации изменим метод `Main()` для вызова нового вспомогательного метода по имени `FormatNumericalData()`. Реализация этого метода в классе `Program` форматирует фиксированное числовое значение несколькими способами.

```
// Демонстрация применения некоторых дескрипторов формата.
static void FormatNumericalData()
{
  Console.WriteLine("The value 99999 in various formats:");
  Console.WriteLine("c format: {0:c}", 99999);
  Console.WriteLine("d9 format: {0:d9}", 99999);
  Console.WriteLine("f3 format: {0:f3}", 99999);
  Console.WriteLine("n format: {0:n}", 99999);

  // Обратите внимание, что использование для символа шестнадцатеричного формата
  // верхнего или нижнего регистра определяет регистр отображаемых символов.
  Console.WriteLine("E format: {0:E}", 99999);
```

```
  Console.WriteLine("e format: {0:e}", 99999);
  Console.WriteLine("X format: {0:X}", 99999);
  Console.WriteLine("x format: {0:x}", 99999);
}
```

Ниже показан вывод, получаемый в результате вызова метода `FormatNumericalData()`:

```
The value 99999 in various formats:
c format: $99,999.00
d9 format: 000099999
f3 format: 99999.000
n format: 99,999.00
E format: 9.999900E+004
e format: 9.999900e+004
X format: 1869F
x format: 1869f
```

В дальнейшем будут встречаться и другие примеры форматирования; если вас интересует дополнительные сведения о форматировании строк в .NET, обращайтесь в документацию .NET Framework 4.6 SDK.

Исходный код. Проект `BasicConsoleIO` доступен в подкаталоге `Chapter_3`.

Форматирование числовых данных за рамками консольных приложений

Напоследок следует отметить, что применение символов форматирования строк .NET не ограничено консольными приложениями. Тот же самый синтаксис форматирования может быть использован при вызове статического метода `string.Format()`. Это удобно, когда необходимо формировать выходные текстовые данные во время выполнения в приложении любого типа (например, в настольном приложении с графическим пользовательским интерфейсом, веб-приложении ASP.NET и т.д.).

Метод `string.Format()` возвращает новый объект `string`, который форматируется согласно предоставляемым флагам. Затем текстовые данные могут применяться любым желаемым образом. Для примера предположим, что требуется создать графическое настольное приложение WPF и сформатировать строку, подлежащую отображению в окне сообщения. В приведенном ниже коде показано, как это сделать, но имейте в виду, что код не скомпилируется до тех пор, пока в проект не будет добавлена ссылка на сборку `PresentationFramework.dll` (добавление ссылок на библиотеки в Visual Studio рассматривалось в главе 2).

```
static void DisplayMessage()
{
  // Использование string.Format() для форматирования строкового литерала.
  string userMessage = string.Format("100000 in hex is {0:x}", 100000);
  // Для успешной компиляции этой строки кода требуется
  // ссылка на библиотеку PresentationFramework.dll!
  System.Windows.MessageBox.Show(userMessage);
}
```

На заметку! В версии .NET 4.6 для представления заполнителей в фигурных скобках появился альтернативный синтаксис, который называется интерполяцией строк. Мы исследуем этот подход позже в главе.

Системные типы данных и соответствующие ключевые слова C#

Подобно любому языку программирования в C# для фундаментальных типов данных определены ключевые слова, которые используются при представлении локальных переменных, переменных-членов данных в классах, возвращаемых значений и параметров методов. Тем не менее, в отличие от других языков программирования такие ключевые слова в C# являются чем-то большим, нежели просто лексемами, распознаваемыми компилятором. В действительности они представляют собой сокращенные обозначения полноценных типов из пространства имен `System`. В табл. 3.4 перечислены системные типы данных вместе с их диапазонами значений, соответствующими ключевыми словами C# и сведениями о совместимости с общеязыковой спецификацией (CLS).

На заметку! В главе 1 говорилось о том, что совместимый с CLS код .NET может быть задействован в любом управляемом языке программирования. Если в программах открыт доступ к данным, не совместимым с CLS, то другие языки могут быть не в состоянии их использовать.

Таблица 3.4. Внутренние типы данных C#

Сокращенное обозначение в C#	Совместимость с CLS	Системный тип	Диапазон	Описание
`bool`	Да	`System.Boolean`	`true` или `false`	Признак истинности или ложности
`sbyte`	Нет	`System.SByte`	от –128 до 127	8-битное число со знаком
`byte`	Да	`System.Byte`	от 0 до 255	8-битное число без знака
`short`	Да	`System.Int16`	от –32 768 до 32 767	16-битное число со знаком
`ushort`	Нет	`System.UInt16`	от 0 до 65 535	16-битное число без знака
`int`	Да	`System.Int32`	от –2 147 483 648 до 2 147 483 647	32-битное число со знаком
`uint`	Нет	`System.UInt32`	от 0 до 4 294 967 295	32-битное число без знака
`long`	Да	`System.Int64`	от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	64-битное число со знаком
`ulong`	Нет	`System.UInt64`	от 0 до 18 446 744 073 709 551 615	64-битное число без знака
`char`	Да	`System.Char`	от U+0000 до U+ffff	Одиночный 16-битный символ Unicode
`float`	Да	`System.Single`	от -3.4×10^{38} до $+3.4\times10^{38}$	32-битное число с плавающей точкой
`double`	Да	`System.Double`	от $\pm5.0\times10^{-324}$ до $\pm1.7\times10^{308}$	64-битное число с плавающей точкой

Окончание табл. 3.4

Сокращенное обозначение в C#	Совместимость с CLS	Системный тип	Диапазон	Описание
decimal	Да	System.Decimal	(от -7.9×10^{28} до 7.9×10^{28}) / ($10^{от\ 0\ до\ 28}$)	128-битное число со знаком
string	Да	System.String	Ограничен объемом системной памяти	Представляет набор символов Unicode
object	Да	System.Object	В переменной object может храниться любой тип данных	Базовый класс для всех типов в мире .NET

На заметку! По умолчанию число с плавающей точкой трактуется как double. Чтобы объявить значение типа float, применяйте суффикс f или F к неформатированному числовому значению (5.3F), а чтобы объявить значение типа decimal применяйте суффикс m или M к числу с плавающей точкой (300.5M). И, наконец, неформатированные целые числа по умолчанию относятся к типу int. Чтобы получить значение типа long, понадобится снабдить его суффиксом l или L (4L).

Объявление и инициализация переменных

Для объявления локальной переменой (например, переменной внутри области видимости члена) необходимо указать тип данных, за которым следует имя переменной. Давайте создадим новый проект консольного приложения по имени BasicDataTypes и модифицируем класс Program так, чтобы в его методе Main() вызывался следующий вспомогательный метод:

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // Локальные переменные объявляются так:
  // типДанных имяПеременной;
  int myInt;
  string myString;
  Console.WriteLine();
}
```

Имейте в виду, что использование локальной переменной до присваивания ей начального значения приведет к *ошибке на этапе компиляции*. Учитывая это, рекомендуется присваивать начальные значения локальным переменным непосредственно при их объявлении. Это можно делать в одной строке или разносить объявление и присваивание на два отдельных оператора кода.

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // Локальные переменные объявляются и инициализируются так:
  // типДанных имяПеременной = начальноеЗначение;
  int myInt = 0;
  // Объявлять и присваивать можно также в двух отдельных строках.
  string myString;
  myString = "This is my character data";
  Console.WriteLine();
}
```

Также разрешено объявлять несколько переменных того же самого типа в одной строке кода, как в случае следующих трех переменных bool:

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  int myInt = 0;
  string myString;
  myString = "This is my character data";
  // Объявить три переменных типа bool в одной строке.
  bool b1 = true, b2 = false, b3 = b1;
  Console.WriteLine();
}
```

Поскольку ключевое слово bool в C# — просто сокращенное обозначение структуры System.Boolean, то любой тип данных можно указывать с применением его полного имени (естественно, это же касается всех остальных ключевых слов C#, представляющих типы данных). Ниже приведена окончательная реализация метода LocalVarDeclarations(), в которой демонстрируются разнообразные способы объявления локальных переменных:

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // Локальные переменные объявляются и инициализируются так:
  // типДанных имяПеременной = начальноеЗначение;
  int myInt = 0;
  string myString;
  myString = "This is my character data";
  // Объявить три переменных типа bool в одной строке.
  bool b1 = true, b2 = false, b3 = b1;
  // Использовать тип данных System.Boolean для объявления булевской переменной.
  System.Boolean b4 = false;
  Console.WriteLine("Your data: {0}, {1}, {2}, {3}, {4}, {5}",
     myInt, myString, b1, b2, b3, b4);

  Console.WriteLine();
}
```

Внутренние типы данных и операция new

Все внутренние типы данных поддерживают так называемый *стандартный конструктор* (см. главу 5). Это средство позволяет создавать переменную, используя ключевое слово new, что автоматически устанавливает переменную в ее стандартное значение:

- переменные типа bool устанавливаются в false;
- переменные числовых типов устанавливаются в 0 (или в 0.0 для типов с плавающей точкой);
- переменные типа char устанавливаются в одиночный пустой символ;
- переменные типа BigInteger устанавливаются в 0;
- переменные типа DateTime устанавливаются в 1/1/0001 12:00:00 AM;
- объектные ссылки (включая переменные типа string) устанавливаются в null.

На заметку! Тип данных `BigInteger`, упомянутый в приведенном выше списке, будет описан чуть позже.

Применение ключевого слова `new` при создании переменных базовых типов дает более громоздкий, но синтаксически корректный код C#:

```
static void NewingDataTypes()
{
  Console.WriteLine("=> Using new to create variables:");
  bool b = new bool();              // Устанавливается в false.
  int i = new int();                // Устанавливается в 0.
  double d = new double();          // Устанавливается в 0.
  DateTime dt = new DateTime();     // Устанавливается в 1/1/0001 12:00:00 AM
  Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
  Console.WriteLine();
}
```

Иерархия классов для типов данных

Интересно отметить, что даже элементарные типы данных в .NET организованы в *иерархию классов*. Если вы не знакомы с концепцией наследования, то найдете все необходимые сведения в главе 6. А до тех пор просто знайте, что типы, находящиеся в верхней части иерархии классов, предоставляют определенное стандартное поведение, которое передается производным типам. На рис. 3.2 показаны отношения между основными системными типами.

Обратите внимание, что каждый тип в конечном итоге оказывается производным от класса `System.Object`, в котором определен набор методов (например, `ToString()`, `Equals()`, `GetHashCode()`), общих для всех типов из библиотек базовых классов .NET (эти методы подробно рассматриваются в главе 6).

Также важно отметить, что многие числовые типы данных являются производными от класса `System.ValueType`. Потомки `ValueType` автоматически размещаются в стеке и по этой причине имеют предсказуемое время жизни и довольно эффективны. С другой стороны, типы, в цепочке наследования которых класс `System.ValueType` отсутствует (такие как `System.Type`, `System.String`, `System.Array`, `System.Exception` и `System.Delegate`), размещаются не в стеке, а в куче с автоматической сборкой мусора. (Более подробно это различие обсуждается в главе 4.)

Не вдаваясь глубоко в детали классов `System.Object` и `System.ValueType`, важно уяснить, что поскольку любое ключевое слово C# (скажем, `int`) представляет собой просто сокращенное обозначение соответствующего системного типа (в этом случае `System.Int32`), то приведенный ниже синтаксис совершенно законен. Дело в том, что тип `System.Int32` (`int` в C#) в конечном итоге является производным от класса `System.Object` и, следовательно, может обращаться к любому из его открытых членов, как продемонстрировано в еще одной вспомогательной функции:

```
static void ObjectFunctionality()
{
  Console.WriteLine("=> System.Object Functionality:");
  // Ключевое слово int языка C# - это в действительности сокращение для
  // типа System.Int32, который наследует от System.Object следующие члены:
  Console.WriteLine("12.GetHashCode() = {0}", 12.GetHashCode());
  Console.WriteLine("12.Equals(23) = {0}", 12.Equals(23));
  Console.WriteLine("12.ToString() = {0}", 12.ToString());
  Console.WriteLine("12.GetType() = {0}", 12.GetType());
  Console.WriteLine();
}
```

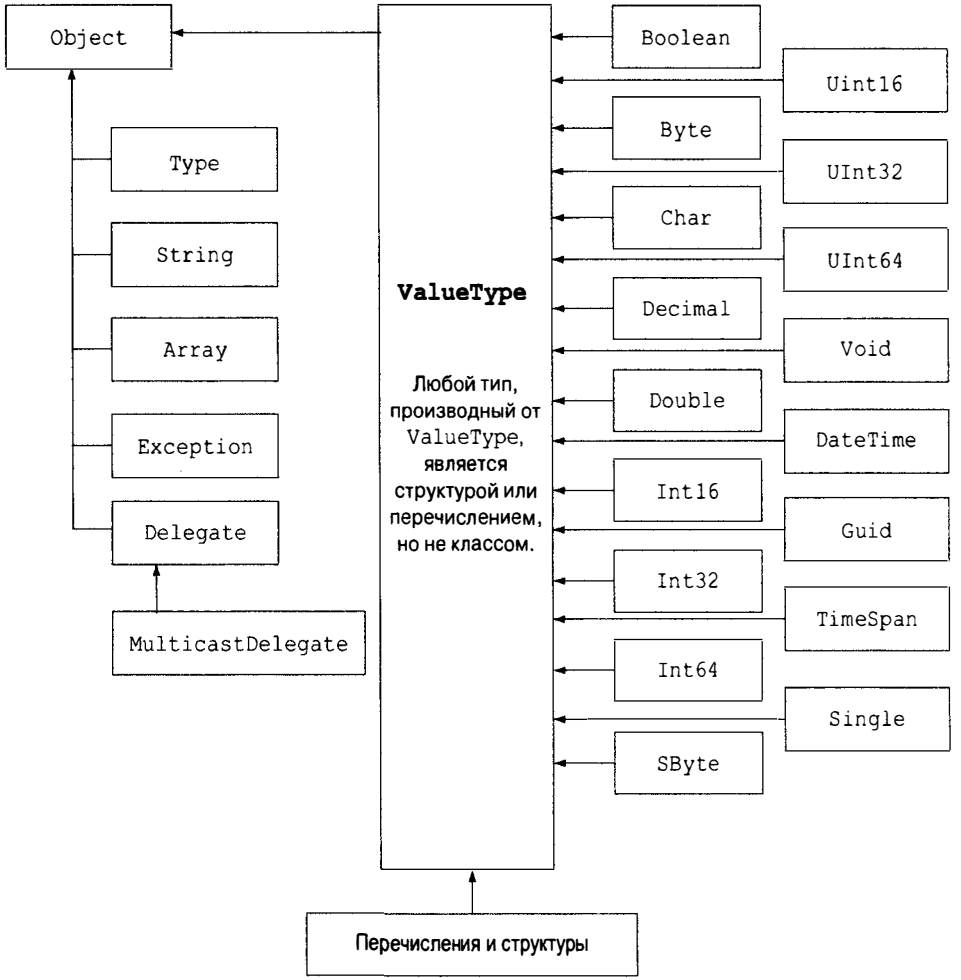


Рис. 3.2. Иерархия классов для системных типов

Вызов этого метода внутри `Main()` дает такой вывод:

```
=> System.Object Functionality:
12.GetHashCode() = 12
12.Equals(23) = False
12.ToString() = 12
12.GetType() = System.Int32
```

Члены числовых типов данных

Продолжая эксперименты со встроенными типами данных C#, следует отметить, что числовые типы .NET поддерживают свойства `MaxValue` и `MinValue`, предоставляющие информацию о диапазоне значений, которые способен хранить конкретный тип. В дополнение к свойствам `MinValue` и `MaxValue` каждый числовой тип может определять собственные полезные члены. Например, тип `System.Double` позволяет получать значения для бесконечно малой (эпсилон) и бесконечно большой величин (которые интересны тем, кто занимается решением математических задач).

В целях иллюстрации рассмотрим следующую вспомогательную функцию:

```
static void DataTypeFunctionality()
{
  Console.WriteLine("=> Data type Functionality:");
  Console.WriteLine("Max of int: {0}", int.MaxValue);
  Console.WriteLine("Min of int: {0}", int.MinValue);
  Console.WriteLine("Max of double: {0}", double.MaxValue);
  Console.WriteLine("Min of double: {0}", double.MinValue);
  Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
  Console.WriteLine("double.PositiveInfinity: {0}",
    double.PositiveInfinity);
  Console.WriteLine("double.NegativeInfinity: {0}",
    double.NegativeInfinity);
  Console.WriteLine();
}
```

Члены System.Boolean

Рассмотрим тип данных `System.Boolean`. К допустимым значениям, которые могут присваиваться типу `bool` в C#, относятся только `true` и `false`. С учетом этого должно быть понятно, что `System.Boolean` не поддерживает свойства `MinValue` и `MaxValue`, но вместо них определяет свойства `TrueString` и `FalseString` (которые выдают, соответственно, строки `"True"` и `"False"`). Вот пример:

```
Console.WriteLine("bool.FalseString: {0}", bool.FalseString);
Console.WriteLine("bool.TrueString: {0}", bool.TrueString);
```

Члены System.Char

Текстовые данные в C# представляются посредством ключевых слов `string` и `char`, которые являются сокращенными обозначениями для типов `System.String` и `System.Char` (оба основаны на Unicode). Как вам уже может быть известно, `string` представляет непрерывное множество символов (например, `"Hello"`), а `char` — одиночную ячейку в `string` (например, `'H'`).

Помимо возможности хранения одиночного элемента символьных данных тип `System.Char` предлагает немало другой функциональности. Используя статические методы `System.Char`, можно выяснять, является данный символ цифрой, буквой, знаком пунктуации или чем-то еще. Рассмотрим следующий метод:

```
static void CharFunctionality()
{
  Console.WriteLine("=> char type Functionality:");
  char myChar = 'a';
  Console.WriteLine("char.IsDigit('a'): {0}", char.IsDigit(myChar));
  Console.WriteLine("char.IsLetter('a'): {0}", char.IsLetter(myChar));
  Console.WriteLine("char.IsWhiteSpace('Hello There', 5): {0}",
    char.IsWhiteSpace("Hello There", 5));
  Console.WriteLine("char.IsWhiteSpace('Hello There', 6): {0}",
    char.IsWhiteSpace("Hello There", 6));
  Console.WriteLine("char.IsPunctuation('?'): {0}",
    char.IsPunctuation('?'));
  Console.WriteLine();
}
```

Как было показано в этом методе, для многих членов System.Char предусмотрены два соглашения о вызове: одиночный символ или строка с числовым индексом, указывающим позицию проверяемого символа.

Разбор значений из строковых данных

Типы данных .NET предоставляют возможность генерировать переменную определенного типа, имея текстовый эквивалент (например, путем выполнения разбора). Этот прием может оказаться исключительно удобным, когда вы хотите преобразовывать в числовые значения некоторые вводимые пользователем данные (такие как элемент, выбранный в раскрывающемся списке внутри графического пользовательского интерфейса). Ниже приведен пример метода ParseFromStrings(), содержащий логику разбора:

```
static void ParseFromStrings()
{
  Console.WriteLine("=> Data type parsing:");
  bool b = bool.Parse("True");
  Console.WriteLine("Value of b: {0}", b);
  double d = double.Parse("99.884");
  Console.WriteLine("Value of d: {0}", d);
  int i = int.Parse("8");
  Console.WriteLine("Value of i: {0}", i);
  char c = Char.Parse("w");
  Console.WriteLine("Value of c: {0}", c);
  Console.WriteLine();
}
```

Типы System.DateTime и System.TimeSpan

В пространстве имен System определено несколько полезных типов данных, для которых отсутствуют ключевые слова языка C#, в том числе структуры DateTime и TimeSpan. (При желании можете самостоятельно ознакомиться с типами System.Guid и System.Void, показанными на рис. 3.2, но имейте в виду, что эти два типа данных редко оказываются практичными в большинстве приложений.)

Тип DateTime содержит данные, представляющие специфичное значение даты (месяц, день, год) и времени, которые могут форматироваться разнообразными способами с применением членов этого типа. Структура TimeSpan позволяет легко определять и трансформировать единицы времени, используя различные ее члены.

```
static void UseDatesAndTimes()
{
  Console.WriteLine("=> Dates and Times:");

  // Этот конструктор принимает год, месяц и день.
  DateTime dt = new DateTime(2015, 10, 17);

  // Какой это день месяца?
  Console.WriteLine("The day of {0} is {1}", dt.Date, dt.DayOfWeek);

  // Сейчас месяц декабрь.
  dt = dt.AddMonths(2);
  Console.WriteLine("Daylight savings: {0}", dt.IsDaylightSavingTime());

  // Этот конструктор принимает часы, минуты и секунды.
  TimeSpan ts = new TimeSpan(4, 30, 0);
  Console.WriteLine(ts);

  // Вычесть 15 минут из текущего значения TimeSpan и вывести результат.
  Console.WriteLine(ts.Subtract(new TimeSpan(0, 15, 0)));
}
```

Сборка System.Numerics.dll

В пространстве имен System.Numerics определена структура по имени BigInteger. Тип данных BigInteger может применяться для представления *огромных* числовых значений, которые не ограничены фиксированным верхним или нижним пределом.

На заметку! В пространстве имен System.Numerics также определена вторая структура по имени Complex, которая позволяет моделировать математически сложные числовые данные (например, мнимые единицы, вещественные данные, гиперболические тангенсы). Дополнительные сведения об этой структуре можно найти в документации .NET Framework 4.6 SDK.

Хотя во многих приложениях .NET потребность в использовании структуры BigInteger может никогда не возникать, но если все же необходимо определить большое числовое значение, то в первую очередь понадобится добавить в проект ссылку на сборку System.Numerics.dll. Выполните следующие действия.

1. Выберите в Visual Studio пункт меню Project⇨Add Reference (Проект⇨Добавить ссылку).
2. Найдите и выберите сборку System.Numerics.dll в списке представленных библиотек, который отображается на вкладке Framework (Платформа) слева.
3. Щелкните на кнопке OK.

После этого добавьте показанную ниже директиву using в файл, в котором будет применяться тип данных BigInteger:

```
// Здесь определен тип BigInteger:
using System.Numerics;
```

Теперь можно создать переменную BigInteger, используя операцию new. Внутри конструктора можно указать числовое значение, включая данные с плавающей точкой. Однако вспомните, что когда определяется целочисленный литерал (вроде 500), исполняющая среда по умолчанию трактует его как относящийся к типу int. Аналогично литерал с плавающей точкой (такой как 55.333) по умолчанию будет отнесен к типу double. Как же тогда установить для BigInteger большое значение, не переполнив стандартные типы данных, которые применяются для неформатированных числовых значений?

Простейший подход предусматривает определение большого числового значения в виде текстового литерала, который затем может быть преобразован в переменную BigInteger посредством статического метода Parse(). При желании можно также передавать байтовый массив непосредственно конструктору класса BigInteger.

На заметку! После присваивания значения переменной BigInteger модифицировать ее больше нельзя, т.к. это неизменяемые данные. Тем не менее, в классе BigInteger определено несколько членов, которые возвращают новые объекты BigInteger на основе модификаций данных (такие как статический метод Multiply(), используемый в следующем примере кода).

В любом случае после определения переменной BigInteger вы обнаружите, что в этом классе определены члены, похожие на члены в других внутренних типах данных C# (например, float, int). Вдобавок в классе BigInteger определен ряд статических членов, которые позволяют применять к переменным BigInteger базовые математические операции (наподобие сложения и умножения). Ниже приведен пример работы с классом BigInteger:

```
static void UseBigInteger()
{
  Console.WriteLine("=> Use BigInteger:");
  BigInteger biggy =
    BigInteger.Parse("99999999999999999999999999999999999999999999999");
  Console.WriteLine("Value of biggy is {0}", biggy);
                    // значение biggy
  Console.WriteLine("Is biggy an even value?: {0}", biggy.IsEven);
                     // biggy - четное?
  Console.WriteLine("Is biggy a power of two?: {0}", biggy.IsPowerOfTwo);
                    // biggy - степень 2?
  BigInteger reallyBig = BigInteger.Multiply(biggy,
    BigInteger.Parse("8888888888888888888888888888888888888888888888"));
  Console.WriteLine("Value of reallyBig is {0}", reallyBig);
                    // значение reallyBig
}
```

Важно отметить, что тип данных `BigInteger` реагирует на внутренние математические операции C#, такие как `+`, `-` и `*`. Следовательно, вместо вызова метода `BigInteger.Multiply()` для перемножения двух больших чисел можно использовать такой код:

```
BigInteger reallyBig2 = biggy * reallyBig;
```

К этому моменту вы должны понимать, что ключевые слова C#, представляющие базовые типы данных, имеют соответствующие типы в библиотеках базовых классов .NET, каждый из которых предлагает фиксированную функциональность. Хотя абсолютно все члены этих типов данных в книге подробно не рассматриваются, не помешает изучить их самостоятельно. Подробные описания разнообразных типов данных .NET можно найти в документации .NET Framework 4.6 SDK — скорее всего, вы будете удивлены объемом их встроенной функциональности.

Исходный код. Проект `BasicDataTypes` доступен в подкаталоге `Chapter_3`.

Работа со строковыми данными

Класс `System.String` предоставляет множество методов, наличие которых в служебном классе подобного рода вполне ожидаемо, в том числе методы для определения длины символьных данных, поиска подстрок в текущей строке и преобразования символов между верхним и нижним регистрами. В табл. 3.5 перечислены некоторые интересные члены этого класса.

Таблица 3.5. Избранные члены класса `System.String`

Член `System.String`	Описание
`Length`	Свойство, которое возвращает длину текущей строки
`Compare()`	Статический метод, который позволяет сравнить две строки
`Contains()`	Метод, который позволяет определить, содержится ли в строке указанная подстрока
`Equals()`	Метод, который позволяет проверить, содержатся ли в двух строковых объектах идентичные символьные данные

Окончание табл. 3.5

Член System.String	Описание
Format()	Статический метод, позволяющий сформатировать строку с использованием других элементарных типов данных (например, числовых данных или других строк) и системы обозначений {0}, которая рассматривалась ранее в этой главе
Insert()	Метод, который позволяет вставить строку внутрь заданной строки
PadLeft() PadRight()	Методы, которые позволяют дополнить строку определенными символами
Remove() Replace()	Методы, которые позволяют получить копию строки с произведенными изменениями (удалением или заменой символов)
Split()	Метод, возвращающий массив string, который содержит подстроки в этом экземпляре, разделенные элементами из указанного массива char или string
Trim()	Метод, который удаляет все вхождения набора указанных символов с начала и конца текущей строки
ToUpper() ToLower()	Методы, которые создают копию текущей строки в верхнем или нижнем регистре

Базовые манипуляции строками

Работа с членами System.String выглядит так, как и можно было ожидать. Нужно просто объявить переменную string и задействовать предлагаемую типом функциональность через операцию точки. Не следует забывать, что несколько членов System.String являются статическими и потому должны вызываться на уровне класса (а не объекта). В целях иллюстрации создадим новый проект консольного приложения по имени FunWithStrings и добавим в него показанный далее метод, который будет вызываться внутри Main():

```
static void BasicStringFunctionality()
{
 Console.WriteLine("=> Basic String functionality:");
 string firstName = "Freddy";
 Console.WriteLine("Value of firstName: {0}", firstName);
                  // Значение firstName.
 Console.WriteLine("firstName has {0} characters.", firstName.Length);
                  // Длина firstname.
 Console.WriteLine("firstName in uppercase: {0}", firstName.ToUpper());
                  // firstName в верхнем регистре.
 Console.WriteLine("firstName in lowercase: {0}", firstName.ToLower());
                  // firstName в нижнем регистре.
 Console.WriteLine("firstName contains the letter y?: {0}", firstName.Contains("y"));
                // Содержит ли firstName букву y?
 Console.WriteLine("firstName after replace: {0}", firstName.Replace("dy", ""));
                 // firstName после замены.
 Console.WriteLine();
}
```

Здесь объяснять особо нечего: метод просто вызывает разнообразные члены, такие как ToUpper() и Contains(), на локальной переменной string, чтобы получить разные форматы и трансформации. Ниже приведен вывод:

```
***** Fun with Strings *****
```

```
=> Basic String functionality:
Value of firstName: Freddy
firstName has 6 characters.
firstName in uppercase: FREDDY
firstName in lowercase: freddy
firstName contains the letter y?: True
firstName after replace: Fred
```

Несмотря на то что вывод не выглядит особенно неожиданным, вывод, полученный в результате вызова метода `Replace()`, может ввести в заблуждение. В действительности переменная `firstName` вообще не изменяется; вместо этого мы получаем новую переменную `string` в модифицированном формате. Чуть позже мы еще вернемся к обсуждению неизменяемой природы строк.

Конкатенация строк

Переменные `string` могут соединяться вместе для построения строк большего размера с помощью операции + языка C#. Как вам должно быть известно, этот прием формально называется *конкатенацией строк*. Рассмотрим следующую вспомогательную функцию:

```
static void StringConcatenation()
{
  Console.WriteLine("=> String concatenation:");
  string s1 = "Programming the ";
  string s2 = "PsychoDrill (PTP)";
  string s3 = s1 + s2;
  Console.WriteLine(s3);
  Console.WriteLine();
}
```

Возможно, будет полезно узнать, что при обработке символа + компилятор C# выпускает вызов статического метода `String.Concat()`. В результате конкатенацию строк можно также выполнять, вызывая метод `String.Concat()` напрямую (хотя, по правде говоря, это не дает никаких преимуществ, а лишь увеличивает объем набираемого кода):

```
static void StringConcatenation()
{
  Console.WriteLine("=> String concatenation:");
  string s1 = "Programming the ";
  string s2 = "PsychoDrill (PTP)";
  string s3 = String.Concat(s1, s2);
  Console.WriteLine(s3);
  Console.WriteLine();
}
```

Управляющие последовательности

Как и в других языках, основанных на C, строковые литералы C# могут содержать разнообразные *управляющие последовательности*, которые позволяют уточнять то, как символьные данные должны быть представлены в потоке вывода. Каждая управляющая последовательность начинается с символа обратной косой черты, за которым следует специфический знак. В табл. 3.6 перечислены наиболее распространенные управляющие последовательности.

Например, чтобы вывести строку, которая содержит символ табуляции после каждого слова, можно задействовать управляющую последовательность `\t`. Или предположим, что нужно создать один строковый литерал с символами кавычек внутри, второй — с определением пути к каталогу и третий — со вставкой трех пустых строк после вывода символьных данных.

Таблица 3.6. Управляющие последовательности в строковых литералах

Управляющая последовательность	Описание
\'	Вставляет в строковый литерал символ одинарной кавычки
\"	Вставляет в строковый литерал символ двойной кавычки
\\	Вставляет в строковый литерал символ обратной косой черты. Это особенно полезно при определении путей к файлам или сетевым ресурсам
\a	Заставляет систему выдать звуковой сигнал, который в консольных приложениях может служить аудио-подсказкой пользователю
\n	Вставляет символ новой строки (на платформах Windows)
\r	Вставляет символ возврата каретки
\t	Вставляет в строковый литерал символ горизонтальной табуляции

Для этого можно применять управляющие последовательности \", \\ и \n. Кроме того, ниже приведен еще один пример, в котором для привлечения внимания каждый строковый литерал сопровождается звуковым сигналом:

```
static void EscapeChars()
{
  Console.WriteLine("=> Escape characters:\a");
  string strWithTabs = "Model\tColor\tSpeed\tPet Name\a ";
  Console.WriteLine(strWithTabs);

  Console.WriteLine("Everyone loves \"Hello World\"\a ");
  Console.WriteLine("C:\\MyApp\\bin\\Debug\a ");

  // Добавить четыре пустых строки и снова выдать звуковой сигнал.
  Console.WriteLine("All finished.\n\n\n\a ");
  Console.WriteLine();
}
```

Определение дословных строк

За счет добавления к строковому литералу префикса @ можно создавать так называемые *дословные строки*. Используя дословные строки, вы отключаете обработку управляющих последовательностей в литералах и заставляете выводить значения string в том виде, как есть. Такая возможность наиболее полезна при работе со строками, представляющими пути к каталогам и сетевым ресурсам. Таким образом, вместо применения управляющей последовательности \\ можно поступить следующим образом:

```
// Следующая строка воспроизводится дословно, так что
// все управляющие последовательности отображаются.
Console.WriteLine(@"C:\MyApp\bin\Debug");
```

Также обратите внимание, что дословные строки могут использоваться для предохранения пробельных символов в строках, разнесенных по нескольким строкам вывода:

```
// При использовании дословных строк пробельные символы предохраняются.
string myLongString = @"This is a very
    very
        very
            long string";
Console.WriteLine(myLongString);
```

Применяя дословные строки, в литеральную строку можно также напрямую вставлять символы двойной кавычки, просто дублируя знак ":

```
Console.WriteLine(@"Cerebus said ""Darrr! Pret-ty sun-sets""");
```

Строки и равенство

Как будет подробно объясняться в главе 4, *ссылочный тип* — это объект, размещаемый в управляемой куче со сборкой мусора. По умолчанию при выполнении проверки на предмет равенства ссылочных типов (с помощью операций == и != языка C#) значение `true` будет возвращаться в случае, если обе ссылки указывают на один и тот же объект в памяти. Однако, несмотря на то, что `string` в действительности является ссылочным типом, операции равенства для него были переопределены так, чтобы можно было сравнивать *значения* объектов `string`, а не ссылки на объекты в памяти.

```
static void StringEquality()
{
  Console.WriteLine("=> String equality:");
  string s1 = "Hello!";
  string s2 = "Yo!";
  Console.WriteLine("s1 = {0}", s1);
  Console.WriteLine("s2 = {0}", s2);
  Console.WriteLine();

  // Проверить строки на равенство.
  Console.WriteLine("s1 == s2: {0}", s1 == s2);
  Console.WriteLine("s1 == Hello!: {0}", s1 == "Hello!");
  Console.WriteLine("s1 == HELLO!: {0}", s1 == "HELLO!");
  Console.WriteLine("s1 == hello!: {0}", s1 == "hello!");
  Console.WriteLine("s1.Equals(s2): {0}", s1.Equals(s2));
  Console.WriteLine("Yo.Equals(s2): {0}", "Yo!".Equals(s2));
  Console.WriteLine();
}
```

Операции равенства C# выполняют в отношении объектов `string` посимвольную проверку равенства с учетом регистра. Следовательно, строка `"Hello!"` не равна строке `"HELLO!"`, которая также отличается от строки `"hello!"`. Кроме того, памятуя о связи между `string` и `System.String`, проверку на предмет равенства можно осуществлять с использованием метода `Equals()` класса `String` и других поддерживаемых им операций равенства. И, наконец, поскольку каждый строковый литерал (такой как "Yo") является допустимым экземпляром `System.String`, доступ к функциональности, ориентированной на работу со строками, можно получать также для фиксированной последовательности символов.

Строки являются неизменяемыми

Один из интересных аспектов класса `System.String` связан с тем, что после присваивания объекту `string` начального значения символьные данные *не могут быть изменены*. На первый взгляд это может показаться противоречащим действительности, ведь строкам постоянно присваиваются новые значения, а в классе `System.String` доступен набор методов, которые, похоже, только то и делают, что изменяют символьные данные тем или иным образом (к примеру, преобразуя их в верхний или нижний регистр). Тем не менее, присмотревшись внимательнее к тому, что происходит "за кулисами", вы заметите, что методы типа `string` на самом деле возвращают новый объект `string` в модифицированном виде:

```
static void StringsAreImmutable()
{
  // Установить начальное значение для строки.
  string s1 = "This is my string.";
  Console.WriteLine("s1 = {0}", s1);

  // Преобразована ли строка s1 в верхний регистр?
  string upperString = s1.ToUpper();
  Console.WriteLine("upperString = {0}", upperString);

  // Нет! Строка s1 осталась в том же виде!
  Console.WriteLine("s1 = {0}", s1);
}
```

Взглянув на показанный далее вывод, можно убедиться, что исходный объект `string` (`s1`) не был преобразован в верхний регистр, когда вызывался метод `ToUpper()`. Взамен была возвращена *копия* переменной типа `string` в измененном формате.

```
s1 = This is my string.
upperString = THIS IS MY STRING.
s1 = This is my string.
```

Тот же самый закон неизменяемости строк действует и в случае применения операции присваивания C#. Чтобы проиллюстрировать, реализуем следующий метод `StringsAreImmutable2()`:

```
static void StringsAreImmutable2()
{
  string s2 = "My other string";
  s2 = "New string value";
}
```

Скомпилируем приложение и загрузим сборку в `ildasm.exe` (см. главу 1). Ниже приведен код CIL, который будет сгенерирован для метода `StringsAreImmutable2()`:

```
.method private hidebysig static void StringsAreImmutable2() cil managed
{
  // Code size 14 (0xe)
  .maxstack 1
  .locals init ([0] string s2)
  IL_0000:  nop
  IL_0001:  ldstr      "My other string"
  IL_0006:  stloc.0
  IL_0007:  ldstr      "New string value"
  IL_000c:  stloc.0
  IL_000d:  ret
} // end of method Program::StringAreImmutable2
```

Хотя низкоуровневые детали языка CIL пока подробно не рассматривались, обратите внимание на многочисленные вызовы кода операции `ldstr` ("load string" — "загрузить строку"). Попросту говоря, код операции `ldstr` языка CIL загружает новый объект `string` в управляемую кучу. Предыдущий объект `string`, который содержал значение `"My other string"`, будет со временем удален сборщиком мусора.

Так что же в точности из всего этого следует? В двух словах, класс `string` может стать неэффективным и при неправильном употреблении приводить к "разбуханию" кода, особенно при выполнении конкатенации строк или работе с большими объемами текстовых данных. Но если необходимо представлять элементарные символьные данные, такие как номер карточки социального страхования, имя и фамилия или простые фрагменты текста, используемые внутри приложения, то тип `string` будет идеальным вариантом.

Однако когда строится приложение, в котором текстовые данные будут часто изменяться (подобное текстовому процессору), то представление обрабатываемых текстовых данных с применением объектов `string` будет неудачным решением, т.к. это практически наверняка (и часто косвенно) приведет к созданию излишних копий строковых данных. Тогда каким образом должен поступить программист? Ответ на этот вопрос вы найдете ниже.

Тип `System.Text.StringBuilder`

С учетом того, что тип `string` может оказаться неэффективным при безответственном использовании, библиотеки базовых классов .NET предоставляют пространство имен `System.Text`. Внутри этого (относительно небольшого) пространства имен находится класс `StringBuilder`. Как и `System.String`, класс `StringBuilder` определяет методы, которые позволяют, к примеру, заменять или форматировать сегменты. Для применения этого класса в файлах кода C# первым делом понадобится импортировать следующее пространство имен в файл кода (в случае нового проекта Visual Studio это уже должно быть сделано):

```
// Здесь определен класс StringBuilder:
using System.Text;
```

Уникальность класса `StringBuilder` в том, что при вызове его членов производится прямое изменение внутренних символьных данных объекта (делая его более эффективным) без получения копии данных в модифицированном формате. При создании экземпляра `StringBuilder` начальные значения объекта могут быть заданы через один из множества *конструкторов*. Если вы не знакомы с понятием конструктора, то пока достаточно знать лишь то, что конструкторы позволяют создавать объект с начальным состоянием, когда используется ключевое слово `new`. Взгляните на следующий пример применения `StringBuilder`:

```
static void FunWithStringBuilder()
{
  Console.WriteLine("=> Using the StringBuilder:");
  StringBuilder sb = new StringBuilder("**** Fantastic Games ****");
  sb.Append("\n");
  sb.AppendLine("Half Life");
  sb.AppendLine("Morrowind");
  sb.AppendLine("Deus Ex" + "2");
  sb.AppendLine("System Shock");
  Console.WriteLine(sb.ToString());
  sb.Replace("2", " Invisible War");
  Console.WriteLine(sb.ToString());
  Console.WriteLine("sb has {0} chars.", sb.Length);
  Console.WriteLine();
}
```

Здесь создается объект `StringBuilder` с начальным значением `"**** Fantastic Games ****"`. Как видите, можно добавлять строки в конец внутреннего буфера, а также заменять или удалять любые символы. По умолчанию `StringBuilder` способен хранить строку только длиной 16 символов или меньше (но при необходимости будет автоматически расширяться); однако значение первоначальной длины можно изменить посредством дополнительного аргумента конструктора:

```
// Создать экземпляр StringBuilder с исходным размером в 256 символов.
StringBuilder sb = new StringBuilder("**** Fantastic Games ****", 256);
```

При добавлении большего количества символов, чем в указанном лимите, объект `StringBuilder` скопирует свои данные в новый экземпляр и увеличит размер буфера на заданный лимит.

Интерполяция строк

Синтаксис с фигурными скобками, продемонстрированный в этой главе (`{0}`, `{1}` и т.д.), существовал внутри платформы .NET еще со времен версии 1.0. Начиная с текущего выпуска, при построении строковых литералов, содержащих заполнители для переменных, программисты C# могут использовать альтернативный синтаксис. Формально он называется *интерполяцией строк*. Несмотря на то что вывод операции идентичен выводу, получаемому с помощью традиционного синтаксиса форматирования строк, новый подход позволяет напрямую внедрять сами переменные, а не помещать их в список с разделителями-запятыми.

Взгляните на показанный ниже дополнительный метод в нашем классе `Program` (`StringInterpolation()`), который строит переменную типа `string` с применением обоих подходов:

```
static void StringInterpolation()
{
  // Некоторые локальные переменные будут включены в крупную строку.
  int age = 4;
  string name = "Soren";

  // Использование синтаксиса с фигурными скобками.
  string greeting = string.Format("Hello {0} you are {1} years old.", name, age);

  // Использование интерполяции строк.
  string greeting2 = $"Hello {name} you are {age} years old.";
}
```

В переменной `greeting2` обратите внимание на то, что конструируемая строка начинается с префикса `$`. Кроме того, фигурные скобки по-прежнему используются для пометки заполнителя под переменную; тем не менее, вместо применения числовой метки имеется возможность указывать непосредственно переменную. Предполагаемое преимущество заключается в том, что новый синтаксис несколько легче читать в линейной манере (слева направо) с учетом того, что не требуется "перескакивать в конец" для просмотра списка значений, подлежащих вставке во время выполнения.

С новым синтаксисом связан еще один интересный аспект: фигурные скобки, используемые в интерполяции строк, обозначают допустимую область видимости. Таким образом, с переменными можно применять операцию точки, чтобы изменять их состояние. Модифицируем код присваивания переменных `greeting` и `greeting2`:

```
string greeting = string.Format("Hello {0} you are {1} years old.",
                                name.ToUpper(), age);
string greeting2 = $"Hello {name.ToUpper()} you are {age} years old.";
```

Здесь посредством вызова `ToUpper()` производится преобразование `name` в верхний регистр. Обратите внимание, что при подходе интерполяции строк завершающая пара круглых скобок к вызову этого метода *не* добавляется. Учитывая это, использовать область видимости, определяемую фигурными скобками, как полноценную область видимости метода, которая содержит многочисленные строки исполняемого кода, невозможно. Взамен допускается только вызывать одиночный метод на объекте с применением операции точки, а также определять простое общее выражение наподобие `{age += 1}`.

Полезно также отметить, что в рамках нового синтаксиса можно по-прежнему использовать управляющие последовательности внутри строкового литерала.

Таким образом, для вставки символа табуляции необходимо применять последовательность \t:

```
string greeting = string.Format("\tHello {0} you are {1} years old.",
                                name.ToUpper(), age);
string greeting2 = $"\tHello {name.ToUpper()} you are {age} years old.";
```

Как и следовало ожидать, при построении переменных типа `string` на лету вы вправе использовать любой из двух подходов. Однако имейте в виду, что в случае работы с более ранней версией платформы .NET применение синтаксиса интерполяции строк приведет к ошибке на этапе компиляции. Следовательно, если вам необходимо обеспечить успешную компиляцию кода C# с помощью множества версий компилятора, то безопаснее придерживаться традиционного подхода с нумерованными заполнителями.

Исходный код. Проект `FunWithStrings` доступен в подкаталоге `Chapter_3`.

Сужающие и расширяющие преобразования типов данных

Теперь, когда вы понимаете, как работать с внутренними типами данных C#, давайте рассмотрим связанную тему *преобразования типов данных*. Создадим новый проект консольного приложения по имени `TypeConversions` и определим в нем следующий класс:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with type conversions *****");
    // Добавить две переменные типа short и вывести результат.
    short numb1 = 9, numb2 = 10;
    Console.WriteLine("{0} + {1} = {2}",
      numb1, numb2, Add(numb1, numb2));
    Console.ReadLine();
  }
  static int Add(int x, int y)
  {
    return x + y;
  }
}
```

Как легко заметить, метод `Add()` ожидает передачи двух параметров `int`. Тем не менее, в методе `Main()` ему на самом деле передаются две переменные типа `short`. Хотя это может выглядеть похожим на несоответствие типов данных, программа компилируется и выполняется без ошибок, возвращая ожидаемый результат 19.

Причина, по которой компилятор считает этот код синтаксически корректным, связана с тем, что потеря данных здесь невозможна. Из-за того, что максимальное значение для типа `short` (32 767) гораздо меньше максимального значения для типа `int` (2 147 483 647), компилятор неявно *расширяет* каждое значение `short` до типа `int`. Формально термин *расширение* используется для определения неявного *восходящего приведения*, которое не вызывает потери данных.

На заметку! Разрешенные расширяющие и сужающие (обсуждаются далее) преобразования, поддерживаемые для каждого типа данных C#, описаны в разделе "Type Conversion Tables" ("Таблицы преобразования типов") документации .NET Framework 4.6 SDK.

Такое неявное расширение типов благоприятствовало в предыдущем примере, но в других ситуациях оно может стать источником ошибок на этапе компиляции. Например, пусть для переменных `numb1` и `numb2` установлены значения, которые (при их сложении) превышают максимальное значение типа `short`. Кроме того, предположим, что возвращаемое значение метода `Add()` сохраняется в новой локальной переменной `short`, а не напрямую выводится на консоль.

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with type conversions *****");
  // Следующий код вызовет ошибку на этапе компиляции!
  short numb1 = 30000, numb2 = 30000;
  short answer = Add(numb1, numb2);
  Console.WriteLine("{0} + {1} = {2}",
    numb1, numb2, answer);
  Console.ReadLine();
}
```

В данном случае компилятор сообщит о следующей ошибке:

```
Cannot implicitly convert type 'int' to 'short'. An explicit conversion
exists (are you missing a cast?)
Не удается неявно преобразовать тип int в short. Существует явное
преобразование (возможно, пропущено приведение)
```

Проблема в том, что хотя метод `Add()` способен возвратить значение `int`, равное 60 000 (т.к. оно умещается в допустимый диапазон для `System.Int32`), это значение не может быть сохранено в переменной `short`, потому что выходит за пределы диапазона допустимых значений для типа `short`. Выражаясь формально, среде CLR не удалось применить *сужающую операцию*. Нетрудно догадаться, что сужающая операция является логической противоположностью расширяющей операции, поскольку предусматривает сохранение большего значения внутри переменной типа данных с меньшим диапазоном допустимых значений.

Важно отметить, что все сужающие преобразования приводят к ошибкам на этапе компиляции, даже когда есть причина полагать, что такое преобразование должно пройти успешно. Например, следующий код также вызовет ошибку при компиляции:

```
// Снова ошибка на этапе компиляции!
static void NarrowingAttempt()
{
  byte myByte = 0;
  int myInt = 200;
  myByte = myInt;
  Console.WriteLine("Value of myByte: {0}", myByte);
}
```

Здесь значение, содержащееся в переменной типа `int` (`myInt`), благополучно умещается в диапазон допустимых значений для типа `byte`; следовательно, можно было бы ожидать, что сужающая операция не должна привести к ошибке во время выполнения. Однако из-за того, что язык C# создавался с расчетом на безопасность в отношении типов, все-таки будет получена ошибка на этапе компиляции.

Если нужно проинформировать компилятор о том, что вы готовы мириться с возможной потерей данных из-за сужающей операции, то потребуется применить *явное приведение*, используя операцию приведения () языка C#. Взгляните на показанную ниже модификацию класса Program:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with type conversions *****");
    short numb1 = 30000, numb2 = 30000;

    // Явно привести int к short (и разрешить потерю данных).
    short answer = (short)Add(numb1, numb2);

    Console.WriteLine("{0} + {1} = {2}",
      numb1, numb2, answer);
    NarrowingAttempt();
    Console.ReadLine();
}

  static int Add(int x, int y)
{
    return x + y;
}

  static void NarrowingAttempt()
{
    byte myByte = 0;
    int myInt = 200;

    // Явно привести int к byte (без потери данных).
    myByte = (byte)myInt;
    Console.WriteLine("Value of myByte: {0}", myByte);
  }
}
```

На этот раз компиляция кода проходит успешно, но результат сложения оказывается совершенно неправильным:

```
***** Fun with type conversions *****
30000 + 30000 = -5536
Value of myByte: 200
```

Как вы только что удостоверились, явное приведение заставляет компилятор применить сужающее преобразование, даже когда оно может вызвать потерю данных. В случае метода NarrowingAttempt() это не было проблемой, т.к. значение 200 умещалось в диапазон допустимых значений для типа byte. Тем не менее, в ситуации со сложением двух значений типа short внутри Main() конечный результат получился полностью неприемлемым (30 000 + 30 000 = –5536?).

Для построения приложений, в которых потеря данных не допускается, язык C# предлагает ключевые слова checked и unchecked, которые позволяют гарантировать, что потеря данных не окажется необнаруженной.

Ключевое слово checked

Давайте начнем с освоения роли ключевого слова checked. Предположим, что в класс Program добавлен новый метод, который пытается просуммировать две переменных типа byte, причем каждой из них было присвоено значение, не превышающее допустимый максимум (255).

По идее, после сложения значений этих двух переменных (с приведением результата int к типу byte) должна быть получена точная сумма.

```
static void ProcessBytes()
{
  byte b1 = 100;
  byte b2 = 250;
  byte sum = (byte)Add(b1, b2);

  // В sum должно содержаться значение 350. Однако там оказывается значение 94!
  Console.WriteLine("sum = {0}", sum);
}
```

Удивительно, но при просмотре вывода этого приложения обнаруживается, что в переменной sum содержится значение 94 (а не 350, как ожидалось). Причина проста. Учитывая, что System.Byte может хранить только значение из диапазона от 0 до 255 включительно, в sum будет помещено значение переполнения (350 – 256 = 94). По умолчанию, если не предпринимаются никакие корректирующие действия, то условия переполнения и потери значимости происходят без выдачи сообщений об ошибках.

Для обработки условий переполнения и потери значимости в приложении доступны два способа. Это можно делать вручную, полагаясь на свои знания и навыки в области программирования. Недостаток такого подхода произрастает из того факта, что мы всего лишь люди, и даже приложив максимум усилий, все равно можем попросту упустить из виду какие-то ошибки.

К счастью, язык C# предоставляет ключевое слово checked. Когда оператор (или блок операторов) помещен в контекст checked, компилятор C# выпускает дополнительные инструкции CIL, обеспечивающие проверку условий переполнения, которые могут возникать при сложении, умножении, вычитании или делении двух значений числовых типов.

Если происходит переполнение, то во время выполнения генерируется исключение System.OverflowException. В главе 7 будут предложены подробные сведения о структурированной обработке исключений, а также об использовании ключевых слов try и catch. Не вдаваясь пока в детали, взгляните на следующий модифицированный код:

```
static void ProcessBytes()
{
  byte b1 = 100;
  byte b2 = 250;

  // На этот раз сообщить компилятору о необходимости добавления
  // кода CIL, необходимого для генерации исключения, если возникает
  // переполнение или потеря значимости.
  try
  {
    byte sum = checked((byte)Add(b1, b2));
    Console.WriteLine("sum = {0}", sum);
  }
  catch (OverflowException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Обратите внимание, что возвращаемое значение метода Add() помещено в контекст ключевого слова checked. Поскольку значение sum выходит за пределы допустимого диапазона для типа byte, генерируется исключение времени выполнения.

Сообщение об ошибке выводится посредством свойства `Message`:

```
Arithmetic operation resulted in an overflow.
Арифметическая операция привела к переполнению.
```

Чтобы обеспечить принудительную проверку переполнения для целого блока операторов, контекст `checked` можно определить так:

```
try
{
  checked
  {
    byte sum = (byte)Add(b1, b2);
    Console.WriteLine("sum = {0}", sum);
  }
}
catch (OverflowException ex)
{
  Console.WriteLine(ex.Message);
}
```

В любом случае интересующий код будет автоматически оцениваться на предмет возможных условий переполнения, и если они обнаружатся, то сгенерируется исключение, связанное с переполнением.

Настройка проверки переполнения на уровне проекта

Если создается приложение, в котором никогда не должно возникать молчаливое переполнение, то может обнаружиться, что в контекст ключевого слова `checked` приходится помещать слишком много строк кода. В качестве альтернативы компилятор C# поддерживает флаг `/checked`. Когда этот флаг указан, все присутствующие в коде арифметические операции будут оцениваться на предмет переполнения, не требуя применения ключевого слова `checked`. Если переполнение было обнаружено, тогда сгенерируется исключение времени выполнения.

Для активизации этого флага в Visual Studio откройте окно свойств проекта, перейдите на вкладку Build (Сборка) и щелкните на кнопке Advanced (Дополнительно). В открывшемся диалоговом окне отметьте флажок Check for arithmetic overflow/underflow (Проверять арифметическое переполнение и потерю значимости), как показано на рис. 3.3.

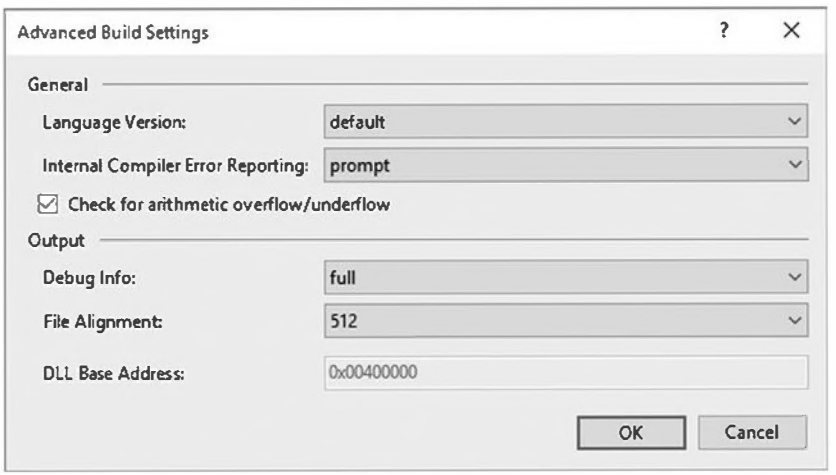


Рис. 3.3. Включение проверки переполнения и потери значимости в масштабах проекта

Включить эту настройку может быть удобно при создании отладочной версии сборки. После устранения всех условий переполнения из кодовой базы флаг /checked можно отключить для последующих построений сборки (что приведет к увеличению производительности приложения).

Ключевое слово unchecked

А теперь предположим, что проверка переполнения и потери значимости включена в масштабах проекта, но есть блок кода, в котором потеря данных *приемлема*. Как с ним быть? Учитывая, что действие флага /checked распространяется на всю арифметическую логику, в языке C# имеется ключевое слово unchecked, которое предназначено для отмены генерации исключений, связанных с переполнением, в отдельных случаях. Это ключевое слово используется аналогично ключевому слову checked, т.е. его можно применять как к единственному оператору, так и к блоку операторов.

```
// Предполагая, что флаг /checked активизирован, этот блок
// не будет генерировать исключение времени выполнения.
unchecked
{
  byte sum = (byte)(b1 + b2);
  Console.WriteLine("sum = {0} ", sum);
}
```

Итак, подводя итоги по ключевым словам checked и unchecked в C#, отметим, что стандартное поведение исполняющей среды .NET предусматривает игнорирование арифметического переполнения и потери значимости. Когда необходимо обрабатывать избранные операторы, должно использоваться ключевое слово checked. Если нужно перехватывать ошибки переполнения по всему приложению, то придется активизировать флаг /checked. Наконец, ключевое слово unchecked может применяться при наличии блока кода, в котором переполнение приемлемо (и, следовательно, не должно приводить к генерации исключения времени выполнения).

Исходный код. Проект TypeConversions доступен в подкаталоге Chapter_3.

Понятие неявно типизированных локальных переменных

Вплоть до этого места в главе при объявлении каждой локальной переменной *явно* указывался ее тип данных:

```
static void DeclareExplicitVars()
{
  // Явно типизированные локальные переменные
  // объявляются следующим образом:
  // типДанных имяПеременной = начальноеЗначение;
  int myInt = 0;
  bool myBool = true;
  string myString = "Time, marches on...";
}
```

Хотя многие (включая и авторов) согласятся с тем, что явное указание типа данных для каждой переменной является рекомендуемой практикой, язык C# поддерживает возможность *неявной типизации* локальных переменных с использованием ключевого слова var. Ключевое слово var может применяться вместо указания конкретного типа

данных (такого как int, bool или string). Когда вы поступаете подобным образом, компилятор будет автоматически выводить лежащий в основе тип данных на основе начального значения, используемого для инициализации локального элемента данных.

Чтобы продемонстрировать роль неявной типизации, создадим новый проект консольного приложения по имени ImplicitlyTypedLocalVars. Обратите внимание, что локальные переменные, которые присутствовали в показанной выше версии метода, теперь объявлены следующим образом:

```
static void DeclareImplicitVars()
{
  // Неявно типизированные локальные переменные
  // объявляются следующим образом:
  // var имяПеременной = начальноеЗначение;
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";
}
```

На заметку! Строго говоря, var не является ключевым словом языка C#. Вполне допустимо объявлять переменные, параметры и поля по имени var, не получая ошибок на этапе компиляции. Однако когда лексема var применяется в качестве типа данных, то в таком контексте она трактуется компилятором как ключевое слово.

В данном случае, основываясь на первоначально присвоенных значениях, компилятор способен вывести для переменной myInt тип System.Int32, для переменной myBool — тип System.Boolean, а для переменной myString — тип System.String. В этом легко убедиться за счет вывода на консоль имен типов с помощью *рефлексии*. Как будет показано в главе 15, рефлексия представляет собой действие по определению состава типа во время выполнения. Например, с помощью рефлексии можно определить тип данных неявно типизированной локальной переменной. Модифицируем метод DeclareImplicitVars(), как показано далее:

```
static void DeclareImplicitVars()
{
  // Неявно типизированные локальные переменные.
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";
  // Вывести имена лежащих в основе типов.
  Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
  Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
  Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

На заметку! Имейте в виду, что такую неявную типизацию можно использовать для любых типов, включая массивы, обобщенные типы (глава 9) и собственные специальные типы. В дальнейшем вы увидите и другие примеры неявной типизации.

Вызвав метод DeclareImplicitVars() внутри Main(), вы получите следующий вывод:

```
***** Fun with Implicit Typing *****

myInt is a: Int32
myBool is a: Boolean
myString is a: String
```

Ограничения неявно типизированных переменных

С использованием ключевого слова var связаны разнообразные ограничения. Прежде всего, неявная типизация применима *только* к локальным переменным внутри области видимости метода или свойства. Использовать ключевое слово var для определения возвращаемых значений, параметров или данных полей в специальном типе не допускается. Например, показанное ниже определение класса приведет к выдаче различных сообщений об ошибках на этапе компиляции:

```
class ThisWillNeverCompile
{
  // Ошибка! Ключевое слово var не может применяться к полям!
  private var myInt = 10;

  // Ошибка! Ключевое слово var не может применяться
  // к возвращаемому значению или типу параметра!
  public var MyMethod(var x, var y){}
}
```

Кроме того, локальным переменным, которые объявлены с ключевым словом var, *должно* быть присвоено начальное значение в самом объявлении, причем присваивать null в качестве начального значения невозможно. Последнее ограничение должно быть рациональным, потому что на основании только null компилятору не удастся вывести тип, на который бы указывала переменная.

```
// Ошибка! Должно быть присвоено значение!
var myData;

// Ошибка! Значение должно присваиваться в самом объявлении!
var myInt;
myInt = 0;

// Ошибка! Нельзя присваивать null в качестве начального значения!
var myObj = null;
```

Тем не менее, присваивать null локальной переменной, тип которой выведен в результате начального присваивания, разрешено (при условии, что это ссылочный тип):

```
// Допустимо, если SportsCar имеет ссылочный тип!
var myCar = new SportsCar();
myCar = null;
```

Вдобавок значение неявно типизированной локальной переменной допускается присваивать другим переменным, неявно или явно типизированным:

```
// Также допустимо!
var myInt = 0;
var anotherInt = myInt;

string myString = "Wake up!";
var myData = myString;
```

Наконец, неявно типизированную локальную переменную разрешено возвращать вызывающему компоненту при условии, что возвращаемый тип метода и выведенный тип переменной, определенной посредством var, совпадают:

```
static int GetAnInt()
{
  var retVal = 9;
  return retVal;
}
```

Неявно типизированные данные являются строго типизированными

Имейте в виду, что неявная типизация локальных переменных дает в результате *строго типизированные данные*. Таким образом, применение ключевого слова `var` в языке C# — это не тот же самый прием, который используется в языках написания сценариев (вроде JavaScript или Perl), и не тип данных `Variant` в COM, где переменная на протяжении своего времени жизни может хранить значения разных типов (что часто называют *динамической типизацией*).

На заметку! В C# поддерживается возможность динамической типизации с применением ключевого слова `dynamic`. Вы узнаете об этом аспекте языка в главе 16.

Вместо этого средство выведения типов сохраняет аспект строгой типизации языка C# и воздействует только на объявление переменных на этапе компиляции. Затем данные трактуются, как если бы они были объявлены с выведенным типом; присваивание такой переменной значения другого типа будет приводить к ошибке при компиляции.

```
static void ImplicitTypingIsStrongTyping()
{
  // Компилятор знает, что s имеет тип System.String.
  var s = "This variable can only hold string data!";
  s = "This is fine...";

  // Можно обращаться к любому члену лежащего в основе типа.
  string upper = s.ToUpper();

  // Ошибка! Присваивание числовых данных строке не допускается!
  s = 44;
}
```

Полезность неявно типизированных локальных переменных

Теперь, когда вы видели синтаксис, используемый для объявления неявно типизируемых локальных переменных, вас наверняка интересует, в каких ситуациях его следует применять. Прежде всего, использование `var` для объявления локальных переменных просто ради интереса особой пользы не принесет. Такой подход может вызвать путаницу у тех, кто будет изучать код, поскольку лишает возможности быстро определить лежащий в основе тип данных и, следовательно, затрудняет понимание общего назначения переменной. Поэтому если вы знаете, что переменная должна относиться к типу `int`, то сразу и объявляйте ее как `int`!

Однако, как будет показано в начале главы 12, в наборе технологий LINQ применяются *выражения запросов*, которые могут выдавать динамически создаваемые результирующие наборы, основанные на формате самого запроса. В таких случаях неявная типизация исключительно удобна, потому что вам не придется явно определять тип, который запрос может возвращать, а в ряде ситуаций это вообще невозможно. Посмотрите, сможете ли вы определить лежащий в основе тип данных `subset` в следующем примере кода LINQ?

```
static void LinqQueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
  // Запрос LINQ!
  var subset = from i in numbers where i < 10 select i;
  Console.Write("Values in subset: ");
```

```
  foreach (var i in subset)
  {
    Console.Write("{0} ", i);
  }
  Console.WriteLine();

  // И к какому же типу относится subset?
  Console.WriteLine("subset is a: {0}", subset.GetType().Name);
  Console.WriteLine("subset is defined in: {0}", subset.GetType().Namespace);
}
```

Вы могли бы предположить, что типом данных `subset` является массив целочисленных значений. Кажется, что это так, но на самом деле он представляет собой низкоуровневый тип данных LINQ, о котором вы вряд ли что-то знаете, если только не работаете с LINQ длительное время или не откроете скомпилированный образ в утилите `ildasm.exe`. Хорошая новость состоит в том, что при использовании LINQ вы редко (если вообще когда-либо) беспокоитесь о типе возвращаемого значения запроса; вы просто присваиваете значение неявно типизированной локальной переменной.

Фактически можно было бы даже утверждать, что *единственным случаем*, когда применение ключевого слова `var` полностью оправдано, является определение данных, возвращаемых из запроса LINQ. Запомните, если вы знаете, что нужна переменная `int`, то просто объявляйте ее как `int`! Злоупотребление неявной типизацией в производственном коде (через ключевое слово `var`) большинство разработчиков расценивают как плохой стиль кодирования.

Исходный код. Проект `ImplicitlyTypedLocalVars` доступен в подкаталоге `Chapter_3`.

Итерационные конструкции C#

Все языки программирования предлагают приемы для повторения блоков кода до тех пор, пока не будет удовлетворено условие завершения. С каким бы языком вы не имели дела в прошлом, итерационные операторы C# не должны вызывать особого удивления или требовать небольшого объяснения. В C# предоставляются четыре итерационных конструкции:

- цикл `for`;
- цикл `foreach/in`;
- цикл `while`;
- цикл `do/while`.

Давайте кратко рассмотрим каждую конструкцию зацикливания по очереди, создав новый проект консольного приложения по имени `IterationsAndDecisions`.

На заметку! Материал этого заключительного раздела главы будет кратким и по существу, т.к. здесь предполагается наличие у вас опыта работы с аналогичными ключевыми словами (`if`, `for`, `switch` и т.д.) в другом языке программирования. Если нужна дополнительная информация, просмотрите темы "Iteration Statements (C# Reference)" ("Операторы итераций (справочник по C#)"), "Jump Statements (C# Reference)" ("Операторы перехода (Справочник по C#)") и "Selection Statements (C# Reference)" ("Операторы выбора (Справочник по C#)") в документации .NET Framework 4.6 SDK.

Цикл for

Когда требуется повторять блок кода фиксированное количество раз, хороший уровень гибкости предлагает оператор for. В действительности вы имеете возможность указывать, сколько раз должен повторяться блок кода, а также задавать условие завершения. Не вдаваясь в излишние подробности, ниже представлен пример синтаксиса:

```
// Базовый цикл for.
static void ForLoopExample()
{
  // Обратите внимание, что переменная i видима только в контексте цикла
for.
  for(int i = 0; i < 4; i++)
  {
    Console.WriteLine("Number is: {0} ", i);
  }
  // Здесь переменная i больше видимой не будет.
}
```

Все трюки, которые вы научились делать в языках C, C++ и Java, по-прежнему могут использоваться при формировании операторов for в C#. Допускается создавать сложные условия завершения, строить бесконечные циклы и циклы в обратном направлении (посредством операции --), а также применять ключевые слова goto, continue и break.

Цикл foreach

Ключевое слово foreach в C# позволяет проходить в цикле по всем элементам внутри контейнера без необходимости в проверке верхнего предела. Тем не менее, в отличие от цикла for цикл foreach будет выполнять проход по контейнеру только линейным (n+1) образом (т.е. не удастся проходить по контейнеру в обратном направлении, пропускать каждый третий элемент и т.п.).

Однако если нужно просто выполнить проход по коллекции элемент за элементом, то цикл foreach будет великолепным выбором. Ниже приведены два примера использования цикла foreach — один для обхода массива строк и еще один для обхода массива целых чисел. Обратите внимание, что тип, указанный перед ключевым словом in, представляет тип данных контейнера.

```
// Проход по элементам массива посредством foreach.
static void ForEachLoopExample()
{
  string[] carTypes = {"Ford", "BMW", "Yugo", "Honda" };
  foreach (string c in carTypes)
    Console.WriteLine(c);

  int[] myInts = { 10, 20, 30, 40 };
  foreach (int i in myInts)
  Console.WriteLine(i);
}
```

За ключевым словом in может быть указан простой массив (как в приведенном примере) или, точнее говоря, любой класс, реализующий интерфейс IEnumerable. Как вы увидите в главе 9, библиотеки базовых классов .NET поставляются с несколькими коллекциями, которые содержат реализации распространенных абстрактных типов данных. Любой из них (скажем, обобщенный тип List<T>) может применяться внутри цикла foreach.

Использование неявной типизации в конструкциях foreach

В итерационных конструкциях foreach также допускается использование неявной типизации. Как и можно было ожидать, компилятор будет выводить корректный "тип типа". Вспомните пример метода LINQ, представленный ранее в главе. Даже не зная точного типа данных переменной subset, с применением неявной типизации все-таки можно выполнять итерацию по результирующему набору:

```
static void LinqQueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
  // Запрос LINQ!
  var subset = from i in numbers where i < 10 select i;
  Console.Write("Values in subset: ");
  foreach (var i in subset)
  {
    Console.Write("{0} ", i);
  }
}
```

Циклы while и do/while

Итерационная конструкция while удобна, когда блок операторов должен выполняться до тех пор, пока не будет удовлетворено некоторое условие завершения. Внутри области видимости цикла while необходимо позаботиться о том, чтобы это условие действительно удовлетворялось, иначе получится бесконечный цикл. В следующем примере сообщение "In while loop" будет постоянно выводиться на консоль, пока пользователь не завершит цикл вводом yes в командной строке:

```
static void WhileLoopExample()
{
  string userIsDone = "";
  // Проверить копию строки в нижнем регистре.
  while(userIsDone.ToLower() != "yes")
  {
    Console.WriteLine("In while loop");
    Console.Write("Are you done? [yes] [no]: ");  // Запрос продолжения
    userIsDone = Console.ReadLine();
  }
}
```

С циклом while тесно связан оператор do/while. Подобно простому циклу while цикл do/while используется, когда какое-то действие должно выполняться неопределенное количество раз. Разница в том, что цикл do/while гарантирует, по крайней мере, однократное выполнение соответствующего блока кода. С другой стороны, вполне возможно, что цикл while вообще его не выполнит, если условие оказывается ложным с самого начала.

```
static void DoWhileLoopExample()
{
  string userIsDone = "";
  do
  {
    Console.WriteLine("In do/while loop");
    Console.Write("Are you done? [yes] [no]: ");
    userIsDone = Console.ReadLine();
  }while(userIsDone.ToLower() != "yes"); //Обратите внимание на точку с запятой!
}
```

Конструкции принятия решений и операции отношения/равенства

Теперь, когда вы умеете многократно выполнять блок операторов, давайте рассмотрим следующую связанную концепцию — управление потоком выполнения программы. Для изменения потока выполнения программы на основе разнообразных обстоятельств в C# определены две простые конструкции:

- оператор `if/else`;
- оператор `switch`.

Оператор `if/else`

Первым мы рассмотрим оператор `if/else`. В отличие от C и C++, в языке C# этот оператор может работать только с булевскими выражениями, но не с произвольными значениями наподобие `-1` и `0`.

Операции отношения и равенства

Обычно для получения литерального булевского значения в операторах `if/else` применяются операции, описанные в табл. 3.7.

Таблица 3.7. Операции отношения и равенства в C#

Операция отношения/равенства	Пример использования	Описание
`==`	`if(age == 30)`	Возвращает `true`, если выражения являются одинаковыми
`!=`	`if("Foo" != myStr)`	Возвращает `true`, если выражения являются разными

`<`
`>`
`<=`
`>=`
	`if(bonus < 2000)`
	`if(bonus > 2000)`
	`if(bonus <= 2000)`
	`if(bonus >= 2000)`
		Возвращает `true`, если выражение слева (`bonus`) меньше, больше, меньше или равно либо больше или равно выражению справа (`2000`)

И снова программисты на C и C++ должны помнить о том, что старые трюки с проверкой условия, которое включает значение, не равное нулю, в языке C# работать не будут. Пусть необходимо проверить, содержит ли текущая строка более нуля символов. У вас может возникнуть соблазн написать такой код:

```
static void IfElseExample()
{
  // Недопустимо, т.к. свойство Length возвращает int, а не bool.
  string stringData = "My textual data";
  if(stringData.Length)
  {
    Console.WriteLine("string is greater than 0 characters");
  }
}
```

Если вы хотите использовать свойство `String.Length` для определения истинности или ложности, то выражение в условии понадобится изменить так, чтобы оно давало в результате булевское значение:

```
// Допустимо, т.к. условие возвращает true или false.
if(stringData.Length > 0)
{
  Console.WriteLine("string is greater than 0 characters");
}
```

Условные операции

Для выполнения более сложных проверок оператор `if` может также включать сложные выражения и содержать операторы `else`. Синтаксис идентичен своим аналогам в C (C++) и Java. Язык C# предлагает вполне ожидаемый набор условных операций, применяемых при построении сложных выражений (табл. 3.8).

Таблица 3.8. Условные операции C#

Операция	Пример	Описание
`&&`	`if(age == 30 && name == "Fred")`	Операция "И". Возвращает `true`, если все выражения дают `true`
`||`	`if(age == 30 || name == "Fred")`	Операция "ИЛИ". Возвращает `true`, если хотя бы одно из выражений дает `true`
`!`	`if(!myBool)`	Операция "НЕ". Возвращает `true`, если выражение дает `false`, или `false`, если выражение дает `true`

На заметку! Операции `&&` и `||` при необходимости поддерживают сокращенный путь выполнения. Другими словами, после того, как было определено, что сложное выражение должно дать в результате `false`, оставшиеся подвыражения вычисляться не будут. Если требуется, чтобы все выражения вычислялись безотносительно к чему-либо, можно использовать операции `&` и `|`.

Оператор `switch`

Еще одной простой конструкцией C#, предназначенной для реализации выбора, является оператор `switch`. Как и в остальных основанных на C языках, оператор `switch` позволяет организовать выполнение программы на основе заранее определенного набора вариантов. Например, в следующем методе `Main()` для каждого из двух возможных вариантов выводится специфичное сообщение (блок `default` обрабатывает недопустимый выбор):

```
// Переход на основе числового значения.
static void SwitchExample()
{
  Console.WriteLine("1 [C#], 2 [VB]");
  Console.Write("Please pick your language preference: ");
              // Выберите предпочитаемый язык:
  string langChoice = Console.ReadLine();
  int n = int.Parse(langChoice);
  switch (n)
  {
    case 1:
      Console.WriteLine("Good choice, C# is a fine language.");
                      // Хороший выбор. C# - замечательный язык.
    break;
    case 2:
      Console.WriteLine("VB: OOP, multithreading, and more!");
                     // VB: ООП, многопоточность и многое другое!
    break;
```

```
    default:
      Console.WriteLine("Well...good luck with that!"); //Хорошо, удачи с этим!
    break;
  }
}
```

На заметку! Язык C# требует, чтобы каждый блок `case` (включая `default`), который содержит исполняемые операторы, завершался оператором `break` или `goto` во избежание сквозного прохода по блокам.

Одна из замечательных особенностей оператора `switch` в C# связана с тем, что в дополнение к числовым значениям он позволяет оценивать данные `string`. Ниже для примера приведена модифицированная версия предыдущего оператора `switch` (обратите внимание, что при таком подходе разбор введенных пользователем данных в числовые значения не требуется):

```
static void SwitchOnStringExample()
{
  Console.WriteLine("C# or VB");
  Console.Write("Please pick your language preference: ");
  string langChoice = Console.ReadLine();
  switch (langChoice)
  {
    case "C#":
      Console.WriteLine("Good choice, C# is a fine language.");
    break;
    case "VB":
      Console.WriteLine("VB: OOP, multithreading and more!");
    break;
    default:
    Console.WriteLine("Well...good luck with that!");
    break;
  }
}
```

Оператор `switch` может также применяться с перечислимым типом данных. Как будет показано в главе 4, ключевое слово `enum` языка C# позволяет определять специальный набор пар "имя-значение". В качестве иллюстрации рассмотрим следующую вспомогательную функцию, которая выполняет проверку `switch` для перечисления `System.DayOfWeek`. Этот пример содержит ряд синтаксических конструкций, которые пока еще не рассматривались, но сосредоточьте внимание на самом использовании `switch` с типом `enum`; недостающие фрагменты прояснятся в последующих главах.

```
static void SwitchOnEnumExample()
{
  Console.Write("Enter your favorite day of the week: ");
                // Введите любимый день недели:
  DayOfWeek favDay;
  try
  {
    favDay = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), Console.ReadLine());
  }
  catch (Exception)
  {
    Console.WriteLine("Bad input!"); // Недопустимое входное значение!
    return;
  }
```

```
    switch (favDay)
    {
      case DayOfWeek.Friday:
        Console.WriteLine("Yes, Friday rules!");   // Да, пятница рулит!
      break;
      case DayOfWeek.Monday:
        Console.WriteLine("Another day, another dollar");
                        // Еще один день, еще один доллар
      break;
      case DayOfWeek.Saturday:
        Console.WriteLine("Great day indeed.");
                        // Действительно великолепный день.
      break;
      case DayOfWeek.Sunday:
        Console.WriteLine("Football!!");           // Футбол!
      break;
      case DayOfWeek.Thursday:
        Console.WriteLine("Almost Friday...");     // Почти пятница...
      break;
      case DayOfWeek.Tuesday:
        Console.WriteLine("At least it is not Monday");
                        // Во всяком случае, не понедельник
      break;
      case DayOfWeek.Wednesday:
        Console.WriteLine("A fine day.");          // Хороший денек.
      break;
    }
  }
```

Исходный код. Проект `IterationsAndDecisions` доступен в подкаталоге `Chapter_3`.

Резюме

Цель этой главы заключалась в демонстрации многочисленных ключевых аспектов языка программирования C#. Мы исследовали привычные конструкции, которые могут быть задействованы при построении любого приложения. После ознакомления с ролью объекта приложения вы узнали о том, что каждая исполняемая программа на C# должна иметь тип, определяющий метод `Main()`, который служит точкой входа в программу. Внутри метода `Main()` обычно создается любое число объектов, благодаря совместной работе которых приложение приводится в действие.

Затем были подробно описаны встроенные типы данных C# и разъяснено, что применяемые для их представления ключевые слова (например, `int`) на самом деле являются сокращенными обозначениями полноценных типов из пространства имен `System` (`System.Int32` в данном случае). С учетом этого каждый тип данных C# имеет набор встроенных членов. Кроме того, была обсуждена роль расширения и сужения, а также ключевых слов `checked` и `unchecked`.

В завершение главы рассматривалась роль неявной типизации с использованием ключевого слова `var`. Как было отмечено, неявная типизация наиболее полезна при работе с моделью программирования LINQ. И, наконец, мы кратко взглянули на различные конструкции C#, предназначенные для организации циклов и принятия решений.

Теперь, когда вы понимаете некоторые базовые механизмы, в главе 4 будет завершено исследование основных средств языка. После этого вы будете хорошо подготовлены к изучению объектно-ориентированных возможностей C#, которое начнется в главе 5.

ГЛАВА 4

Главные конструкции программирования на C#: часть II

В этой главе завершается обзор основных аспектов языка программирования C#, который был начат в главе 3. Здесь будут рассматриваться различные детали, касающиеся построения методов, в частности, ключевые слова `out`, `ref` и `params`. Наряду с этим вы узнаете о роли необязательных и именованных параметров.

После ознакомления с концепцией *перегрузки методов* вы научитесь манипулировать массивами, используя синтаксис C#, и получите представление о функциональности, которая содержится в связанном с массивами классе `System.Array`.

Вдобавок в главе также будет показано, как создавать типы перечислений и структур, и подробно описаны отличия между *типами значений* и *ссылочными типами*. В завершение главы объясняется роль типов данных, допускающих `null`, и относящихся к ним операций.

После освоения материалов настоящей главы можно смело переходить к изучению объектно-ориентированных возможностей языка C#, рассмотрение которых начнется в главе 5.

Методы и модификаторы параметров

Для начала мы займемся исследованием деталей определения методов. Подобно методу `Main()` (см. главу 3) ваши специальные методы могут принимать или не принимать параметры, а также возвращать или не возвращать значения вызывающему коду. В последующих нескольких главах вы увидите, что методы могут быть реализованы внутри области видимости классов или структур (а заодно прототипироваться внутри интерфейсных типов) и декорированы разнообразными ключевыми словами (например, `static`, `virtual`, `public`, `new`) с целью уточнения их поведения. До настоящего момента в книге каждый из рассматриваемых методов следовал такому базовому формату:

```
// Вспомните, что статические методы могут вызываться
// напрямую без создания экземпляра класса.
class Program
{
  // static возвращаемыйТип ИмяМетода(список параметров) { /* Реализация */ }
  static int Add(int x, int y){ return x + y; }
}
```

Хотя определение метода в C# выглядит довольно-таки понятно, с помощью модификаторов, описанных в табл. 4.1, можно управлять способом передачи аргументов интересующему методу.

Таблица 4.1. Модификаторы параметров в C#

Модификатор параметра	Практический смысл
(отсутствует)	Если параметр не помечен модификатором, то предполагается, что он должен передаваться по значению, т.е. вызываемый метод получает копию исходных данных
`out`	Выходным параметрам должны присваиваться значения внутри вызываемого метода, следовательно, они передаются по ссылке. Если в вызываемом методе выходным параметрам значения не были присвоены, то компилятор сообщит об ошибке
`ref`	Значение первоначально присваивается в вызывающем коде и может быть необязательно изменено в вызываемом методе (поскольку данные также передаются по ссылке). Если в вызываемом методе параметру `ref` значение не присвоено, то никакой ошибки компилятор не генерирует
`params`	Этот модификатор позволяет передавать переменное количество аргументов как единственный логический параметр. Метод может иметь только один модификатор `params`, которым должен быть помечен последний параметр метода. В реальности потребность в использовании модификатора `params` возникает не особенно часто, однако имейте в виду, что он применяется многочисленными методами внутри библиотек базовых классов

Чтобы проиллюстрировать использование этих модификаторов, мы создадим новый проект консольного приложения по имени `FunWithMethods`. А теперь давайте рассмотрим их роль.

Стандартное поведение передачи параметров по значению

По умолчанию параметр передается функции *по значению*. Другими словами, если аргумент не помечен модификатором параметра, то в функцию передается копия данных. Как объясняется в конце этой главы, то, *что* в точности копируется, зависит от того, относится ли параметр к типу значения или же к ссылочному типу. В настоящий момент предположим, что внутри класса `Program` имеется следующий метод, который оперирует с двумя параметрами числового типа, передаваемыми по значению:

```
// По умолчанию аргументы передаются по значению.
static int Add(int x, int y)
{
  int ans = x + y;
  // Вызывающий код не увидит эти изменения,
  // т.к. модифицируется копия исходных данных.
  x = 10000;
  y = 88888;
  return ans;
}
```

Числовые данные относятся к категории *типов значений*. Следовательно, в случае изменения значений параметров внутри контекста члена вызывающий код будет оставаться в полном неведении об этом, поскольку изменения вносятся только в *копию* первоначальных данных из вызывающего кода:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****\n");

  // Передать две переменные по значению.
  int x = 9, y = 10;
  Console.WriteLine("Before call: X: {0}, Y: {1}", x, y);
  Console.WriteLine("Answer is: {0}", Add(x, y));
  Console.WriteLine("After call: X: {0}, Y: {1}", x, y);
  Console.ReadLine();
}
```

Как и следовало ожидать, и это подтверждается показанным ниже выводом, значения x и y остаются идентичными до и после вызова метода `Add()`, потому что элементы данных были переданы по значению. Таким образом, любые изменения параметров, производимые внутри метода `Add()`, вызывающему коду не видны, т.к. метод `Add()` оперирует на копии данных.

```
***** Fun with Methods *****

Before call: X: 9, Y: 10
Answer is: 19
After call: X: 9, Y: 10
```

Модификатор `out`

Далее мы рассмотрим *выходные параметры*. Метод, который был определен для приема выходных параметров (посредством ключевого слова `out`), перед выходом обязан присваивать им соответствующие значения (иначе компилятор сообщит об ошибке).

В целях демонстрации ниже приведена альтернативная версия метода `Add()`, которая возвращает сумму двух целых чисел с применением модификатора `out` (обратите внимание, что возвращаемым значением метода теперь является `void`):

```
//Значения выходных параметров должны быть установлены внутри вызываемого метода.
static void Add(int x, int y, out int ans)
{
  ans = x + y;
}
```

Вызов метода с выходными параметрами также требует использования модификатора `out`. Однако предварительно устанавливать значения локальных переменных, которые передаются в качестве выходных параметров, вовсе не обязательно (после вызова эти значения все равно будут утеряны). Причина, по которой компилятор позволяет передавать на первый взгляд неинициализированные данные, связана с тем, что вызываемый метод *должен* выполнить присваивание. Рассмотрим пример:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
  ...
  // Присваивать начальные значения локальным переменным,
  // используемым как выходные параметры, не обязательно
  // при условии, что в таком качестве они применяются впервые.
  int ans;
  Add(90, 90, out ans);
  Console.WriteLine("90 + 90 = {0}", ans);
  Console.ReadLine();
}
```

Предыдущий пример был показан исключительно для иллюстрации; на самом деле нет никаких причин возвращать значение суммы через выходной параметр. Тем не менее, модификатор `out` в C# служит действительно практичной цели: он позволяет вызывающему коду получать несколько выходных значений из единственного вызова метода.

```
// Возвращение множества выходных параметров.
static void FillTheseValues(out int a, out string b, out bool c)
{
  a = 9;
  b = "Enjoy your string.";
  c = true;
}
```

Теперь вызывающий код может обратиться к методу `FillTheseValues()`, как продемонстрировано ниже. Не забывайте, что модификатор `out` должен применяться как при вызове, так и при реализации данного метода:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
  ...
  int i; string str; bool b;
  FillTheseValues(out i, out str, out b);
  Console.WriteLine("Int is: {0}", i);
  Console.WriteLine("String is: {0}", str);
  Console.WriteLine("Boolean is: {0}", b);
  Console.ReadLine();
}
```

И, наконец, помните, что перед выходом из области видимости метода, определяющего выходные параметры, этим параметрам *должны* быть присвоены допустимые значения. Таким образом, следующий код вызовет ошибку на этапе компиляции, потому что внутри метода отсутствует присваивание значения выходному параметру:

```
static void ThisWontCompile(out int a)
{
  Console.WriteLine("Error! Forgot to assign output arg!");
              // Ошибка! Забыли присвоить значение выходному параметру!
}
```

Модификатор `ref`

А теперь посмотрим, как в C# используется модификатор `ref`. Ссылочные параметры необходимы, когда вы хотите разрешить методу манипулировать различными элементами данных (и обычно изменять их значения), которые объявлены в вызывающем коде, таком как процедура сортировки или обмена. Обратите внимание на следующие отличия между ссылочными и выходными параметрами.

- Выходные параметры не нуждаются в инициализации перед передачей методу. Причина в том, что метод самостоятельно должен присваивать значения выходным параметрам до своего завершения.
- Ссылочные параметры должны быть инициализированы перед передачей методу. Причина связана с передачей ссылок на существующие переменные. Если начальные значения им не присвоены, то это будет равнозначно работе с неинициализированными локальными переменными.

Давайте рассмотрим применение ключевого слова `ref` на примере метода, меняющего местами значения двух переменных типа `string` (естественно, здесь мог бы использоваться любой тип данных, включая `int`, `bool`, `float` и т.д.):

```
// Ссылочные параметры.
public static void SwapStrings(ref string s1, ref string s2)
{
  string tempStr = s1;
  s1 = s2;
  s2 = tempStr;
}
```

Этот метод можно вызвать следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
  ...
  string str1 = "Flip";
  string str2 = "Flop";
  Console.WriteLine("Before: {0}, {1} ", str1, str2);
  SwapStrings(ref str1, ref str2);
  Console.WriteLine("After: {0}, {1} ", str1, str2);
  Console.ReadLine();
}
```

Здесь вызывающий код присваивает начальные значения локальным строковым данным (`str1` и `str2`). После вызова метода `SwapStrings()` строка `str1` будет содержать значение `"Flop"`, а строка `str2` — значение `"Flip"`:

```
Before: Flip, Flop
After: Flop, Flip
```

На заметку! Мы еще возвратимся к ключевому слову `ref` языка C# в разделе "Типы значений и ссылочные типы" далее в главе. Вы увидите, что поведение этого ключевого слова немного изменяется в зависимости от того, является аргумент типом значения или ссылочным типом.

Модификатор `params`

В C# поддерживаются *массивы параметров* с применением ключевого слова `params`. Чтобы понять это языковое средство, необходимо знать, как манипулировать массивами C#. Если данная тема пока вам не знакома, то имеет смысл возвратиться к чтению настоящего раздела после изучения раздела "Понятие массивов C#" далее в главе.

Ключевое слово `params` позволяет передавать методу переменное количество идентично типизированных параметров (или классов, связанных отношением наследования) в виде *единственного логического параметра*. Кроме того, могут обрабатываться аргументы, помеченные ключевым словом `params`, если вызывающий код передает строго типизированный массив или список элементов, разделенных запятыми. Конечно, это может вызывать путаницу. Чтобы стало понятнее, предположим, что вы хотите создать функцию, которая позволит вызывающему коду передавать любое количество аргументов и возвращает их среднее значение.

Если вы прототипируете данный метод так, чтобы он принимал массив значений `double`, то вызывающий код должен будет сначала определить массив, затем заполнить его значениями и, наконец, передать его методу. Однако если вы определите метод `CalculateAverage()` как принимающий параметр `params` типа `double[]`, то вызываю-

щий код может просто передавать список значений double, разделенных запятыми. "За кулисами" исполняющая среда .NET автоматически упакует набор значений double в массив типа double.

```
// Возвращение среднего из некоторого количества значений double.
static double CalculateAverage(params double[] values)
{
  Console.WriteLine("You sent me {0} doubles.", values.Length);

  double sum = 0;
  if(values.Length == 0)
    return sum;
  for (int i = 0; i < values.Length; i++)
    sum += values[i];
  return (sum / values.Length);
}
```

Этот метод был определен для приема массива параметров типа double. По сути, метод ожидает передачи любого количества (включая ноль) значений double и вычисляет их среднее значение. Метод может вызываться любым из показанных ниже способов:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
  ...
  // Передать список значений double, разделенных запятыми...
  double average;
  average = CalculateAverage(4.0, 3.2, 5.7, 64.22, 87.2);
  Console.WriteLine("Average of data is: {0}", average);
  // ...или передать массив значений double.
  double[] data = { 4.0, 3.2, 5.7 };
  average = CalculateAverage(data);
  Console.WriteLine("Average of data is: {0}", average);
  // Среднее из 0 равно 0!
  Console.WriteLine("Average of data is: {0}", CalculateAverage());
  Console.ReadLine();
}
```

При отсутствии модификатора params в определении CalculateAverage() первый же вызов этого метода приведет к ошибке на этапе компиляции, потому что компилятор будет безуспешно искать версию CalculateAverage(), принимающую пять аргументов double.

На заметку! Во избежание любой неоднозначности язык C# требует, чтобы метод поддерживал только один параметр params, который должен быть последним в списке параметров.

Как и можно было догадаться, данный прием — всего лишь удобство для вызывающего кода, т.к. среда CLR самостоятельно создает массив по мере необходимости. В момент, когда массив окажется внутри области видимости вызываемого метода, его можно трактовать как полноценный массив .NET, обладающий всей функциональностью базового библиотечного типа System.Array. Взгляните на вывод:

```
You sent me 5 doubles.
Average of data is: 32.864
You sent me 3 doubles.
Average of data is: 4.3
You sent me 0 doubles.
Average of data is: 0
```

Определение необязательных параметров

Язык C# дает возможность создавать методы, которые могут принимать *необязательные аргументы*. Этот прием позволяет вызывать метод, опуская ненужные аргументы, при условии, что подходят их стандартные значения.

На заметку! Как будет показано в главе 16, главной побудительной причиной для добавления в язык C# необязательных аргументов послужило стремление упростить взаимодействие с объектами COM. Несколько объектных моделей Microsoft (например, Microsoft Office) открывают доступ к своей функциональности через объекты COM, многие из которых были написаны давно и рассчитаны на использование необязательных параметров, не поддерживаемых в ранних версиях C#.

Для иллюстрации работы с необязательными аргументами предположим, что имеется метод по имени `EnterLogData()` с одним необязательным параметром:

```
static void EnterLogData(string message, string owner = "Programmer")
{
  Console.Beep();
  Console.WriteLine("Error: {0}", message);
  Console.WriteLine("Owner of Error: {0}", owner);
}
```

Здесь последнему аргументу `string` было присвоено стандартное значение `"Programmer"` с применением операции присваивания внутри определения параметров. В результате метод `EnterLogData()` можно вызывать из `Main()` двумя способами:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
  ...
  EnterLogData("Oh no! Grid can't find data");
  EnterLogData("Oh no! I can't find the payroll data", "CFO");
  Console.ReadLine();
}
```

Из-за того, что в первом вызове `EnterLogData()` не был указан второй аргумент `string`, будет использоваться его стандартное значение — `"Programmer"`. Во втором вызове `EnterLogData()` для второго аргумента передано значение `"CFO"`.

Важно понимать, что значение, присваиваемое необязательному параметру, должно быть известно на этапе компиляции и не может вычисляться во время выполнения (если вы попытаетесь сделать это, то компилятор сообщит об ошибке). В целях иллюстрации предположим, что модифицировали метод `EnterLogData()`, добавив к нему дополнительный необязательный параметр:

```
// Ошибка! Стандартное значение для необязательного
// аргумента должно быть известно на этапе компиляции!
static void EnterLogData(string message,
          string owner = "Programmer", DateTime timeStamp = DateTime.Now)
{
  Console.Beep();
  Console.WriteLine("Error: {0}", message);
  Console.WriteLine("Owner of Error: {0}", owner);
  Console.WriteLine("Time of Error: {0}", timeStamp);
}
```

Этот не скомпилируется, т.к. значение свойства `Now` класса `DateTime` вычисляется во время выполнения, а не на этапе компиляции.

На заметку! Во избежание неоднозначности необязательные параметры должны всегда помещаться *в конец* сигнатуры метода. Если необязательные параметры появятся перед обязательными, то компилятор сообщит об ошибке.

Вызов методов с использованием именованных параметров

Еще одним полезным языковым средством C# является поддержка *именованных аргументов*. По правде говоря, на первый взгляд может показаться, что данная языковая конструкция способна лишь запутывать код. И если быть полностью честными, то это действительно может быть так! Подобно необязательным аргументам причиной включения поддержки именованных параметров отчасти было желание упростить работу с уровнем взаимодействия с COM (см. главу 16).

Именованные аргументы позволяют вызывать метод с указанием значений параметров в любом желаемом порядке. Таким образом, вместо передачи параметров исключительно по позициям (как это делается в большинстве случаев) можно указывать имя каждого аргумента, двоеточие и конкретное значение. Чтобы продемонстрировать применение именованных аргументов, добавим в класс `Program` следующий метод:

```
static void DisplayFancyMessage(ConsoleColor textColor,
  ConsoleColor backgroundColor, string message)
{
  // Сохранить старые цвета для их восстановления после вывода сообщения.
  ConsoleColor oldTextColor = Console.ForegroundColor;
  ConsoleColor oldbackgroundColor = Console.BackgroundColor;
  // Установить новые цвета и вывести сообщение.
  Console.ForegroundColor = textColor;
  Console.BackgroundColor = backgroundColor;
  Console.WriteLine(message);
  // Восстановить предыдущие цвета.
  Console.ForegroundColor = oldTextColor;
  Console.BackgroundColor = oldbackgroundColor;
}
```

Теперь, когда метод `DisplayFancyMessage()` написан, можно было бы ожидать, что при его вызове будут передаваться две переменные типа `ConsoleColor`, за которыми следует переменная типа `string`. Однако, используя именованные аргументы, метод `DisplayFancyMessage()` допустимо вызывать и так, как показано ниже:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
  ...
  DisplayFancyMessage(message: "Wow! Very Fancy indeed!",
    textColor: ConsoleColor.DarkRed,
    backgroundColor: ConsoleColor.White);
  DisplayFancyMessage(backgroundColor: ConsoleColor.Green,
    message: "Testing...",
    textColor: ConsoleColor.DarkBlue);
  Console.ReadLine();
}
```

С именованными аргументами связана одна особенность — при вызове метода позиционные параметры должны находиться перед всеми именованными параметрами. Другими словами, именованные аргументы должны всегда размещаться в конце вызова метода. Вот пример:

```
// Здесь все в порядке, т.к. позиционные аргументы находятся перед именованными.
DisplayFancyMessage(ConsoleColor.Blue,
  message: "Testing...",
  backgroundColor: ConsoleColor.White);

// Это ОШИБКА, поскольку позиционные аргументы идут после именованных.
DisplayFancyMessage(message: "Testing...",
  backgroundColor: ConsoleColor.White,
  ConsoleColor.Blue);
```

Даже если оставить в стороне указанное ограничение, все равно может возникать вопрос: при каких условиях вообще требуется эта языковая конструкция? В конце концов, для чего нужно менять позиции аргументов метода?

Как выясняется, при наличии метода, в котором определены необязательные аргументы, данное средство может оказаться по-настоящему полезным. Предположим, что метод `DisplayFancyMessage()` переписан с целью поддержки необязательных аргументов, для которых указаны подходящие стандартные значения:

```
static void DisplayFancyMessage(ConsoleColor textColor = ConsoleColor.Blue,
  ConsoleColor backgroundColor = ConsoleColor.White,
  string message = "Test Message")
{
  ...
}
```

Учитывая, что каждый аргумент имеет стандартное значение, именованные аргументы позволяют в вызывающем коде указывать только те параметры, которые не должны принимать стандартные значения. Следовательно, если нужно, чтобы значение `"Hello!"` появлялось в виде текста синего цвета на белом фоне, то в вызывающем коде можно просто записать так:

```
DisplayFancyMessage(message: "Hello!");
```

Если же необходимо, чтобы строка `"Test Message"` выводилась синим цветом на зеленом фоне, то должен применяться такой вызов:

```
DisplayFancyMessage(backgroundColor: ConsoleColor.Green);
```

Как видите, необязательные аргументы и именованные параметры часто работают бок о бок. В завершение темы построения методов C# необходимо ознакомиться с концепцией *перегрузки методов*.

Исходный код. Проект `FunWithMethods` доступен в подкаталоге `Chapter_4`.

Понятие перегрузки методов

Подобно другим современным языкам объектно-ориентированного программирования в C# разрешена *перегрузка* методов. Выражаясь просто, когда определяется набор идентично именованных методов, которые отличаются друг от друга количеством (или типами) параметров, то говорят, что такой метод был перегружен.

Чтобы оценить удобство перегрузки методов, давайте представим себя на месте разработчика, использующего Visual Basic 6.0 (VB6). Предположим, что требуется создать набор методов, возвращающих сумму значений разнообразных типов (`Integer`, `Double` и т.д.). С учетом того, что VB6 не поддерживает перегрузку методов, придется определить уникальный набор методов, каждый из которых будет делать по существу одно и то же (возвращать сумму аргументов):

```
' Примеры кода VB6.
Public Function AddInts(ByVal x As Integer, ByVal y As Integer) As Integer
  AddInts = x + y
End Function
Public Function AddDoubles(ByVal x As Double, ByVal y As Double) As Double
  AddDoubles = x + y
End Function
Public Function AddLongs(ByVal x As Long, ByVal y As Long) As Long
  AddLongs = x + y
End Function
```

Такой код не только становится трудным в сопровождении, но и заставляет помнить имена всех методов. Применяя перегрузку, вызывающему коду можно предоставить возможность обращения к единственному методу по имени `Add()`. Ключевой аспект в том, чтобы обеспечить для каждой версии метода отличающийся набор аргументов (различий только в возвращаемом типе не достаточно).

На заметку! Как объясняется в главе 9, существует возможность построения обобщенных методов, которые переносят концепцию перегрузки на новый уровень. Используя обобщения, можно определять *заполнители типов* для реализации метода, которая указывается во время вызова данного члена.

Чтобы попрактиковаться с перегруженными методами, создадим новый проект консольного приложения по имени `MethodOverloading` и добавим в него следующее определение класса:

```
// Код C#.
class Program
{
  static void Main(string[] args)
  {
  }

  // Перегруженный метод Add().
  static int Add(int x, int y)
  { return x + y; }

  static double Add(double x, double y)
  { return x + y; }
  static long Add(long x, long y)
  { return x + y; }
}
```

Теперь вызывающий код может просто обращаться к методу `Add()` с требуемыми аргументами, а компилятор будет самостоятельно находить подходящую для вызова реализацию на основе предоставленных аргументов:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Method Overloading *****\n");

  // Вызов int-версии Add().
  Console.WriteLine(Add(10, 10));
  // Вызов long-версии Add().
  Console.WriteLine(Add(900000000000, 900000000000));
  // Вызов double-версии Add().
  Console.WriteLine(Add(4.3, 4.4));

  Console.ReadLine();
}
```

Среда Visual Studio оказывает помощь при вызове перегруженных методов. Когда вводится имя перегруженного метода (такого как хорошо знакомый метод `Console.WriteLine()`), средство IntelliSense отображает список всех его доступных версий. Обратите внимание, что по списку можно перемещаться с применением клавиш со стрелками вниз и вверх (рис. 4.1).

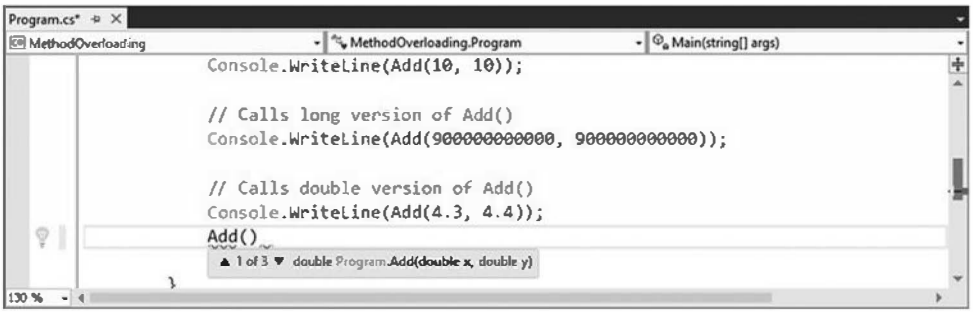


Рис. 4.1. Средство IntelliSense в Visual Studio для перегруженных методов

Исходный код. Проект `MethodOverloading` доступен в подкаталоге `Chapter_4`.

На этом начальный обзор построения методов с использованием синтаксиса C# завершен. Теперь давайте посмотрим, каким образом создавать и манипулировать массивами, перечислениями и структурами.

Понятие массивов C#

Как вам уже наверняка известно, *массив* — это набор элементов данных, для доступа к которым применяется числовой индекс. Выражаясь более конкретно, массив представляет собой набор расположенных рядом элементов данных одного и того же типа (массив элементов `int`, массив элементов `string`, массив элементов `SportCar` и т.д.). Объявлять, заполнять и получать доступ к массиву в языке C# довольно просто. В целях иллюстрации создадим новый проект консольного приложения (по имени `FunWithArrays`), который содержит вспомогательный метод `SimpleArrays()`, вызываемый из `Main()`:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Arrays *****");
    SimpleArrays();
    Console.ReadLine();
  }
  static void SimpleArrays()
  {
    Console.WriteLine("=> Simple Array Creation.");
    // Создать массив int, содержащий 3 элемента с индексами 0, 1, 2.
    int[] myInts = new int[3];
    // Создать массив string, содержащий 100 элементов с индексами 0 - 99.
    string[] booksOnDotNet = new string[100];
    Console.WriteLine();
  }
}
```

Взгляните на комментарии в коде. При объявлении массива C# с использованием подобного синтаксиса число, указанное в объявлении, обозначает общее количество элементов, а не верхнюю границу. Кроме того, обратите внимание, что нижняя граница в массиве всегда начинается с 0. Таким образом, в результате записи `int[] myInts = new int[3]` получается массив, который содержит три элемента, проиндексированные по позициям 0, 1, 2.

После определения переменной массива можно переходить к заполнению элементов от индекса к индексу, как показано ниже в модифицированном методе `SimpleArrays()`:

```
static void SimpleArrays()
{
  Console.WriteLine("=> Simple Array Creation.");

  // Создать и заполнить массив из трех целочисленных значений.
  int[] myInts = new int[3];
  myInts[0] = 100;
  myInts[1] = 200;
  myInts[2] = 300;

  // Вывести все значения.
  foreach(int i in myInts)
    Console.WriteLine(i);
  Console.WriteLine();
}
```

На заметку! Имейте в виду, что если массив объявлен, но его элементы явно не заполнены по каждому индексу, то они получат стандартное значение для соответствующего типа данных (например, элементы массива `bool` будут установлены в `false`, а элементы массива `int` — в 0).

Синтаксис инициализации массивов C#

В дополнение к заполнению массива элемент за элементом есть также возможность заполнять его с применением *синтаксиса инициализации массива*. Для этого понадобится указать значения всех элементов массива в фигурных скобках (`{}`). Такой синтаксис удобен при создании массива известного размера, когда нужно быстро задать его начальные значения. Например, вот как выглядят альтернативные версии объявления массива:

```
static void ArrayInitialization()
{
  Console.WriteLine("=> Array Initialization.");

  // Синтаксис инициализации массива с использованием ключевого слова new.
  string[] stringArray = new string[]
    { "one", "two", "three" };
  Console.WriteLine("stringArray has {0} elements", stringArray.Length);

  // Синтаксис инициализации массива без использования ключевого слова new.
  bool[] boolArray = { false, false, true };
  Console.WriteLine("boolArray has {0} elements", boolArray.Length);

  //Инициализация массива с применением ключевого слова new и указанием размера.
  int[] intArray = new int[4] { 20, 22, 23, 0 };
  Console.WriteLine("intArray has {0} elements", intArray.Length);
  Console.WriteLine();
}
```

Обратите внимание, что в случае использования синтаксиса с фигурными скобками нет необходимости указывать размер массива (как видно на примере создания переменной stringArray), поскольку размер автоматически вычисляется на основе количества элементов внутри фигурных скобок. Кроме того, применять ключевое слово new не обязательно (как при создании массива boolArray).

В случае объявления intArray снова вспомните, что указанное числовое значение представляет количество элементов в массиве, а не верхнюю границу. Если объявленный размер и количество инициализаторов не совпадают (инициализаторов либо слишком много, либо их не хватает), то на этапе компиляции возникнет ошибка. Пример представлен ниже:

```
// Несоответствие размера и количества элементов!
int[] intArray = new int[2] { 20, 22, 23, 0 };
```

Неявно типизированные локальные массивы

В главе 3 рассматривалась тема неявно типизированных локальных переменных. Как вы помните, ключевое слово var позволяет определять переменную, тип которой выводится компилятором. Аналогичным образом ключевое слово var можно использовать для определения *неявно типизированных локальных массивов*. Такой подход позволяет выделять память под новую переменную массива, не указывая тип элементов внутри массива (обратите внимание, что применение этого подхода предусматривает обязательное использование ключевого слова new):

```
static void DeclareImplicitArrays()
{
  Console.WriteLine("=> Implicit Array Initialization.");

  // Переменная a на самом деле имеет тип int[].
  var a = new[] { 1, 10, 100, 1000 };
  Console.WriteLine("a is a: {0}", a.ToString());
  // Переменная b на самом деле имеет тип double[].
  var b = new[] { 1, 1.5, 2, 2.5 };
  Console.WriteLine("b is a: {0}", b.ToString());
  // Переменная c на самом деле имеет тип string[].
  var c = new[] { "hello", null, "world" };
  Console.WriteLine("c is a: {0}", c.ToString());
  Console.WriteLine();
}
```

Разумеется, как и при создании массива с применением явного синтаксиса C#, элементы в списке инициализации массива должны принадлежать к одному и тому же типу (например, должны быть все int, все string или все SportsCar). В отличие от возможных ожиданий, неявно типизированный локальный массив не получает по умолчанию тип System.Object, так что следующий код приведет к ошибке на этапе компиляции:

```
// Ошибка! Смешанные типы!
var d = new[] { 1, "one", 2, "two", false };
```

Определение массива элементов типа `object`

В большинстве случаев массив определяется путем указания явного типа элементов, которые могут в нем содержаться. Хотя это выглядит довольно прямолинейным, существует одна важная особенность. Как будет показано в главе 6, изначальным базовым классом для каждого типа (включая фундаментальные типы данных) в системе типов .NET является System.Object. С учетом этого факта, если определить массив типа данных System.Object, то его элементы могут представлять все что угодно. Взгляните на

следующий метод ArrayOfObjects(), который для тестирования может быть вызван внутри Main():

```
static void ArrayOfObjects()
{
  Console.WriteLine("=> Array of Objects.");

  // Массив объектов может содержать все что угодно.
  object[] myObjects = new object[4];
  myObjects[0] = 10;
  myObjects[1] = false;
  myObjects[2] = new DateTime(1969, 3, 24);
  myObjects[3] = "Form & Void";
  foreach (object obj in myObjects)
  {
    // Вывести тип и значение каждого элемента в массиве.
    Console.WriteLine("Type: {0}, Value: {1}", obj.GetType(), obj);
  }
  Console.WriteLine();
}
```

Здесь во время итерации по содержимому массива myObjects для каждого элемента выводится лежащий в основе тип, получаемый с помощью метода GetType() класса System.Object, и его значение. Не вдаваясь пока в детали System.Object.GetType(), просто отметим, что этот метод может использоваться для получения полностью заданного имени элемента (службы извлечения информации о типах и рефлексии исследуются в главе 15). Приведенный далее вывод является результатом вызова метода ArrayOfObjects():

```
=> Array of Objects.
Type: System.Int32, Value: 10
Type: System.Boolean, Value: False
Type: System.DateTime, Value: 3/24/1969 12:00:00 AM
Type: System.String, Value: Form & Void
```

Работа с многомерными массивами

В дополнение к одномерным массивам, которые вы видели до сих пор, язык C# также поддерживает два вида многомерных массивов. Первый вид называется *прямоугольным массивом*, который имеет несколько измерений, а содержащиеся в нем строки обладают одной и той же длиной. Прямоугольный многомерный массив объявляется и заполняется следующим образом:

```
static void RectMultidimensionalArray()
{
  Console.WriteLine("=> Rectangular multidimensional array.");
  // Прямоугольный многомерный массив.
  int[,] myMatrix;
  myMatrix = new int[3,4];

  // Заполнить массив (3 * 4).
  for(int i = 0; i < 3; i++)
    for(int j = 0; j < 4; j++)
      myMatrix[i, j] = i * j;

  // Вывести содержимое массива (3 * 4).
  for(int i = 0; i < 3; i++)
  {
```

```
    for(int j = 0; j < 4; j++)
      Console.Write(myMatrix[i, j] + "\t");
    Console.WriteLine();
  }
  Console.WriteLine();
}
```

Второй вид многомерных массивов носит название *зубчатого (или ступенчатого) массива.* Такой массив содержит какое-то число внутренних массивов, каждый из которых может иметь отличающийся верхний предел. Вот пример:

```
static void JaggedMultidimensionalArray()
{
  Console.WriteLine("=> Jagged multidimensional array.");
  // Зубчатый многомерный массив (т.е. массив массивов).
  // Здесь мы имеем массив из 5 разных массивов.
  int[][] myJagArray = new int[5][];

  // Создать зубчатый массив.
  for (int i = 0; i < myJagArray.Length; i++)
    myJagArray[i] = new int[i + 7];

  // Вывести все строки (помните, что каждый элемент имеет
  // стандартное значение 0).
  for(int i = 0; i < 5; i++)
  {
    for(int j = 0; j < myJagArray[i].Length; j++)
      Console.Write(myJagArray[i][j] + " ");
    Console.WriteLine();
  }
  Console.WriteLine();
}
```

Ниже показан вывод, полученный в результате вызова внутри `Main()` методов `RectMultidimensionalArray()` и `JaggedMultidimensionalArray()`:

```
=> Rectangular multidimensional array:

0       0       0       0
0       1       2       3
0       2       4       6

=> Jagged multidimensional array:

0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

Использование массивов в качестве аргументов и возвращаемых значений

После создания массив можно передавать как аргумент или получать его в виде возвращаемого значения. Например, приведенный ниже метод `PrintArray()` принимает входной массив значений `int` и выводит все его элементы на консоль, а метод `GetStringArray()` заполняет массив значений `string` и возвращает его вызывающему коду:

```
static void PrintArray(int[] myInts)
{
  for(int i = 0; i < myInts.Length; i++)
    Console.WriteLine("Item {0} is {1}", i, myInts[i]);
}
static string[] GetStringArray()
{
  string[] theStrings = {"Hello", "from", "GetStringArray"};
  return theStrings;
}
```

Эти методы могут вызываться вполне ожидаемым образом:

```
static void PassAndReceiveArrays()
{
  Console.WriteLine("=> Arrays as params and return values.");
  // Передать массив в качестве параметра.
  int[] ages = {20, 22, 23, 0} ;
  PrintArray(ages);

  // Получить массив как возвращаемое значение.
  string[] strs = GetStringArray();
  foreach(string s in strs)
    Console.WriteLine(s);

  Console.WriteLine();
}
```

К настоящему моменту вы должны освоить процесс определения, заполнения и исследования содержимого переменной типа массива C#. Для полноты картины давайте проанализируем роль класса `System.Array`.

Базовый класс `System.Array`

Каждый создаваемый массив получает значительную часть своей функциональности от класса `System.Array`. Общие члены этого класса позволяют работать с массивом, применяя согласованную объектную модель. В табл. 4.2 приведено краткое описание наиболее интересных членов класса `System.Array` (полное описание всех членов данного класса можно найти в документации .NET Framework 4.6 SDK).

Таблица 4.2. Избранные члены класса `System.Array`

Член класса `System.Array`	Описание
`Clear()`	Этот статический метод устанавливает для заданного диапазона элементов в массиве пустые значения (0 для чисел, `null` для объектных ссылок и `false` для булевских выражений)
`CopyTo()`	Этот метод используется для копирования элементов из исходного массива в целевой массив
`Length`	Это свойство возвращает количество элементов в массиве
`Rank`	Это свойство возвращает количество измерений в массиве
`Reverse()`	Этот статический метод обращает содержимое одномерного массива
`Sort()`	Этот статический метод сортирует одномерный массив внутренних типов. Если элементы в массиве реализуют интерфейс `IComparer`, можно также сортировать специальные типы (глава 9)

Давайте посмотрим на некоторые из этих членов в действии. Показанный далее вспомогательный метод использует статические методы `Reverse()` и `Clear()` для вывода на консоль информации о массиве строковых типов:

```
static void SystemArrayFunctionality()
{
  Console.WriteLine("=> Working with System.Array.");
  // Инициализировать элементы при запуске.
  string[] gothicBands = {"Tones on Tail", "Bauhaus", "Sisters of Mercy"};

  // Вывести имена в порядке их объявления.
  Console.WriteLine("-> Here is the array:");
  for (int i = 0; i < gothicBands.Length; i++)
  {
    // Вывести имя.
    Console.Write(gothicBands[i] + ", ");
  }
  Console.WriteLine("\n");

  // Обратить порядок следования элементов...
  Array.Reverse(gothicBands);
  Console.WriteLine("-> The reversed array");

  // ... и вывести их.
  for (int i = 0; i < gothicBands.Length; i++)
  {
    // Вывести имя.
    Console.Write(gothicBands[i] + ", ");
  }
  Console.WriteLine("\n");

  // Удалить все элементы кроме первого.
  Console.WriteLine("-> Cleared out all but one...");
  Array.Clear(gothicBands, 1, 2);
  for (int i = 0; i < gothicBands.Length; i++)
  {
    // Вывести имя.
    Console.Write(gothicBands[i] + ", ");
  }
  Console.WriteLine();
}
```

Вызов этого метода в `Main()` дает в результате следующий вывод:

```
=> Working with System.Array.
-> Here is the array:
Tones on Tail, Bauhaus, Sisters of Mercy,
-> The reversed array
Sisters of Mercy, Bauhaus, Tones on Tail,

-> Cleared out all but one...
Sisters of Mercy, , ,
```

Обратите внимание, что многие члены класса `System.Array` определены как статические и потому вызываются на уровне класса (примерами могут служить методы `Array.Sort()` и `Array.Reverse()`). Методам подобного рода передается массив, подлежащий обработке. Другие члены `System.Array` (такие как свойство `Length`) действуют на уровне объекта, поэтому могут вызываться прямо на типе массива.

Исходный код. Проект `FunWithArrays` доступен в подкаталоге `Chapter_4`.

Тип enum

Вспомните из главы 1, что система типов .NET образована из классов, структур, перечислений, интерфейсов и делегатов. Чтобы начать исследование этих типов, давайте рассмотрим роль *перечисления* (enum), создав новый проект консольного приложения по имени FunWithEnums.

На заметку! Не путайте термины *перечисление* и *перечислитель*; они обозначают совершенно разные концепции. Перечисление — это специальный тип данных, состоящих из пар "имя-значение". Перечислитель — это тип класса или структуры, который реализует интерфейс .NET по имени IEnumerable. Обычно этот интерфейс реализуется классами коллекций, а также классом System.Array. Как будет показано в главе 8, объекты, поддерживающие IEnumerable, могут работать с циклами foreach.

При построении какой-либо системы зачастую удобно создавать набор символических имен, которые отображаются на известные числовые значения. Например, в случае создания системы начисления заработной платы может возникнуть необходимость в ссылке на типы сотрудников с применением констант вроде VicePresident (вице-президент), Manager (менеджер), Contractor (подрядчик) и Grunt (рядовой сотрудник). Для этой цели в C# поддерживается понятие специальных перечислений. Например, далее представлено специальное перечисление по имени EmpType (его можно определить в том же файле, где находится класс Program, прямо перед определением класса):

```
// Специальное перечисление.
enum EmpType
{
  Manager,        // = 0
  Grunt,          // = 1
  Contractor,     // = 2
  VicePresident   // = 3
}
```

В перечислении EmpType определены четыре именованных константы, соответствующие дискретным числовым значениям. По умолчанию первому элементу присваивается значение 0, а остальным элементам значения устанавливаются по схеме n+1. При желании исходное значение можно изменять подходящим образом. Например, если имеет смысл нумеровать члены EmpType с 102 до 105, то можно поступить следующим образом:

```
// Начать нумерацию со значения 102.
enum EmpType
{
  Manager = 102,
  Grunt,          // = 103
  Contractor,     // = 104
  VicePresident   // = 105
}
```

Нумерация в перечислениях не обязательно должна быть последовательной и содержать только уникальные значения. Если (по той или иной причине) перечисление EmpType необходимо сконфигурировать так, как показано ниже, то компиляция пройдет гладко и без ошибок:

```
//Значения элементов в перечислении не обязательно должны быть последовательными!
enum EmpType
```

```
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 9
}
```

Управление хранилищем, лежащим в основе перечисления

По умолчанию для хранения значений перечисления используется тип `System.Int32` (`int` в языке C#); тем не менее, при желании его легко заменить. Перечисления в C# можно определять в похожей манере для любых основных системных типов (`byte`, `short`, `int` или `long`). Например, чтобы значения перечисления `EmpType` хранились с применением типа `byte`, а не `int`, можно записать так:

```
// На этот раз для элементов EmpType используется тип byte.
enum EmpType : byte
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 9
}
```

Изменение типа, лежащего в основе перечисления, может быть полезным при построении приложения .NET, которое планируется развертывать на устройствах с небольшим объемом памяти, поэтому необходимо экономить память везде, где только возможно. Конечно, если в качестве типа хранилища для перечисления указан `byte`, то каждое значение должно входить в диапазон его допустимых значений. Например, следующая версия `EmpType` приведет к ошибке на этапе компиляции, т.к. значение 999 не умещается в диапазон допустимых значений типа `byte`:

```
// Ошибка на этапе компиляции! Значение 999 слишком велико для типа byte!
enum EmpType : byte
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 999
}
```

Объявление переменных типа перечисления

После установки диапазона и типа хранилища перечисление можно использовать вместо так называемых "магических чисел". Поскольку перечисления — это всего лишь определяемые пользователем типы данных, их можно применять как возвращаемые значения функций, параметры методов, локальные переменные и т.д. Предположим, что есть метод по имени `AskForBonus()`, который принимает в качестве единственного параметра переменную `EmpType`. Базируясь на значении этого входного параметра, в окно консоли будет выводиться подходящий ответ на запрос о надбавке к зарплате.

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("**** Fun with Enums *****");
    // Создать переменную типа EmpType.
```

```
    EmpType emp = EmpType.Contractor;
    AskForBonus(emp);
    Console.ReadLine();
  }
  // Перечисления как параметры.
  static void AskForBonus(EmpType e)
  {
    switch (e)
    {
      case EmpType.Manager:
        Console.WriteLine("How about stock options instead?");
                        // Не желаете ли взамен фондовые опционы?
      break;
      case EmpType.Grunt:
        Console.WriteLine("You have got to be kidding...");
                        // Вы, должно быть, шутите...
      break;
      case EmpType.Contractor:
        Console.WriteLine("You already get enough cash...");
                        // Вы уже получаете вполне достаточно...
      break;
      case EmpType.VicePresident:
        Console.WriteLine("VERY GOOD, Sir!");
                        // Очень хорошо, сэр!
      break;
    }
  }
}
```

Обратите внимание, что когда переменной `enum` **присваивается значение, вы должны указывать перед этим значением (**`Grunt`**) имя самого перечисления (**`EmpType`**). Из-за того, что перечисления представляют собой фиксированные наборы пар "имя-значение", установка переменной** `enum` **в значение, которое не определено прямо в перечислимом типе, не допускается:**

```
static void ThisMethodWillNotCompile()
{
  // Ошибка! SalesManager отсутствует в перечислении EmpType!
  EmpType emp = EmpType.SalesManager;
  // Ошибка! Не указано имя EmpType перед значением Grunt!
  emp = Grunt;
}
```

Тип `System.Enum`

С перечислениями .NET связан один интересный аспект — они получают свою функциональность от класса `System.Enum`**. В этом классе определено множество методов, которые позволяют исследовать и трансформировать заданное перечисление. Одним из них является метод** `Enum.GetUnderlyingType()`**, который возвращает тип данных, используемый для хранения значений перечислимого типа (в текущем объявлении** `EmpType` **это** `System.Byte`**):**

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
  // Создать переменную типа EmpType.
  EmpType emp = EmpType.Contractor;
  AskForBonus(emp);
```

```
  // Вывести тип хранилища для значений перечисления.
  Console.WriteLine("EmpType uses a {0} for storage",
    Enum.GetUnderlyingType(emp.GetType()));
  Console.ReadLine();
}
```

Заглянув в браузер объектов Visual Studio, можно удостовериться, что метод `Enum.GetUnderlyingType()` требует передачи `System.Type` в качестве первого параметра. В главе 15 будет показано, что класс `Type` представляет описание метаданных для конкретной сущности .NET.

Один из возможных способов получения метаданных (как было показано ранее) предусматривает применение метода `GetType()`, который является общим для всех типов в библиотеках базовых классов .NET. Другой подход заключается в использовании операции `typeof` языка C#. Преимущество этого способа связано с тем, что он не требует объявления переменной сущности, описание метаданных которой требуется получить:

```
// На этот раз для получения информации о типе используется операция typeof.
Console.WriteLine("EmpType uses a {0} for storage",
   Enum.GetUnderlyingType(typeof(EmpType)));
```

Динамическое извлечение пар "имя-значение" перечисления

Кроме метода `Enum.GetUnderlyingType()` все перечисления C# поддерживают метод по имени `ToString()`, который возвращает строковое имя текущего значения перечисления. Ниже приведен пример:

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
  EmpType emp = EmpType.Contractor;
  AskForBonus(emp);
  // Выводит строку "emp is a Contractor".
  Console.WriteLine("emp is a {0}.", emp.ToString());
  Console.ReadLine();
}
```

Если интересует не имя, а значение заданной переменной перечисления, то можно просто привести ее к лежащему в основе типу хранения, например:

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
  EmpType emp = EmpType.Contractor;
  ...
  // Выводит строку "Contractor = 100".
  Console.WriteLine("{0} = {1}", emp.ToString(), (byte)emp);
  Console.ReadLine();
}
```

На заметку! Статический метод `Enum.Format()` предлагает более высокий уровень форматирования за счет указания флага желаемого формата. Более подробную информацию о методе `System.Enum.Format()` ищите в документации .NET Framework 4.6 SDK.

В типе `System.Enum` определен еще один статический метод по имени `GetValues()`. Этот метод возвращает экземпляр класса `System.Array`. Каждый элемент в массиве соответствует члену в указанном перечислении. Рассмотрим следующий метод, который выводит на консоль пары "имя-значение" из перечисления, переданного в качестве параметра:

```
// Этот метод выводит детали любого перечисления.
static void EvaluateEnum(System.Enum e)
{
  Console.WriteLine("=> Information about {0}", e.GetType().Name);
  Console.WriteLine("Underlying storage type: {0}",
    Enum.GetUnderlyingType(e.GetType()));
  // Получить все пары "имя-значение" для входного параметра.
  Array enumData = Enum.GetValues(e.GetType());
  Console.WriteLine("This enum has {0} members.", enumData.Length);
  // Вывести строковое имя и ассоциированное значение,
  // используя флаг формата D (см. главу 3).
  for(int i = 0; i < enumData.Length; i++)
  {
    Console.WriteLine("Name: {0}, Value: {0:D}", enumData.GetValue(i));
  }
  Console.WriteLine();
}
```

Чтобы протестировать этот новый метод, модифицируем `Main()` для создания переменных нескольких типов перечислений, объявленных в пространстве имен `System` (вместе с перечислением `EmpType`):

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
  ...
  EmpType e2 = EmpType.Contractor;
  // Эти типы являются перечислениями из пространства имен System.
  DayOfWeek day = DayOfWeek.Monday;
  ConsoleColor cc = ConsoleColor.Gray;
  EvaluateEnum(e2);
  EvaluateEnum(day);
  EvaluateEnum(cc);
  Console.ReadLine();
}
```

Ниже показана часть вывода:

```
=> Information about DayOfWeek
Underlying storage type: System.Int32
This enum has 7 members.
Name: Sunday, Value: 0
Name: Monday, Value: 1
Name: Tuesday, Value: 2
Name: Wednesday, Value: 3
Name: Thursday, Value: 4
Name: Friday, Value: 5
Name: Saturday, Value: 6
```

Как вы увидите в ходе чтения этой книги, перечисления широко применяются во всех библиотеках базовых классов .NET. Например, в ADO.NET используются многочисленные перечисления для представления состояния подключения к базе данных (открыто или закрыто) либо состояния строки в объекте `DataTable` (изменена, новая или отсоединена). Таким образом, при работе с любым перечислением всегда помните о возможности взаимодействия с парами "имя-значение", применяя члены класса `System.Enum`.

Исходный код. Проект `FunWithEnums` доступен в подкаталоге `Chapter_4`.

Понятие структуры (как типа значения)

Теперь, когда вы понимаете роль типов перечислений, давайте посмотрим, как использовать *структуры* .NET. Типы структур хорошо подходят для моделирования в приложении математических, геометрических и других "атомарных" сущностей. Структура (такая как перечисление) — это определяемый пользователем тип; тем не менее, структура не является просто коллекцией пар "имя-значение". Взамен структуры представляют собой типы, которые могут содержать любое количество полей данных и членов, действующих на этих полях.

На заметку! Если вы имеете опыт объектно-ориентированного программирования, то можете считать структуры "легковесными типами классов", т.к. они предоставляют способ определения типа, который поддерживает инкапсуляцию, но не может использоваться для построения семейства взаимосвязанных типов. Когда возникает потребность в создании семейства типов, связанных отношением наследования, необходимо применять классы.

На первый взгляд процесс определения и использования структур выглядит простым, но, как часто бывает, самое сложное скрыто в деталях. Чтобы приступить к изучению основ типов структур, создадим новый проект по имени `FunWithStructures`. В языке C# структуры определяются с применением ключевого слова `struct`. Определим новую структуру по имени `Point`, которая содержит две переменные типа `int` и набор методов для взаимодействия с ними:

```
struct Point
{
  // Поля структуры.
  public int X;
  public int Y;
  // Добавить 1 к позиции (X, Y).
  public void Increment()
  {
    X++; Y++;
  }
  // Вычесть 1 из позиции (X, Y).
  public void Decrement()
  {
    X--; Y--;
  }
  // Отобразить текущую позицию.
  public void Display()
  {
    Console.WriteLine("X = {0}, Y = {1}", X, Y);
  }
}
```

Здесь мы определили два целочисленных поля (`X` и `Y`), используя ключевое слово `public`, которое является модификатором управления доступом (более подробно эта тема раскрывается в главе 5). Объявление данных с ключевым словом `public` обеспечивает вызывающему коду возможность прямого доступа к этим данным через переменную `Point` (посредством операции точки).

На заметку! Определение открытых данных внутри класса или структуры обычно считается плохим стилем кодирования. Вместо этого рекомендуется определять *закрытые* данные, доступ и изменение которых производится с применением *открытых* свойств. Более подробные сведения можно найти в главе 5.

Вот метод `Main()`, который позволяет протестировать тип `Point`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** A First Look at Structures *****\n");

  // Создать начальную переменную типа Point.
  Point myPoint;
  myPoint.X = 349;
  myPoint.Y = 76;
  myPoint.Display();

  // Скорректировать значения X и Y.
  myPoint.Increment();
  myPoint.Display();
  Console.ReadLine();
}
```

Вывод выглядит вполне ожидаемо:

```
***** A First Look at Structures *****

X = 349, Y = 76
X = 350, Y = 77
```

Создание переменных типа структур

Для создания переменной типа структуры на выбор доступно несколько вариантов. В следующем коде мы просто создаем переменную типа `Point` и присваиваем значения каждому ее открытому полю данных до того, как обращаться к членам этой переменной. Если *не* присвоить значения открытым полям данных (в этом случае `X` и `Y`) перед использованием структуры, то компилятор сообщит об ошибке:

```
// Ошибка! Полю Y не присвоено значение.
Point p1;
p1.X = 10;
p1.Display();

// Все в порядке! Перед использованием значения присвоены обоим полям.
Point p2;
p2.X = 10;
p2.Y = 10;
p2.Display();
```

В качестве альтернативы переменные типа структур можно создавать с применением ключевого слова `new` языка C#, что приводит к вызову *стандартного конструктора* структуры. По определению стандартный конструктор не принимает аргументов. Преимущество вызова стандартного конструктора структуры связано с тем, что каждое поле данных автоматически получает свое стандартное значение:

```
// Установить для всех полей стандартные значения,
// используя стандартный конструктор.
Point p1 = new Point();

// Выводит X=0,Y=0
p1.Display();
```

Допускается также проектирование структуры со *специальным конструктором*. Это позволит указывать значения для полей данных при создании переменной, а не устанавливать их по отдельности. Конструкторы подробно рассматриваются в главе 5; однако в целях иллюстрации изменим структуру `Point` следующим образом:

```
struct Point
{
  // Поля структуры.
  public int X;
  public int Y;

  // Специальный конструктор.
  public Point(int XPos, int YPos)
  {
    X = XPos;
    Y = YPos;
  }
  ...
}
```

Затем переменные типа `Point` можно создавать так:

```
// Вызвать специальный конструктор.
Point p2 = new Point(50, 60);

// Выводит X=50,Y=60
p2.Display();
```

Как упоминалось ранее, работа со структурами на первый взгляд довольно проста. Тем не менее, чтобы углубить понимание особенностей этого типа, необходимо ознакомиться с отличиями между типами значений и ссылочными типами .NET.

Исходный код. Проект `FunWithStructures` доступен в подкаталоге `Chapter_4`.

Типы значений и ссылочные типы

На заметку! В последующем обсуждении типов значений и ссылочных типов предполагается наличие у вас базовых знаний объектно-ориентированного программирования. Если это не так, то имеет смысл возвратиться к чтению настоящего раздела после изучения глав 5 и 6.

В отличие от массивов, строк и перечислений структуры C# не имеют идентично именуемого представления в библиотеке .NET (т.е. класс вроде `System.Structure` отсутствует), но они являются неявно производными от абстрактного класса `System.ValueType`. Выражаясь просто, роль класса `System.ValueType` заключается в обеспечении размещения экземпляра производного типа (например, любой структуры) в *стеке*, а не в *куче* с автоматической сборкой мусора. Данные, размещаемые в стеке, могут создаваться и уничтожаться быстро, т.к. время их жизни определяется областью видимости, в которой они объявлены. С другой стороны, данные, размещаемые в куче, отслеживаются сборщиком мусора .NET и имеют время жизни, которое определяется большим числом факторов, объясняемых в главе 13.

Функционально единственное назначение класса `System.ValueType` — переопределение виртуальных методов, объявленных в классе `System.Object`, с целью использования семантики на основе значений, а не ссылок. Скорее всего, вы уже знаете, что переопределение представляет собой процесс изменения реализации виртуального (или возможно абстрактного) метода, определенного внутри базового класса. Базовым классом для `ValueType` является `System.Object`. В действительности методы экземпляра, определенные в `System.ValueType`, идентичны методам экземпляра, которые определены в `System.Object`:

```
// Структуры и перечисления неявно расширяют класс System.ValueType.
public abstract class ValueType : object
{
  public virtual bool Equals(object obj);
  public virtual int GetHashCode();
  public Type GetType();
  public virtual string ToString();
}
```

Учитывая, что типы значений применяют семантику на основе значений, время жизни структуры (что относится ко всем числовым типам данных (`int`, `float`), а также к любому перечислению или структуре) предсказуемо. Когда переменная типа структуры покидает область определения, она немедленно удаляется из памяти:

```
// Локальные структуры извлекаются из стека, когда метод возвращает управление.
static void LocalValueTypes()
{
  // Вспомните, что int - это на самом деле структура System.Int32.
  int i = 0;

  // Вспомните, что Point - в действительности тип структуры.
  Point p = new Point();
} // Здесь i и p покидают стек!
```

Типы значений, ссылочные типы и операция присваивания

Когда переменная одного типа значения присваивается переменной другого типа значения, выполняется почленное копирование полей данных. В случае простого типа данных, такого как `System.Int32`, единственным копируемым членом будет числовое значение. Однако для типа `Point` в новую переменную структуры будут копироваться значения полей `X` и `Y`. В целях демонстрации создадим новый проект консольного приложения по имени `ValueAndReferenceTypes` и скопируем предыдущее определение `Point` в новое пространство имен. После этого добавим к типу `Program` следующий метод:

```
// Присваивание двух внутренних типов значений дает
// в результате две независимые переменные в стеке.
static void ValueTypeAssignment()
{
  Console.WriteLine("Assigning value types\n");

  Point p1 = new Point(10, 10);
  Point p2 = p1;

  // Вывести значения обеих переменных Point.
  p1.Display();
  p2.Display();

  // Изменить p1.X и снова вывести значения переменных.
  // Значение p2.X не изменилось.
  p1.X = 100;
  Console.WriteLine("\n=> Changed p1.X\n");
  p1.Display();
  p2.Display();
}
```

Здесь создается переменная типа `Point` (`p1`), которая присваивается другой переменной типа `Point` (`p2`). Поскольку `Point` — тип значения, в стеке находятся две копии `Point`, каждой из которых можно манипулировать независимым образом. Поэтому при изменении значения `p1.X` значение `p2.X` остается незатронутым:

```
Assigning value types
X = 10, Y = 10
X = 10, Y = 10
=> Changed p1.X
X = 100, Y = 10
X = 10, Y = 10
```

По контрасту с типами значений, когда операция присваивания применяется к переменным ссылочных типов (т.е. экземплярам всех классов), происходит переадресация того, на что ссылочная переменная указывает в памяти. В целях иллюстрации создадим новый класс по имени `PointRef` с теми же членами, что и у структуры `Point`, но только переименуем конструктор в соответствии с именем этого класса:

```
// Классы всегда являются ссылочными типами.
class PointRef
{
  // Те же самые члены, что и в структуре Point...
  // Не забудьте изменить имя конструктора на PointRef!
  public PointRef(int XPos, int YPos)
  {
    X = XPos;
    Y = YPos;
  }
}
```

Теперь задействуем готовый тип `PointRef` в следующем новом методе. Обратите внимание, что помимо использования вместо структуры `Point` класса `PointRef` код идентичен коду метода `ValueTypeAssignment()`:

```
static void ReferenceTypeAssignment()
{
  Console.WriteLine("Assigning reference types\n");
  PointRef p1 = new PointRef(10, 10);
  PointRef p2 = p1;

  // Вывести значения обеих переменных PointRef.
  p1.Display();
  p2.Display();

  // Изменить p1.X и снова вывести значения.
  p1.X = 100;
  Console.WriteLine("\n=> Changed p1.X\n");
  p1.Display();
  p2.Display();
}
```

В данном случае есть две ссылки, указывающие на *один и тот же* объект в управляемой куче. Таким образом, когда изменяется значение `X` с использованием ссылки `p1`, изменится также и значение `p2.X`. Вот вывод, получаемый в результате вызова этого нового метода внутри `Main()`:

```
Assigning reference types
X = 10, Y = 10
X = 10, Y = 10
=> Changed p1.X
X = 100, Y = 10
X = 100, Y = 10
```

Типы значений, содержащие ссылочные типы

А теперь, когда вы лучше понимаете базовые отличия между типами значений и ссылочными типами, давайте обратимся к более сложному примеру. Предположим, что имеется следующий ссылочный тип (класс), который поддерживает информационную строку (`infoString`), устанавливаемую с применением специального конструктора:

```
class ShapeInfo
{
  public string infoString;
  public ShapeInfo(string info)
  {
    infoString = info;
  }
}
```

Далее представим, что переменная этого типа класса должна содержаться внутри типа значения по имени `Rectangle`. Кроме того, чтобы позволить вызывающему коду устанавливать значение внутренней переменной-члена типа `ShapeInfo`, также предусмотрен специальный конструктор. Вот полное определение типа `Rectangle`:

```
struct Rectangle
{
  // Структура Rectangle содержит член ссылочного типа.
  public ShapeInfo rectInfo;

  public int rectTop, rectLeft, rectBottom, rectRight;

  public Rectangle(string info, int top, int left, int bottom, int right)
  {
    rectInfo = new ShapeInfo(info);
    rectTop = top; rectBottom = bottom;
    rectLeft = left; rectRight = right;
  }
  public void Display()
  {
    Console.WriteLine("String = {0}, Top = {1}, Bottom = {2}, " +
      "Left = {3}, Right = {4}",
      rectInfo.infoString, rectTop, rectBottom, rectLeft, rectRight);
  }
}
```

На этой стадии ссылочный тип содержится внутри типа значения. Здесь возникает важный вопрос: что произойдет в результате присваивания одной переменной типа `Rectangle` другой переменной того же типа? Учитывая то, что уже известно о типах значений, можно корректно предположить, что целочисленные данные (которые на самом деле являются структурой — `System.Int32`) должны быть независимой сущностью для каждой переменной `Rectangle`. Но что можно сказать о внутреннем ссылочном типе? Будет ли полностью скопировано *состояние* этого объекта или же только ссылка на него? Чтобы получить ответ на данный вопрос, определим новый метод и вызовем его внутри `Main()`:

```
static void ValueTypeContainingRefType()
{
  // Создать первую переменную Rectangle.
  Console.WriteLine("-> Creating r1");
  Rectangle r1 = new Rectangle("First Rect", 10, 10, 50, 50);
```

```
  // Присвоить новой переменной Rectangle переменную r1.
  Console.WriteLine("-> Assigning r2 to r1");
  Rectangle r2 = r1;

  // Изменить некоторые значения в r2.
  Console.WriteLine("-> Changing values of r2");
  r2.rectInfo.infoString = "This is new info!";
  r2.rectBottom = 4444;

  // Вывести значения из обеих переменных Rectangle.
  r1.Display();
  r2.Display();
}
```

Вывод будет таким:

```
-> Creating r1
-> Assigning r2 to r1
-> Changing values of r2
String = This is new info!, Top = 10, Bottom = 50, Left = 10, Right = 50
String = This is new info!, Top = 10, Bottom = 4444, Left = 10, Right = 50
```

Как видите, когда мы модифицируем значение информационной строки с использованием ссылки `r2`, для ссылки `r1` отображается то же самое значение. По умолчанию, если тип значения содержит другие ссылочные типы, то присваивание приводит к копированию ссылок. В результате получаются две независимые структуры, каждая из которых содержит ссылку, указывающую на один и тот же объект в памяти (т.е. создается поверхностная копия). Для выполнения глубокого копирования, при котором в новый объект полностью копируется состояние внутренних ссылок, можно реализовать интерфейс `ICloneable` (как будет показано в главе 8).

Исходный код. Проект `ValueAndReferenceTypes` доступен в подкаталоге `Chapter_4`.

Передача ссылочных типов по значению

Вполне очевидно, что ссылочные типы и типы значений могут передаваться методам в виде параметров. Тем не менее, передача ссылочного типа (например, класса) по ссылке совершенно отличается от его передачи по значению. Чтобы понять разницу, предположим, что есть простой класс `Person`, определенный в новом проекте консольного приложения по имени `RefTypeValTypeParams`:

```
class Person
{
  public string personName;
  public int personAge;

  // Конструкторы.
  public Person(string name, int age)
  {
    personName = name;
    personAge = age;
  }
  public Person(){}

  public void Display()
  {
    Console.WriteLine("Name: {0}, Age: {1}", personName, personAge);
  }
}
```

А что если мы создадим метод, который позволит вызывающему коду передавать объект Person по значению (обратите внимание на отсутствие модификаторов параметров, таких как out или ref)?

```
static void SendAPersonByValue(Person p)
{
  // Изменить значение возраста в p?
  p.personAge = 99;

  // Увидит ли вызывающий код это изменение?
  p = new Person("Nikki", 99);
}
```

Здесь видно, что метод SendAPersonByValue() пытается присвоить входной ссылке на Person новый объект Person, а также изменить некоторые данные состояния. Протестируем этот метод, вызвав его внутри Main():

```
static void Main(string[] args)
{
  // Передача ссылочных типов по значению.
  Console.WriteLine("***** Passing Person object by value *****");
  Person fred = new Person("Fred", 12);
  Console.WriteLine("\nBefore by value call, Person is:");  // перед вызовом
  fred.Display();

  SendAPersonByValue(fred);
  Console.WriteLine("\nAfter by value call, Person is:");   // после вызова
  fred.Display();
  Console.ReadLine();
}
```

Ниже показан результирующий вывод:

```
***** Passing Person object by value *****

Before by value call, Person is:
Name: Fred, Age: 12

After by value call, Person is:
Name: Fred, Age: 99
```

Легко заметить, что значение PersoneAge было изменено. Кажется, что такое поведение противоречит смыслу передачи параметра "по значению". Учитывая, что попытка изменения состояния входного объекта Person прошла успешно, возникает вопрос: что же тогда было скопировано? Ответ: была получена копия ссылки на объект из вызывающего кода. Следовательно, т.к. метод SendAPersonByValue() указывает на тот же самый объект, что и вызывающий код, становится возможным изменение данных состояния этого объекта. Нельзя только переустанавливать ссылку так, чтобы она указывала на какой-то *другой* объект.

Передача ссылочных типов по ссылке

Теперь пусть имеется метод SendAPersonByReference(), в котором ссылочный тип передается по ссылке (обратите внимание на наличие модификатора параметра ref):

```
static void SendAPersonByReference(ref Person p)
{
  // Изменить некоторые данные в p.
  p.personAge = 555;
```

```
  // p теперь указывает на новый объект в куче!
  p = new Person("Nikki", 999);
}
```

Как и можно было ожидать, вызываемому коду предоставлена полная свобода в плане манипулирования входным параметром. Вызываемый код не только может изменять состояние объекта, но и переопределять ссылку так, чтобы она указывала на новый объект `Person`. Давайте протестируем метод `SendAPersonByReference()`, вызвав его в методе `Main()`:

```
static void Main(string[] args)
{
  // Передача ссылочных типов по ссылке.
  Console.WriteLine("***** Passing Person object by reference *****");
  ...
  Person mel = new Person("Mel", 23);
  Console.WriteLine("Before by ref call, Person is:");  // перед вызовом
  mel.Display();
  SendAPersonByReference(ref mel);
  Console.WriteLine("After by ref call, Person is:");   // после вызова
  mel.Display();
  Console.ReadLine();
}
```

Вывод выглядит следующим образом:

```
***** Passing Person object by reference *****
Before by ref call, Person is:
Name: Mel, Age: 23
After by ref call, Person is:
Name: Nikki, Age: 999
```

Здесь видно, что после вызова объект по имени `Mel` возвращается как объект по имени `Nikki`, поскольку метод имел возможность изменить то, на что указывала в памяти входная ссылка. Ниже представлены основные правила, которые необходимо соблюдать при передаче ссылочных типов.

- Если ссылочный тип передается по ссылке, то вызываемый код может изменять значения данных состояния объекта, а также объект, на который указывает ссылка.
- Если ссылочный тип передается по значению, то вызываемый код может изменять значения данных состояния объекта, но не объект, на который указывает ссылка.

Исходный код. Проект `RefTypeValTypeParams` доступен в подкаталоге `Chapter_4`.

Заключительные детали относительно типов значений и ссылочных типов

В завершение этой темы рассмотрим табл. 4.3 со сводкой по основным отличиям между типами значений и ссылочными типами.

Таблица 4.3. Отличия между типами значений и ссылочными типами

Интересующий вопрос	Тип значения	Ссылочный тип
Где размещены объекты?	Размещаются в стеке	Размещаются в управляемой куче
Как представлена переменная?	Переменные типов значений являются локальными копиями	Переменные ссылочных типов указывают на память, занимаемую размещенным экземпляром
Какой тип является базовым?	Неявно расширяет `System.ValueType`	Может быть производным от любого другого типа (кроме `System.ValueType`), если только этот тип не запечатан (см. главу 6)
Может ли этот тип выступать в качестве базового для других типов?	Нет. Типы значений всегда запечатаны, и наследовать от них нельзя	Да. Если тип не запечатан, то он может выступать в качестве базового для других типов
Каково стандартное поведение передачи параметров?	Переменные передаются по значению (т.е. вызываемой функции передается копия переменной)	Для ссылочных типов ссылка копируется по значению
Можно ли переопределять метод `System.Object.Finalize()` в этом типе?	Нет	Да, косвенно (как показано в главе 13)
Можно ли определять конструкторы для этого типа?	Да, но стандартный конструктор является зарезервированным (т.е. все специальные конструкторы должны иметь аргументы)	Безусловно!
Когда переменные этого типа прекращают свое существование?	Когда покидают область видимости, в которой они были определены	Когда объект подвергается сборке мусора

Несмотря на различия, типы значений и ссылочные типы имеют возможность реализовывать интерфейсы и могут поддерживать любое количество полей, методов, перегруженных операций, констант, свойств и событий.

Понятие типов C#, допускающих `null`

В заключение настоящей главы давайте исследуем роль *типов данных, допускающих значение* `null`, с применением консольного приложения по имени `NullableTypes`. Как вам уже известно, типы данных C# обладают фиксированным диапазоном значений и представлены в виде типов пространства имен `System`. Например, тип данных `System.Boolean` может принимать только значения из набора {`true`, `false`}. Вспомните, что все числовые типы данных (а также `Boolean`) являются *типами значений*. Типам значений никогда не может быть присвоено значение `null`, потому что оно служит для представления пустой объектной ссылки.

```
static void Main(string[] args)
{
  // Ошибка на этапе компиляции!
```

```
  // Типы значений не могут быть установлены в null!
  bool myBool = null;
  int myInt = null;
  // Все в порядке! Строки являются ссылочными типами.
  string myString = null;
}
```

Язык C# поддерживает концепцию *типов данных, допускающих значение* `null`. Говоря упрощенно, допускающий `null` тип может представлять все значения лежащего в основе типа плюс `null`. Таким образом, если вы объявите переменную типа `bool`, допускающего `null`, то ей можно будет присваивать значение из набора {`true`, `false`, `null`}. Это может быть чрезвычайно удобно при работе с реляционными базами данных, поскольку в таблицах баз данных довольно часто встречаются столбцы, для которых значения не определены. Без концепции типов данных, допускающих `null`, в C# не было бы удобного способа для представления числовых элементов данных без значений.

Чтобы определить переменную типа, допускающего `null`, необходимо добавить к имени интересующего типа данных префикс в виде знака вопроса (?). Обратите внимание, что такой синтаксис законен, только когда применяется к типам значений. При попытке создать ссылочный тип, допускающий `null` (включая `string`), компилятор сообщит об ошибке. Как и переменным с типами, не допускающими `null`, локальным переменным, которые имеют типы, допускающие `null`, должно быть присвоено начальное значение, прежде чем ими можно будет пользоваться:

```
static void LocalNullableVariables()
{
  // Определить несколько локальных переменных, допускающих null.
  int? nullableInt = 10;
  double? nullableDouble = 3.14;
  bool? nullableBool = null;
  char? nullableChar = 'a';
  int?[] arrayOfNullableInts = new int?[10];
  // Ошибка! Строки являются ссылочными типами!
  // string? s = "oops";
}
```

В языке C# система обозначений в форме суффикса ? представляет собой сокращение для создания экземпляра обобщенного типа структуры `System.Nullable<T>`. Хотя подробное исследование обобщений мы отложим до главы 9, сейчас важно понимать, что тип `System.Nullable<T>` предоставляет набор членов, которые могут применяться всеми типами, допускающими `null`.

Например, с помощью свойства `HasValue` или операции `!=` можно программно выяснять, действительно ли переменной, допускающей `null`, было присвоено значение `null`. Значение, которое присвоено типу, допускающему `null`, можно получать напрямую или через свойство `Value`. Учитывая, что суффикс ? является просто сокращением для использования `Nullable<T>`, метод `LocalNullableVariables()` можно было бы реализовать следующим образом:

```
static void LocalNullableVariablesUsingNullable()
{
  // Определить несколько типов, допускающих null, с применением Nullable<T>.
  Nullable<int> nullableInt = 10;
  Nullable<double> nullableDouble = 3.14;
  Nullable<bool> nullableBool = null;
  Nullable<char> nullableChar = 'a';
  Nullable<int>[] arrayOfNullableInts = new Nullable<int>[10];
}
```

Работа с типами, допускающими null

Как утверждалось ранее, типы данных, допускающие null, особенно полезны при взаимодействии с базами данных, потому что столбцы в таблицах данных могут быть намеренно оставлены пустыми (скажем, быть неопределенными). В целях демонстрации рассмотрим показанный далее класс, эмулирующий процесс доступа к базе данных с таблицей, в которой два столбца могут принимать значения null. Обратите внимание, что метод GetIntFromDatabase() не присваивает значение члену целочисленного типа, допускающего null, тогда как метод GetBoolFromDatabase() присваивает допустимое значение члену типа bool?.

```
class DatabaseReader
{
  // Поле данных типа, допускающего null.
  public int? numericValue = null;
  public bool? boolValue = true;

  // Обратите внимание на возвращаемый тип, допускающий null.
  public int? GetIntFromDatabase()
  { return numericValue; }

  // Обратите внимание на возвращаемый тип, допускающий null.
  public bool? GetBoolFromDatabase()
  { return boolValue; }
}
```

Теперь предположим, что в следующем методе Main() происходит обращение к каждому члену класса DatabaseReader и выяснение присвоенных значений с применением членов HasValue и Value, а также операции равенства C# (точнее, операции "не равно"):

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Nullable Data *****\n");
  DatabaseReader dr = new DatabaseReader();

  // Получить значение int из "базы данных".
  int? i = dr.GetIntFromDatabase();
  if (i.HasValue)
    Console.WriteLine("Value of 'i' is: {0}", i.Value);
                    // вывод значения переменной i
  else
    Console.WriteLine("Value of 'i' is undefined.");
                    // значение переменной i не определено

  // Получить значение bool из "базы данных".
  bool? b = dr.GetBoolFromDatabase();
  if (b != null)
    Console.WriteLine("Value of 'b' is: {0}", b.Value);
                    // вывод значения переменной b
  else
    Console.WriteLine("Value of 'b' is undefined.");
                    // значение переменной b не определено
  Console.ReadLine();
}
```

Операция объединения с null

Следующий важный аспект связан с тем, что любая переменная, которая может иметь значение null (т.е. переменная ссылочного типа или переменная типа, допускающего null), может использоваться с операцией ?? языка C#, формально называемой *операцией объединения с null*. Эта операция позволяет присваивать значение типу, допускающему null, если извлеченное значение на самом деле равно null. В рассматриваемом примере мы предположим, что в случае возвращения методом GetIntFromDatabase() значения null (данный метод действительно запрограммирован так, что *всегда* возвращает null, но вы должны уловить общую идею) локальной переменной целочисленного типа, допускающего null, необходимо присвоить значение 100:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Nullable Data *****\n");
  DatabaseReader dr = new DatabaseReader();
  ...
  // Если значение, возвращаемое из GetIntFromDatabase(),
  // равно null, то присвоить локальной переменной значение 100.
  int myData = dr.GetIntFromDatabase() ?? 100;
  Console.WriteLine("Value of myData: {0}", myData);
  Console.ReadLine();
}
```

Преимущество применения операции ?? заключается в том, что она дает более компактную версию кода, чем традиционный условный оператор if/else. Однако при желании можно было бы написать показанный ниже функционально эквивалентный код, который в случае возвращения null обеспечит установку переменной в значение 100:

```
// Более длинный код, в котором не используется синтаксис операции ??.
int? moreData = dr.GetIntFromDatabase();
if (!moreData.HasValue)
   moreData = 100;
Console.WriteLine("Value of moreData: {0}", moreData);
```

null-условная операция

При разработке программного обеспечения распространенной практикой является проверка на предмет null входных параметров, которым передаются значения, возвращаемые членами типов (методами, свойствами, индексаторами). Например, пусть имеется метод, который принимает в качестве единственного параметра строковый массив. В целях безопасности его желательно проверять на предмет null, прежде чем приступать к обработке. Поступая подобным образом, вы не получите ошибку во время выполнения, если массив окажется пустым. Следующий код демонстрирует традиционный способ реализации такой проверки:

```
static void TesterMethod(string[] args)
{
  // Перед доступом к данным массива мы должны проверить его на равенство null!
  if (args != null)
  {
    Console.WriteLine($"You sent me {args.Length} arguments.");
  }
}
```

Чтобы устранить обращение к свойству Length массива string в случае, когда он равен null, здесь используется условный оператор. Если вызывающий код не создаст массив данных и вызовет метод TesterMethod() примерно так, как показано ниже, то никаких ошибок во время выполнения не возникнет:

```
TesterMethod(null);
```

В текущей версии языка C# появилась возможность задействовать *null-условную операцию* (знак вопроса, помещенный после типа переменной, но перед операцией доступа к члену), которая позволяет упростить представленную ранее проверку на предмет null. Вместо явного условного оператора, проверяющего на неравенство значению null, теперь можно написать такой код:

```
static void TesterMethod(string[] args)
{
  // Мы должны проверять на предмет null перед доступом к данным массива!
  Console.WriteLine($"You sent me {args?.Length} arguments.");
}
```

В этом случае условный оператор не применяется. Взамен к переменной массива string в качестве суффикса добавляется операция ?. Если args окажется null, то обращение к свойству Length не приведет к ошибке во время выполнения. Чтобы вывести действительное значение, можно было бы воспользоваться операцией объединения с null и установить стандартное значение:

```
Console.WriteLine($"You sent me {args?.Length ?? 0} arguments.");
```

Существуют и дополнительные области кодирования, в которых новая null-условная операция C# 6.0 будет очень удобна, особенно при работе с делегатами и событиями. Тем не менее, данные темы рассматриваются позже, в главе 10, где вы найдете соответствующие сценарии применения. На этом первоначальное знакомство с языком программирования C# завершено. В главе 5 вы начнете погружаться в детали объектно-ориентированной разработки.

Исходный код. Проект NullableTypes доступен в подкаталоге Chapter_4.

Резюме

Эта глава начиналась с исследования нескольких ключевых слов C#, которые позволяют строить специальные методы. Вспомните, что по умолчанию параметры передаются по значению, однако параметры можно передавать и по ссылке, если пометить их модификаторами ref или out. Кроме того, вы узнали о роли необязательных и именованных параметров, а также о том, как определять и вызывать методы, принимающие массивы параметров.

После рассмотрения темы перегрузки методов в главе приводились подробные сведения, касающиеся способов определения массивов, перечислений и структур в C# и их представления в библиотеках базовых классов .NET. По ходу дела обсуждались основные характеристики типов значений и ссылочных типов, включая их поведение при передаче в качестве параметров методам, и способы взаимодействия с типами данных, допускающими null, и переменными, которые могут иметь значение null (например, переменными ссылочных типов и переменными типов значений, допускающих null), с использованием операций ? и ??.

ЧАСТЬ III

Объектно-ориентированное программирование на C#

В этой части

ГЛАВА 5

Инкапсуляция

В главах 3 и 4 мы исследовали несколько основных синтаксических конструкций, присущих любому приложению .NET, которое вам доведется разрабатывать. Начиная с данной главы, мы приступаем к изучению объектно-ориентированных возможностей C#. Первое, что вам предстоит узнать — процесс построения четко определенных типов классов, которые поддерживают любое количество *конструкторов*. После ознакомления с основами определения классов и размещения объектов мы посвятим остаток главы теме *инкапсуляции*. В ходе изложения вы узнаете, как определять свойства классов, а также получите подробные сведения о ключевом слове `static`, синтаксисе инициализации объектов, полях только для чтения, константных данных и частичных классах.

Знакомство с типом класса C#

С точки зрения платформы .NET наиболее фундаментальной программной конструкцией является *тип класса*. Формально класс — это определяемый пользователем тип, состоящий из полей данных (часто называемых *переменными-членами*) и членов, которые оперируют этими данными (к ним относятся конструкторы, свойства, методы, события и т.д.). Коллективно набор полей данных представляет "состояние" экземпляра класса (по-другому называемого *объектом*). Мощь объектно-ориентированных языков, таких как C#, заключается в том, что за счет группирования данных и связанной с ними функциональности в унифицированное определение класса вы получаете возможность моделировать свое программное обеспечение в соответствии с сущностями реального мира.

Для начала создадим новый проект консольного приложения C# по имени `SimpleClassExample`. Затем добавим в проект новый файл класса (`Car.cs`), используя пункт меню Project⇨Add Class (Проект⇨Добавить класс). В открывшемся диалоговом окне понадобится выбрать значок Class (Класс), как показано на рис. 5.1, и щелкнуть на кнопке Add (Добавить).

Класс определяется в C# с применением ключевого слова `class`. Вот как выглядит простейшее из возможных объявление класса:

```
class Car
{
}
```

После определения типа класса необходимо определить набор переменных-членов, которые будут использоваться для представления его состояния. Например, вы можете решить, что объекты `Car` (автомобили) должны иметь поле данных типа `int`, представляющее текущую скорость, и поле данных типа `string` для представления дружественного названия автомобиля.

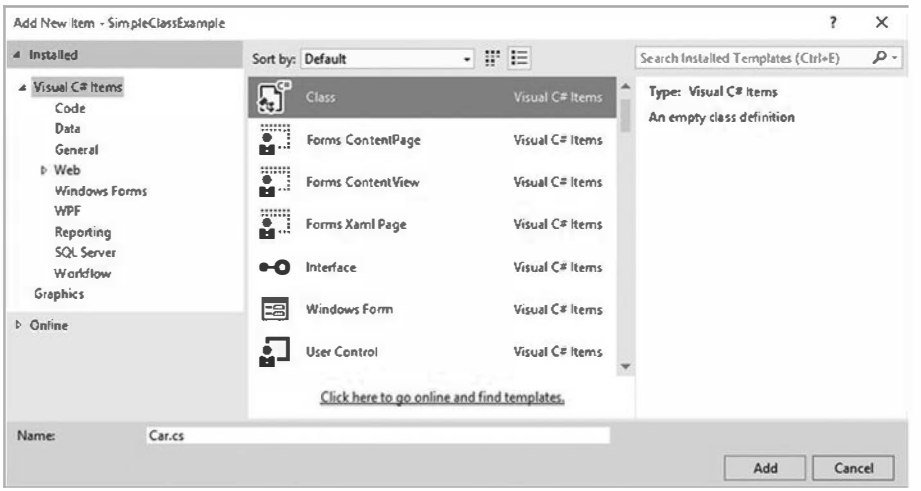


Рис. 5.1. Добавление нового типа класса C#

С учетом этих начальных проектных положений класс `Car` будет выглядеть следующим образом:

```
class Car
{
  // 'Состояние' объекта Car.
  public string petName;
  public int currSpeed;
}
```

Обратите внимание, что переменные-члены объявлены с применением модификатора доступа `public`. Открытые (`public`) члены класса доступны напрямую после того, как создан *объект* этого типа. Вспомните, что термин *объект* используется для описания экземпляра заданного типа класса, который создан с помощью ключевого слова `new`.

На заметку! Поля данных класса редко (если вообще когда-нибудь) должны определяться как открытые. Чтобы обеспечить целостность данных состояния, намного лучше объявлять данные закрытыми (`private`) или, возможно, защищенными (`protected`) и разрешать контролируемый доступ к данным через свойства (как будет показано далее в главе). Однако для максимального упрощения первого примера мы определили поля данных как открытые.

После определения набора переменных-членов, представляющих состояние класса, следующим шагом в проектировании будет установка членов, которые моделируют его поведение. Для этого примера в классе `Car` определены методы по имени `SpeedUp()` и `PrintState()`. Модифицируйте код класса, как показано ниже:

```
class Car
{
  // 'Состояние' объекта Car.
  public string petName;
  public int currSpeed;
  // Функциональность Car.
  public void PrintState()
  {
    Console.WriteLine("{0} is going {1} MPH.", petName, currSpeed);
  }
```

```
  public void SpeedUp(int delta)
  {
    currSpeed += delta;
  }
}
```

Метод `PrintState()` — это простая диагностическая функция, которая выводит текущее состояние объекта `Car` в окно командной строки. Метод `SpeedUp()` увеличивает скорость автомобиля, представляемого объектом `Car`, на величину, которая передается во входном параметре типа `int`. Теперь обновите код метода `Main()` следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");

  // Разместить в памяти и сконфигурировать объект Car.
  Car myCar = new Car();
  myCar.petName = "Henry";
  myCar.currSpeed = 10;

  // Увеличить скорость автомобиля в несколько раз и вывести новое состояние.
  for (int i = 0; i <= 10; i++)
  {
    myCar.SpeedUp(5);
    myCar.PrintState();
  }
  Console.ReadLine();
}
```

Запустив программу, вы увидите, что переменная `Car` (`myCar`) поддерживает свое текущее состояние на протяжении жизни приложения:

```
***** Fun with Class Types *****

Henry is going 15 MPH.
Henry is going 20 MPH.
Henry is going 25 MPH.
Henry is going 30 MPH.
Henry is going 35 MPH.
Henry is going 40 MPH.
Henry is going 45 MPH.
Henry is going 50 MPH.
Henry is going 55 MPH.
Henry is going 60 MPH.
Henry is going 65 MPH.
```

Размещение объектов с помощью ключевого слова `new`

Как было показано в предыдущем примере кода, объекты должны быть размещены в памяти с применением ключевого слова `new`. Если вы не укажете ключевое слово `new` и попытаетесь использовать переменную класса в последующем операторе кода, то получите ошибку на этапе компиляции. Например, приведенный ниже метод `Main()` не скомпилируется:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");
```

```
  // Ошибка на этапе компиляции! Забыли использовать new для создания объекта!
  Car myCar;
  myCar.petName = "Fred";
}
```

Чтобы корректно создать объект с применением ключевого слова `new`, определить и разместить в памяти объект `Car` можно в одной строке кода:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");
  Car myCar = new Car();
  myCar.petName = "Fred";
}
```

В качестве альтернативы определение и размещение в памяти экземпляра класса может осуществляться в отдельных строках кода:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");
  Car myCar;
  myCar = new Car();
  myCar.petName = "Fred";
}
```

Здесь первый оператор кода просто объявляет *ссылку* на еще не созданный объект типа `Car`. После явного присваивания ссылка будет указывать на действительный объект в памяти. В любом случае к этому моменту мы имеем простейший класс, в котором определено несколько элементов данных и ряд базовых операций. Чтобы расширить функциональность текущего класса `Car`, необходимо разобраться с ролью *конструкторов*.

Понятие конструкторов

Учитывая наличие у объекта состояния (представленного значениями его переменных-членов), обычно желательно присвоить подходящие значения полям объекта перед тем, как работать с ним. В настоящее время класс `Car` требует присваивания значений полям `petName` и `currSpeed` по отдельности. Для текущего примера это не слишком проблематично, поскольку открытых элементов данных всего два. Тем не менее, зачастую класс содержит несколько десятков полей, с которыми надо что-то делать. Ясно, что было бы нежелательно писать 20 операторов инициализации для всех 20 элементов данных.

К счастью, язык C# поддерживает использование *конструкторов*, которые позволяют устанавливать состояние объекта в момент его создания. Конструктор — это специальный метод класса, который вызывается неявно при создании объекта с применением ключевого слова `new`. Однако в отличие от "нормального" метода конструктор никогда не имеет возвращаемого значения (даже `void`) и всегда именуется идентично имени класса, объекты которого он конструирует.

Роль стандартного конструктора

Каждый класс C# снабжается "бесплатным" *стандартным конструктором*, который в случае необходимости может быть переопределен. По определению стандартный конструктор никогда не принимает аргументов. После размещения нового объекта в памяти стандартный конструктор гарантирует установку всех полей данных в соответствующие стандартные значения (стандартные значения для типов данных C# были описаны в главе 3).

Если вас не удовлетворяют такие стандартные присваивания, то можете переопределить стандартный конструктор в соответствии со своими нуждами. В целях иллюстрации модифицируем класс C# следующим образом:

```
class Car
{
  // 'Состояние' объекта Car.
  public string petName;
  public int currSpeed;

  // Специальный стандартный конструктор.
  public Car()
  {
    petName = "Chuck";
    currSpeed = 10;
  }
  ...
}
```

В данном случае мы заставляем объекты `Car` начинать свое существование под именем `Chuck` и со скоростью 10 миль в час. Создать объект `Car` со стандартными значениями можно так:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");

  // Вызов стандартного конструктора.
  Car chuck = new Car();

  // Выводит строку "Chuck is going 10 MPH."
  chuck.PrintState();
  ...
}
```

Определение специальных конструкторов

Обычно помимо стандартного конструктора в классах определяются дополнительные конструкторы. Тем самым пользователю объекта предоставляется простой и согласованный способ инициализации состояния объекта прямо во время его создания. Взгляните на следующее изменение класса `Car`, который теперь поддерживает целых три конструктора:

```
class Car
{
  // 'Состояние' объекта Car.
  public string petName;
  public int currSpeed;

  // Специальный стандартный конструктор.
  public Car()
  {
    petName = "Chuck";
    currSpeed = 10;
  }

  // Здесь currSpeed получает стандартное значение для типа int (0).
  public Car(string pn)
  {
    petName = pn;
  }
```

```
  // Позволяет вызывающему коду установить полное состояние объекта Car.
  public Car(string pn, int cs)
  {
    petName = pn;
    currSpeed = cs;
  }
  ...
}
```

Имейте в виду, что один конструктор отличается от другого (с точки зрения компилятора C#) числом и/или типами аргументов. Вспомните из главы 4, что определение метода с тем же самым именем, но разным количеством или типами аргументов, называется *перегрузкой* метода. Таким образом, конструктор класса `Car` перегружен, чтобы предложить несколько способов создания объекта во время объявления. В любом случае теперь есть возможность создавать объекты `Car`, используя любой из его открытых конструкторов. Вот пример:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");

  // Создать объект Car по имени Chuck со скоростью 10 миль в час.
  Car chuck = new Car();
  chuck.PrintState();

  // Создать объект Car по имени Mary со скоростью 0 миль в час.
  Car mary = new Car("Mary");
  mary.PrintState();

  // Создать объект Car по имени Daisy со скоростью 75 миль в час.
  Car daisy = new Car("Daisy", 75);
  daisy.PrintState();
  ...
}
```

Еще раз о стандартном конструкторе

Как вы только что узнали, все классы снабжаются стандартным конструктором. Следовательно, если добавить в текущий проект новый класс по имени `Motorcycle`, определенный примерно так:

```
class Motorcycle
{
  public void PopAWheely()
  {
    Console.WriteLine("Yeeeeeee Haaaaaeewww!");
  }
}
```

то сразу можно будет создавать экземпляры `Motorcycle` с помощью стандартного конструктора:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");
  Motorcycle mc = new Motorcycle();
  mc.PopAWheely();
  ...
}
```

Тем не менее, как только определен специальный конструктор с любым числом параметров, стандартный конструктор молча удаляется из класса и перестает быть доступным. Воспринимайте это так: если вы не определили специальный конструктор, то компилятор C# снабжает класс стандартным конструктором, давая возможность пользователю размещать в памяти экземпляр вашего класса с набором полей данных, которые установлены в корректные стандартные значения. Однако когда вы определяете уникальный конструктор, компилятор предполагает, что вы решили взять власть в свои руки.

Поэтому если вы хотите позволить пользователю создавать экземпляр вашего типа с помощью стандартного конструктора, а также специального конструктора, то должны *явно* переопределить стандартный конструктор. Важно понимать, что в подавляющем большинстве случаев реализация стандартного конструктора класса намеренно оставляется пустой, т.к. все, что требуется — это создание объекта со стандартными значениями. Внесем в класс `Motorcycle` следующие изменения:

```
class Motorcycle
{
  public int driverIntensity;

  public void PopAWheely()
  {
    for (int i = 0; i <= driverIntensity; i++)
    {
      Console.WriteLine("Yeeeeeee Haaaaaeewww!");
    }
  }

  // Вернуть стандартный конструктор, который будет
  // устанавливать все члены данных в стандартные значения.
  public Motorcycle() {}

  // Специальный конструктор.
  public Motorcycle(int intensity)
  {
    driverIntensity = intensity;
  }
}
```

На заметку! Теперь, когда вы лучше понимаете роль конструкторов класса, полезно узнать об одном удобном сокращении. В IDE-среде Visual Studio предлагается фрагмент кода `ctor`. Если вы наберете `ctor` и два раза нажмете клавишу <Tab>, то IDE-среда автоматически определит специальный стандартный конструктор. Затем можно добавить нужные параметры и логику реализации. Испытайте такой прием.

Роль ключевого слова `this`

Язык C# предлагает ключевое слово `this`, которое обеспечивает доступ к текущему экземпляру класса. Один из возможных сценариев применения ключевого слова `this` предусматривает разрешение неоднозначности с областью видимости, которая может возникнуть, когда входной параметр имеет такое же имя, как и поле данных класса. Разумеется, вы могли бы просто придерживаться соглашения об именовании, которое не приводит к такой неоднозначности; тем не менее, чтобы проиллюстрировать такой сценарий, добавим в класс `Motorcycle` новое поле типа `string` (по имени `name`), предназначенное для представления имени водителя. Затем добавим метод по имени `SetDriverName()` с приведенной ниже реализацией:

```
class Motorcycle
{
  public int driverIntensity;

  // Новые члены для представления имени водителя.
  public string name;
  public void SetDriverName(string name)
  {
    name = name;
  }
  ...
}
```

Хотя данный код нормально скомпилируется, Visual Studio отобразит сообщение с предупреждением о том, что переменная присваивается сама себе! Чтобы удостовериться в этом, добавим в метод `Main()` вызов `SetDriverName()` и выведем значение поля `name`. Вы можете быть удивлены, обнаружив, что значением поля `name` является пустая строка.

```
// Создать объект Motorcycle с мотоциклистом по имени Tiny?
Motorcycle c = new Motorcycle(5);
c.SetDriverName("Tiny");
c.PopAWheely();
Console.WriteLine("Rider name is {0}", c.name); // Выводит пустое значение name!
```

Проблема в том, что реализация метода `SetDriverName()` присваивает входному параметру значение *его самого*, т.к. компилятор предполагает, что `name` ссылается на переменную, находящуюся в области действия метода, а не на поле `name` из области видимости класса. Для информирования компилятора о том, что необходимо установить поле данных `name` текущего объекта в значение входного параметра `name`, просто используйте ключевое слово `this`, чтобы разрешить такую неоднозначность:

```
public void SetDriverName(string name)
{
  this.name = name;
}
```

Имейте в виду, что если неоднозначность отсутствует, то применять ключевое слово `this`, когда классу нужно обращаться к собственным данным или членам, вовсе не обязательно. Например, если вы переименуете член данных типа `string` с `name` на `driverName` (что также повлечет за собой модификацию кода в методе `Main()`), то потребность в использовании `this` отпадет, поскольку неоднозначности с областью видимости больше нет:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  public void SetDriverName(string name)
  {
    // Эти два оператора функционально эквивалентны.
    driverName = name;
    this.driverName = name;
  }
  ...
}
```

Хотя применение `this` в неоднозначных ситуациях дает не особенно большой выигрыш, вы можете счесть это ключевое слово удобным при реализации членов класса, т.к. IDE-среды, подобные Visual Studio, будут активизировать средство IntelliSense, когда присутствует `this`. Это может оказаться полезным, если вы забыли имя члена класса и хотите быстро вспомнить его определение. Взгляните на рис. 5.2.

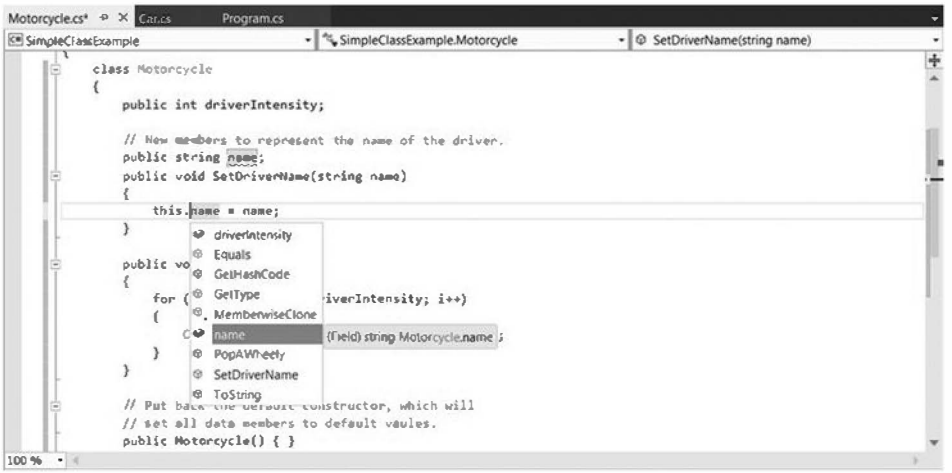


Рис. 5.2. Активизация средства IntelliSense для `this`

Построение цепочки вызовов конструкторов с использованием `this`

Еще один сценарий применения ключевого слова `this` касается проектирования класса с использованием приема, который называется *построением цепочки конструкторов*. Такой шаблон проектирования полезен при наличии класса, определяющего множество конструкторов. Учитывая тот факт, что конструкторы часто проверяют входные аргументы на соблюдение разнообразных бизнес-правил, довольно часто внутри набора конструкторов обнаруживается избыточная логика проверки достоверности. Рассмотрим следующее измененное определение класса `Motorcycle`:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;
  public Motorcycle() { }
  // Избыточная логика конструктора!
  public Motorcycle(int intensity)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
  }
  public Motorcycle(int intensity, string name)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
```

```
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

Здесь (возможно, как попытка обеспечить безопасность мотоциклиста) в каждом конструкторе производится проверка того, что уровень мощности не превышает 10. Наряду с тем, что это правильно, в двух конструкторах присутствует избыточный код. Подход далек от идеала, поскольку в случае изменения правил (например, если уровень мощности не должен превышать 5, а не 10) код придется модифицировать в нескольких местах.

Один из способов улучшить создавшуюся ситуацию предусматривает определение в классе `Motorcycle` метода, который будет выполнять проверку входных аргументов. Если вы решите поступить так, то каждый конструктор сможет вызывать этот метод перед присваиванием значений полям. Хотя такой подход позволяет изолировать код, который придется обновлять при изменении бизнес-правил, теперь имеется другая избыточность:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  // Конструкторы.
  public Motorcycle() { }

  public Motorcycle(int intensity)
  {
    SetIntensity(intensity);
  }
  public Motorcycle(int intensity, string name)
  {
    SetIntensity(intensity);
    driverName = name;
  }
  public void SetIntensity(int intensity)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
  }
  ...
}
```

Более совершенный подход предполагает назначение конструктора, который принимает *наибольшее число аргументов*, в качестве "главного конструктора", и выполнение требуемой логики проверки достоверности внутри его реализации. Остальные конструкторы могут применять ключевое слово `this` для передачи входных аргументов главному конструктору и при необходимости предоставлять любые дополнительные параметры. В таком случае вам придется беспокоиться только о поддержке единственного конструктора для всего класса, в то время как оставшиеся конструкторы будут в основном пустыми.

Ниже показана финальная реализация класса `Motorcycle` (с одним дополнительным конструктором в целях иллюстрации). При связывании конструкторов в цепочку обра-

тите внимание, что ключевое слово this располагается за пределами самого конструктора и отделяется от его объявления двоеточием:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  // Связывание конструкторов в цепочку.
  public Motorcycle() {}
  public Motorcycle(int intensity)
    : this(intensity, "") {}
  public Motorcycle(string name)
    : this(0, name) {}
  // Это 'главный' конструктор, выполняющий всю реальную работу.
  public Motorcycle(int intensity, string name)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

Имейте в виду, что использовать ключевое слово this для связывания вызовов конструкторов в цепочку вовсе не обязательно. Однако такой подход позволяет получить лучше сопровождаемое и более краткое определение класса. Применяя этот прием, также можно упростить решение задач программирования, потому что реальная работа делегируется единственному конструктору (обычно принимающему большую часть параметров), в то время как остальные просто "перекладывают на него ответственность".

На заметку! Вспомните из главы 4, что в языке C# поддерживаются необязательные параметры. Если вы будете использовать в конструкторах своих классов необязательные параметры, то сможете добиться тех же преимуществ, что и при связывании конструкторов в цепочку, но со значительно меньшим объемом кода. Вскоре вы увидите, как это делается.

Изучение потока управления конструкторов

Напоследок отметим, что как только конструктор передал аргументы выделенному главному конструктору (и этот конструктор обработал данные), первоначально вызванный конструктор продолжит выполнение всех оставшихся операторов кода. В целях прояснения модифицируем конструкторы класса Motorcycle, добавив в них вызов метода Console.WriteLine():

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;
  // Связывание конструкторов в цепочку.
  public Motorcycle()
  {
    Console.WriteLine("In default ctor"); // Внутри стандартного конструктора
  }
```

```
  public Motorcycle(int intensity)
    : this(intensity, "")
  {
    Console.WriteLine("In ctor taking an int");
                    // Внутри конструктора, принимающего int
  }
  public Motorcycle(string name)
    : this(0, name)
  {
    Console.WriteLine("In ctor taking a string");
                    // Внутри конструктора, принимающего string
  }
  // Это 'главный' конструктор, выполняющий всю реальную работу.
  public Motorcycle(int intensity, string name)
  {
    Console.WriteLine("In master ctor "); // Внутри главного конструктора
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

Теперь изменим метод `Main()`, чтобы он работал с объектом `Motorcycle` следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with class Types *****\n");
  // Создать объект Motorcycle.
  Motorcycle c = new Motorcycle(5);
  c.SetDriverName("Tiny");
  c.PopAWheely();
  Console.WriteLine("Rider name is {0}", c.driverName); // вывод имени гонщика
  Console.ReadLine();
}
```

Взгляните на вывод, полученный из показанного выше метода `Main()`:

```
***** Fun with class Types *****
In master ctor
In ctor taking an int
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Rider name is Tiny
```

Ниже описан поток логики конструкторов.

- Прежде всего, создается объект путем вызова конструктора, принимающего один аргумент типа `int`.
- Этот конструктор передает полученные данные главному конструктору и предоставляет любые дополнительные начальные аргументы, не указанные вызывающим кодом.

- Главный конструктор присваивает входные данные полям данных объекта.
- Управление возвращается первоначально вызванному конструктору, который выполняет оставшиеся операторы кода.

В построении цепочек конструкторов примечательно то, что этот шаблон программирования будет работать с любой версией языка C# и платформой .NET. Тем не менее, если целевой платформой является .NET 4.0 или последующая версия, то решение задач можно упростить еще больше, применяя необязательные аргументы в качестве альтернативы построению традиционных цепочек конструкторов.

Еще раз о необязательных аргументах

В главе 4 вы изучили необязательные и именованные аргументы. Вспомните, что необязательные аргументы позволяют определять стандартные значения для входных аргументов. Если вызывающий код удовлетворяют эти стандартные значения, то указывать уникальные значения не обязательно, но это нужно делать, чтобы снабдить объект специальными данными. Рассмотрим следующую версию класса `Motorcycle`, которая теперь предлагает несколько возможностей конструирования объектов, используя *единственное* определение конструктора:

```
class Motorcycle
{
  // Единственный конструктор, использующий необязательные аргументы.
  public Motorcycle(int intensity = 0, string name = "")
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

С помощью этого единственного конструктора можно создавать объект `Motorcycle`, указывая ноль, один или два аргумента. Вспомните, что синтаксис именованных аргументов по существу позволяет пропускать приемлемые стандартные установки (см. главу 4).

```
static void MakeSomeBikes()
{
  // driverName = "", driverIntensity = 0
  Motorcycle m1 = new Motorcycle();
  Console.WriteLine("Name= {0}, Intensity= {1}",
    m1.driverName, m1.driverIntensity);

  // driverName = "Tiny", driverIntensity = 0
  Motorcycle m2 = new Motorcycle(name:"Tiny");
  Console.WriteLine("Name= {0}, Intensity= {1}",
    m2.driverName, m2.driverIntensity);

  // driverName = "", driverIntensity = 7
  Motorcycle m3 = new Motorcycle(7);
  Console.WriteLine("Name= {0}, Intensity= {1}",
    m3.driverName, m3.driverIntensity);
}
```

В любом случае к этому моменту вы способны определить класс с полями данными (т.е. переменными-членами) и разнообразными операциями, такими как методы и конструкторы. Теперь давайте формализуем роль ключевого слова `static`.

Исходный код. Проект `SimpleClassExample` доступен в подкаталоге `Chapter_5`.

Понятие ключевого слова `static`

Класс C# может определять любое количество *статических членов*, которые объявляются с применением ключевого слова `static`. В таком случае интересующий член должен вызываться прямо на уровне класса, а не через переменную со ссылкой на объект. Чтобы проиллюстрировать разницу, обратимся к нашему старому знакомому `System.Console`. Как вы уже видели, метод `WriteLine()` не вызывается на уровне объекта:

```
// Ошибка на этапе компиляции! WriteLine() - не метод уровня объекта!
Console c = new Console();
c.WriteLine("I can't be printed...");
```

Вместо этого статический член `WriteLine()` предваряется именем класса:

```
// Правильно! WriteLine() - статический метод.
Console.WriteLine("Much better! Thanks...");
```

Попросту говоря, статические члены — это элементы, которые проектировщик класса посчитал настолько общими, что перед обращением к ним даже нет нужды создавать экземпляр класса. Наряду с тем, что определять статические члены можно в любом классе, чаще всего они обнаруживаются внутри *обслуживающих классов*. По определению обслуживающий класс представляет собой такой класс, который не поддерживает какое-либо состояние на уровне объектов и не предполагает создание своих экземпляров с помощью ключевого слова `new`. Взамен обслуживающий класс открывает доступ ко всей функциональности посредством членов уровня класса (также известных под названием статических).

Например, если бы вы воспользовались браузером объектов Visual Studio (выбрав пункт меню View⇨Object Browser (Вид⇨Браузер объектов)) для просмотра пространства имен `System` из сборки `mscorlib.dll`, то увидели бы, что все члены классов `Console`, `Math`, `Environment` и `GC` (среди прочих) открывают доступ к своей функциональности через статические члены. Они являются лишь несколькими обслуживающими классами, которые можно найти в библиотеках базовых классов .NET.

И снова следует отметить, что статические члены находятся не только в обслуживающих классах; они могут быть частью в принципе любого определения класса. Просто запомните, что статические члены продвигают заданный элемент на уровень класса вместо уровня объектов. Как будет показано в нескольких последующих разделах, ключевое слово `static` может применяться к перечисленным ниже конструкциям:

- данные класса;
- методы класса;
- свойства класса;
- конструктор;
- полное определение класса;
- в сочетании с ключевым словом `using`.

Давайте рассмотрим все варианты, начав с концепции статических данных.

На заметку! Роль статических свойств будет объясняться позже в этой главе во время исследования самих свойств.

Определение статических полей данных

При проектировании класса в большинстве случаев данные определяются на уровне экземпляра — другими словами, как нестатические данные. Когда определяются данные уровня экземпляра, то известно, что каждый создаваемый новый объект поддерживает собственную независимую копию этих данных. По контрасту при определении *статических* данных класса выделенная под них память разделяется всеми объектами этой категории.

Чтобы увидеть разницу, создадим новый проект консольного приложения под названием `StaticDataAndMembers` и добавим в него новый класс по имени `SavingsAccount`. Начнем с определения элемента данных уровня экземпляра (для моделирования текущего баланса) и специального конструктора для установки начального баланса:

```
// Простой класс депозитного счета.
class SavingsAccount
{
  // Данные уровня экземпляра.
  public double currBalance;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }
}
```

При создании объектов `SavingsAccount` память под поле `currBalance` выделяется для каждого объекта. Таким образом, можно было бы создать пять разных объектов `SavingsAccount`, каждый с собственным уникальным балансом. Более того, в случае изменения баланса в одном объекте счета другие объекты не затрагиваются.

С другой стороны, память для статических данных распределяется однажды и разделяется всеми объектами того же самого класса. Добавьте в класс `SavingsAccount` статический элемент данных по имени `currInterestRate`, который устанавливается в стандартное значение `0.04`:

```
// Простой класс депозитного счета.
class SavingsAccount
{
  // Данные уровня экземпляра.
  public double currBalance;

  // Статический элемент данных.
  public static double currInterestRate = 0.04;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }
}
```

После создания трех экземпляров класса `SavingsAccount`, как показано ниже, размещение данных в памяти будет выглядеть примерно так, как иллюстрируется на рис. 5.3:

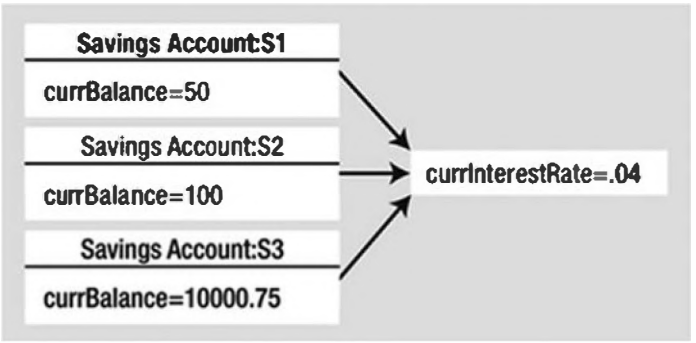


Рис. 5.3. Память под статические данные распределяется один раз и разделяется между всеми экземплярами класса

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Static Data *****\n");
  SavingsAccount s1 = new SavingsAccount(50);
  SavingsAccount s2 = new SavingsAccount(100);
  SavingsAccount s3 = new SavingsAccount(10000.75);
  Console.ReadLine();
}
```

Здесь предполагается, что все депозитные счета должны иметь одну и ту же процентную ставку. Поскольку статические данные разделяются всеми объектами той же самой категории, если вы измените процентную ставку каким-либо образом, то все объекты будут "видеть" новое значение при следующем доступе к статическим данным, т.к. все они по существу просматривают одну и ту же ячейку памяти. Чтобы понять, как изменять (или получать) статические данные, понадобится рассмотреть роль статических методов.

Определение статических методов

Давайте модифицируем класс `SavingsAccount`, определив два статических метода. Первый статический метод (`GetInterestRate()`) будет возвращать текущую процентную ставку, а второй (`SetInterestRate()`) позволит изменять эту процентную ставку:

```
// Простой класс депозитного счета.
class SavingsAccount
{
  // Данные уровня экземпляра.
  public double currBalance;
  // Статический элемент данных.
  public static double currInterestRate = 0.04;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }

  // Статические члены для установки/получения процентной ставки.
  public static void SetInterestRate(double newRate)
  { currInterestRate = newRate; }
  public static double GetInterestRate()
  { return currInterestRate; }
}
```

Рассмотрим показанный ниже сценарий использования класса:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Static Data *****\n");
  SavingsAccount s1 = new SavingsAccount(50);
  SavingsAccount s2 = new SavingsAccount(100);

  // Вывести текущую процентную ставку.
  Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());

  // Создать новый объект; это не 'сбросит' процентную ставку.
  SavingsAccount s3 = new SavingsAccount(10000.75);
  Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());

  Console.ReadLine();
}
```

Вывод предыдущего метода `Main()` выглядит так:

```
***** Fun with Static Data *****

Interest Rate is: 0.04
Interest Rate is: 0.04
```

Как видите, при создании новых экземпляров класса `SavingsAccount` значение статических данных не сбрасывается, поскольку среда CLR выделяет для них место в памяти только один раз. После этого все объекты типа `SavingsAccount` имеют дело с одним и тем же значением в статическом поле `currInterestRate`.

При проектировании любого класса C# одна из задач связана с выяснением того, какие порции данных должны быть определены как статические члены, а какие — нет. Хотя строгих правил не существует, запомните, что поле статических данных разделяется между всеми объектами конкретного класса. Поэтому, если необходимо, чтобы часть данных совместно использовалась *всеми* объектами, то статические члены будут самым подходящим вариантом.

Посмотрим, что произойдет, если бы поле `currInterestRate` *не* было определено с ключевым словом `static`. Это означает, что каждый объект `SavingAccount` будет иметь собственную копию поля `currInterestRate`. Предположим, что вы создали сто объектов `SavingAccount` и нуждаетесь в изменении размера процентной ставки. Такое действие потребовало бы вызова метода `SetInterestRate()` сто раз! Ясно, что подобный способ моделирования "разделяемых данных" трудно считать удобным. Статические данные безупречны, когда есть значение, которое должно быть общим для всех объектов заданной категории.

На заметку! Ссылка на нестатические члены внутри реализации статического члена приводит к ошибке на этапе компиляции. В качестве связанного с этим замечания: ошибкой также будет применение ключевого слова `this` к статическому члену, потому что `this` подразумевает объект!

Определение статических конструкторов

Типичный конструктор используется для установки значений данных уровня экземпляра во время его создания. Однако что произойдет, если вы попытаетесь присвоить значение статическому элементу данных в типичном конструкторе? Вы можете быть удивлены, обнаружив, что значение сбрасывается каждый раз, когда создается новый объект!

В целях иллюстрации изменим код конструктора класса SavingsAccount, как показано ниже (также обратите внимание, что поле currInterestRate больше не устанавливается при объявлении):

```
class SavingsAccount
{
  public double currBalance;
  public static double currInterestRate;

  // Обратите внимание, что наш конструктор устанавливает
  // значение статического поля currInterestRate.
  public SavingsAccount(double balance)
  {
    currInterestRate = 0.04;  // Это статические данные!
    currBalance = balance;
  }
  ...
}
```

Теперь поместим в метод Main() следующий код:

```
static void Main( string[] args )
{
  Console.WriteLine("***** Fun with Static Data *****\n");

  // Создать объект счета.
  SavingsAccount s1 = new SavingsAccount(50);

  // Вывести текущую процентную ставку.
  Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());

  // Попытаться изменить процентную ставку через свойство.
  SavingsAccount.SetInterestRate(0.08);

  // Создать второй объект счета.
  SavingsAccount s2 = new SavingsAccount(100);

  // Должно быть выведено 0.08... не так ли??
  Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());
  Console.ReadLine();
}
```

При выполнении этого метода Main() вы увидите, что переменная currInterestRate сбрасывается каждый раз, когда создается новый объект SavingsAccount, и она всегда установлена в 0.04. Очевидно, что установка значений статических данных в нормальном конструкторе уровня экземпляра сводит на нет все их предназначение. Когда бы ни создавался новый объект, данные уровня класса сбрасываются! Один из подходов к установке статического поля предполагает применение синтаксиса инициализации членов, как это делалось изначально:

```
class SavingsAccount
{
  public double currBalance;

  // Статические данные.
  public static double currInterestRate = 0.04;
  ...
}
```

Такой подход обеспечит установку статического поля только один раз независимо от того, сколько объектов создается. Но что, если значение статических данных необходимо получать во время выполнения? Например, в типичном банковском приложении

значение переменной, представляющей процентную ставку, будет читаться из базы данных или внешнего файла. Решение задач подобного рода требует области действия метода, такого как конструктор, для выполнения соответствующих операторов кода.

По этой причине язык C# позволяет определять статический конструктор, который дает возможность безопасно устанавливать значения статических данных. Взгляните на следующее изменение в коде класса:

```
class SavingsAccount
{
  public double currBalance;
  public static double currInterestRate;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }

  // Статический конструктор!
  static SavingsAccount()
  {
    Console.WriteLine("In static ctor!");
    currInterestRate = 0.04;
  }
  ...
}
```

Выражаясь просто, статический конструктор — это специальный конструктор, который является идеальным местом для инициализации значений статических данных, если их значения не известны на этапе компиляции (например, когда значения нужно прочитать из внешнего файла или базы данных, сгенерировать случайные числа либо еще каким-то образом получить значения). Если вы снова запустите предыдущий метод `Main()`, то увидите ожидаемый вывод. Обратите внимание, что сообщение `"In static ctor!"` выводится только один раз, т.к. среда CLR вызывает все статические конструкторы перед первым использованием (и никогда не вызывает их заново для этого экземпляра приложения):

```
***** Fun with Static Data *****

In static ctor!
Interest Rate is: 0.04
Interest Rate is: 0.08
```

Ниже приведено несколько интересных моментов, касающихся статических конструкторов.

- В отдельно взятом классе может быть определен только один статический конструктор. Другими словами, перегружать статический конструктор нельзя.
- Статический конструктор не имеет модификатора доступа и не может принимать параметры.
- Статический конструктор выполняется только один раз вне зависимости от количества создаваемых объектов заданного класса.
- Исполняющая система вызывает статический конструктор, когда создает экземпляр класса или перед доступом к первому статическому члену из вызывающего кода.
- Статический конструктор выполняется перед любым конструктором уровня экземпляра.

Учитывая сказанное, при создании новых объектов SavingsAccount значения статических данных предохраняются, поскольку статический член устанавливается только один раз внутри статического конструктора независимо от количества созданных объектов.

Исходный код. Проект StaticDataAndMembers доступен в подкаталоге Chapter_5.

Определение статических классов

Ключевое слово static допускается также применять прямо на уровне класса. Когда класс определен как статический, его экземпляры нельзя создавать с использованием ключевого слова new, и он может содержать только члены или поля данных, помеченные ключевым словом static. В случае нарушения этого правила возникают ошибки на этапе компиляции.

На заметку! Вспомните, что класс (или структура), который открывает доступ только к статической функциональности, часто называется *обслуживающим классом*. При проектировании обслуживающего класса рекомендуется применять ключевое слово static к самому определению класса.

На первый взгляд это может показаться довольно странным средством, учитывая невозможность создания экземпляров класса. Тем не менее, прежде всего, класс, который содержит только статические члены и/или константные данные, не нуждается в выделении для него памяти. Чтобы продемонстрировать это, создадим новый проект консольного приложения по имени SimpleUtilityClass. Определим в нем следующий класс:

```
// Статические классы могут содержать только статические члены!
static class TimeUtilClass
{
  public static void PrintTime()
  { Console.WriteLine(DateTime.Now.ToShortTimeString()); }
  public static void PrintDate()
  { Console.WriteLine(DateTime.Today.ToShortDateString()); }
}
```

С учетом того, что класс TimeUtilClass определен с ключевым словом static, создавать его экземпляры с помощью ключевого слова new нельзя. Вместо этого вся функциональность доступна на уровне класса:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Static Classes *****\n");
  // Это работает нормально.
  TimeUtilClass.PrintDate();
  TimeUtilClass.PrintTime();
  // Ошибка на этапе компиляции! Создание экземпляра статического класса невозможно!
  TimeUtilClass u = new TimeUtilClass ();
  Console.ReadLine();
}
```

Импортирование статических членов с применением ключевого слова `using` языка C#

Последняя версия компилятора C# поддерживает новый способ использования ключевого слова `using`. Теперь можно определять директиву `using` языка C#, которая будет импортировать все статические члены в файл кода, где она находится. В целях иллюстрации предположим, что в файле C# определен обслуживающий класс. Из-за того, что в нем производятся вызовы метода `WriteLine()` класса `Console`, а также обращения к свойству `Now` класса `DateTime`, должен быть предусмотрен оператор `using` для пространства имен `System`. Поскольку все члены этих классов являются статическими, в файле кода можно указать следующие *директивы* `using static`:

```
// Импортировать статические члены классов Console и DateTime.
using static System.Console;
using static System.DateTime;
```

После такого "статического импортирования" далее в файле кода становится возможным напрямую применять статические методы классов `Console` и `DateTime`, не снабжая их префиксом в виде имени класса, в котором они определены (хотя можно по-прежнему поступать так при условии, что было импортировано пространство имен `System`). Например, модифицируем наш обслуживающий класс `TimeUtilClass`, как показано ниже:

```
static class TimeUtilClass
{
   public static void PrintTime()
   { WriteLine(Now.ToShortTimeString()); }

   public static void PrintDate()
   { WriteLine(Today.ToShortDateString()); }
}
```

Можно было бы утверждать, что данная версия класса несколько привлекательнее, т.к. уменьшилась кодовая база. В более реалистичном примере упрощения кода мог бы участвовать класс C#, интенсивно использующий класс `System.Math` (или какой-то другой обслуживающий класс). Поскольку этот класс содержит только статические члены, отчасти было бы проще указать для него оператор `using static` и затем напрямую обращаться членам класса `Math` в своем файле кода.

Однако имейте в виду, что злоупотребление операторами статического импортирования может привести в результате к путанице. Во-первых, как быть, если метод `WriteLine()` определен сразу в нескольких классах? Будет сбит с толку как компилятор, так и другие читающие ваш код. Во-вторых, если разработчик не особенно хорошо знаком с библиотеками кода .NET, то он может не знать о том, что `WriteLine()` является членом класса `Console`. До тех пор, пока разработчик не заметит набор операторов статического импортирования в начале файла кода C#, он не может быть полностью уверен в том, где данный метод в действительности определен. По указанным причинам применение операторов `using static` в настоящей книге ограничено.

К этому моменту вы должны уметь определять простые типы классов, содержащие конструкторы, поля и разнообразные статические (и нестатические) члены. Обладая такими базовыми знаниями о конструкции классов, можно приступать к ознакомлению с тремя основными принципами объектно-ориентированного программирования (ООП).

Исходный код. Проект `SimpleUtilityClass` доступен в подкаталоге `Chapter_5`.

Основные принципы объектно-ориентированного программирования

Все объектно-ориентированные языки (C#, Java, C++, Smalltalk, Visual Basic и т.д.) должны поддерживать три основных принципа ООП.

- *Инкапсуляция.* Каким образом язык скрывает детали внутренней реализации объектов и предохраняет целостность данных?
- *Наследование.* Каким образом язык стимулирует многократное использование кода?
- *Полиморфизм.* Каким образом язык позволяет трактовать связанные объекты сходным образом?

Прежде чем погрузиться в синтаксические детали каждого принципа, важно понять их базовую роль. Ниже предлагается обзор всех принципов, а в оставшейся части этой и во всей следующей главе приведены подробные сведения, связанные с ними.

Роль инкапсуляции

Первый основной принцип ООП называется *инкапсуляцией*. Эта характерная черта описывает способность языка скрывать излишние детали реализации от пользователя объекта. Например, предположим, что вы имеете дело с классом по имени `DaabaseReader`, в котором определены два главных метода: `Open()` и `Close()`.

```
// Пусть этот класс инкапсулирует детали открытия и закрытия базы данных.
DatabaseReader dbReader = new DatabaseReader();
dbReader.Open(@"C:\AutoLot.mdf");

// Сделать что-то с файлом данных и закрыть файл.
dbReader.Close();
```

Вымышленный класс `DatabaseReader` инкапсулирует внутренние детали обнаружения, загрузки, манипулирования и закрытия файла данных. Программистам нравится инкапсуляция, т.к. этот основной принцип ООП упрощает задачи кодирования. Отсутствует необходимость беспокоиться о многочисленных строках кода, которые работают "за кулисами", чтобы обеспечить функционирование класса `DatabaseReader`. Все, что понадобится — это создать экземпляр и отправить ему подходящие сообщения (например, открыть файл по имени `AutoLot.mdf`, расположенный на диске C:).

С понятием инкапсуляции программной логики тесно связана идея защиты данных. В идеале данные состояния объекта должны быть определены с применением ключевого слова `private` (или, возможно, `protected`). Таким образом, внешний мир должен вежливо попросить об изменении либо извлечении лежащего в основе значения. Это полезный подход, т.к. открыто объявленные элементы данных очень легко могут стать поврежденными (конечно, лучше случайно, чем намеренно). Вскоре будет дано формальное определение такого аспекта инкапсуляции.

Роль наследования

Следующий принцип ООП, *наследование*, отражает способность языка разрешать определение новых классов на основе определений существующих классов. По сути, наследование позволяет расширять поведение базового (или *родительского*) класса, наследуя его основную функциональность в производном подклассе (также называемом *дочерним классом*). На рис. 5.4 показан простой пример.

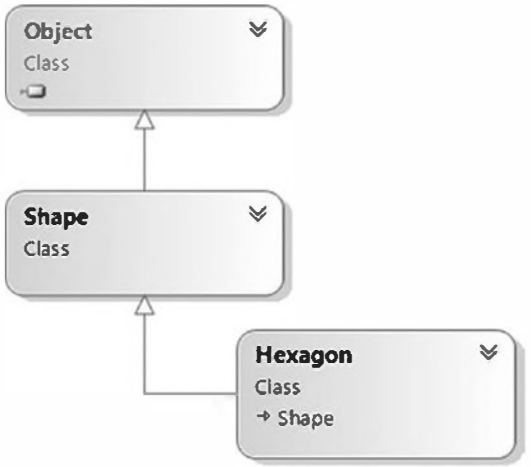


Рис. 5.4. Отношение "является" ("is-a")

Диаграмму на рис. 5.4 можно прочесть так: "шестиугольник (`Hexagon`) является фигурой (`Shape`), которая является объектом (`Object`)". При наличии классов, связанных такой формой наследования, между типами устанавливается *отношение "является" ("is-a")*. Отношение "является" называется *наследованием*.

Здесь можно предположить, что класс `Shape` определяет некоторое количество членов, являющихся общими для всех наследников (скажем, значение для представления цвета фигуры и другие значения, которые задают высоту и ширину). Учитывая, что класс `Hexagon` расширяет `Shape`, он наследует основную функциональность, определяемую классами `Shape` и `Object`, а также сам определяет дополнительные детали, связанные с шестиугольником (какими бы они ни были).

На заметку! В рамках платформы .NET класс `System.Object` всегда находится на вершине любой иерархии классов, являясь первоначальным родительским классом, и определяет общую функциональность для всех типов (как показано в главе 6).

В мире ООП существует еще одна форма повторного использования кода: модель включения/делегации, также известная как *отношение "имеет" ("has-a")* или *агрегация*. Такая форма повторного использования не применяется для установки отношений "родительский–дочерний". Вместо этого отношение "имеет" позволяет одному классу определять переменную-член другого класса и опосредованно представлять его функциональность (когда необходимо) пользователю объекта.

Например, предположим, что снова моделируется автомобиль. Может понадобиться выразить идею, что автомобиль "имеет" радиоприемник. Было бы нелогично пытаться наследовать класс `Car` от класса `Radio` или наоборот (ведь `Car` не "является" `Radio`). Взамен имеются два независимых класса, работающих совместно, причем класс `Car` создает и открывает доступ к функциональности класса `Radio`:

```
class Radio
{
  public void Power(bool turnOn)
  {
    Console.WriteLine("Radio on: {0}", turnOn);
  }
}
```

```
class Car
{
  // Car 'имеет' Radio.
  private Radio myRadio = new Radio();

  public void TurnOnRadio(bool onOff)
  {
    // Делегировать вызов внутреннему объекту.
    myRadio.Power(onOff);
  }
}
```

Обратите внимание, что пользователю объекта ничего не известно о том, что класс `Car` использует внутренний объект `Radio`:

```
static void Main(string[] args)
{
  // Внутренне вызов передается объекту Radio.
  Car viper = new Car();
  viper.TurnOnRadio(false);
}
```

Роль полиморфизма

Последним основным принципом ООП является *полиморфизм*. Эта характерная черта обозначает способность языка трактовать связанные объекты в сходной манере. В частности, данный принцип ООП позволяет базовому классу определять набор членов (формально называемый *полиморфным интерфейсом*), которые доступны всем наследникам. Полиморфный интерфейс класса конструируется с применением любого количества *виртуальных* или *абстрактных* членов (подробности ищите в главе 6).

Выражаясь кратко, *виртуальный член* — это член базового класса, определяющий стандартную реализацию, которая может быть изменена (или, выражаясь более формально, *переопределена*) в производном классе. В отличие от него *абстрактный метод* — это член базового класса, который не предоставляет стандартную реализацию, а предлагает только сигнатуру. Если класс унаследован от базового класса, который определяет абстрактный метод, то этот метод *должен* быть переопределен в производном классе. В любом случае, когда производные классы переопределяют члены, определенные в базовом классе, по существу они переопределяют свою реакцию на один и тот же запрос.

Чтобы увидеть полиморфизм в действии, давайте предоставим некоторые детали иерархии фигур, показанной на рис. 5.4. Предположим, что в классе `Shape` определен виртуальный метод `Draw()`, не принимающий параметров. С учетом того, что каждой фигуре необходимо визуализировать себя уникальным образом, подклассы вроде `Hexagon` и `Circle` могут переопределять этот метод по своему усмотрению (рис. 5.5).

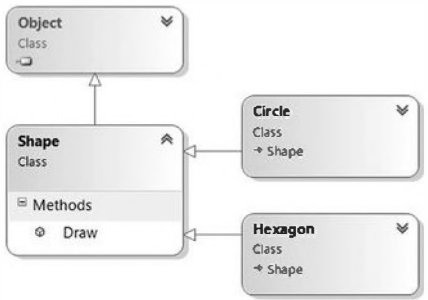
После того как полиморфный интерфейс спроектирован, можно начинать делать предположения в коде. Например, поскольку классы `Hexagon` и `Circle` унаследованы от общего родителя (`Shape`), массив элементов типа `Shape` может содержать любые объекты классов, производных от этого базового класса. Более того, учитывая, что класс `Shape` определяет полиморфный интерфейс для всех производных типов (метод `Draw()` в данном примере), уместно предположить, что каждый член массива обладает такой функциональностью.

Рассмотрим следующий метод `Main()`, который заставляет массив элементов производных от `Shape` типов визуализировать себя с использованием метода `Draw()`:

```
class Program
{
  static void Main(string[] args)
  {
    Shape[] myShapes = new Shape[3];
    myShapes[0] = new Hexagon();
    myShapes[1] = new Circle();
    myShapes[2] = new Hexagon();

    foreach (Shape s in myShapes)
    {
      // Использовать полиморфный интерфейс!
      s.Draw();
    }
    Console.ReadLine();
  }
}
```

На этом беглый обзор основных принципов ООП завершен. Остальной материал главы посвящен дальнейшим подробностям поддержки инкапсуляции в языке C#. Детали наследования и полиморфизма представлены в главе 6.



Вызов метода `Draw()` на объекте `Circle` приводит к рисованию двумерного круга.

Вызов метода `Draw()` на объекте `Hexagon` приводит к рисованию двумерного шестиугольника.

Рис. 5.5. Классический полиморфизм

Модификаторы доступа C#

При работе с инкапсуляцией вы должны всегда принимать во внимание то, какие аспекты типа являются видимыми различным частям приложения. В частности, типы (классы, интерфейсы, структуры, перечисления и делегаты), а также их члены (свойства, методы, конструкторы и поля) определяются с использованием специального ключевого слова, управляющего "видимостью" элемента для других частей приложения. Хотя в C# для управления доступом предусмотрены многочисленные ключевые слова, они отличаются в том, к чему могут быть успешно применены (к типу или члену). Модификаторы доступа и особенности их использования описаны в табл. 5.1.

В этой главе рассматриваются только ключевые слова `public` и `private`. В последующих главах будет исследована роль модификаторов `internal` и `protected internal` (удобных при построении библиотек кода .NET) и модификатора `protected` (полезного при создании иерархий классов).

Стандартные модификаторы доступа

По умолчанию члены типов являются *неявно закрытыми* (`private`), тогда как типы — *неявно внутренними* (`internal`).

Таблица 5.1. Модификаторы доступа C#

Модификатор доступа	К чему может быть применен	Практический смысл
`public`	Типы или члены типов	Открытые (`public`) элементы не имеют ограничений доступа. Открытый член может быть доступен из объекта, а также из любого производного класса. Открытый тип может быть доступен из других внешних сборок
`private`	Члены типов или вложенные типы	Закрытые (`private`) элементы могут быть доступны только классу (или структуре), где они определены
`protected`	Члены типов или вложенные типы	Защищенные (`protected`) элементы могут использоваться классом, который их определяет, и любым дочерним классом. Однако защищенные элементы не доступны внешнему миру через операцию точки (`.`)
`internal`	Типы или члены типов	Внутренние (`internal`) элементы доступны только в рамках текущей сборки. Таким образом, если в библиотеке классов .NET определен набор внутренних типов, то другие сборки не смогут ими пользоваться
`protected internal`	Члены типов или вложенные типы	Когда в объявлении элемента указана комбинация ключевых слов `protected` и `internal`, то такой элемент будет доступен внутри определяющей его сборки, внутри определяющего класса и для всех его наследников

Таким образом, следующее определение класса автоматически установлено как `internal`, а стандартный конструктор этого типа — как `private` (тем не менее, как и можно было предполагать, закрытые конструкторы классов нужны редко):

```
// Внутренний класс с закрытым стандартным конструктором.
class Radio
{
  Radio(){}
}
```

Если вы предпочитаете явное объявление, то можете добавить соответствующие ключевые слова без каких-либо негативных последствий (кроме дополнительных усилий по набору):

```
// Внутренний класс с закрытым стандартным конструктором.
internal class Radio
{
  private Radio(){}
}
```

Чтобы позволить другим частям программы обращаться к членам объекта, вы должны определить эти члены с ключевым словом `public` (или возможно с ключевым словом `protected`, которое объясняется в следующей главе). Вдобавок, если вы хотите открыть доступ к `Radio` внешним сборкам (что удобно при построении библиотек кода .NET, как показано в главе 14), то придется добавить к нему модификатор `public`:

```
// Открытый класс с открытым стандартным конструктором.
public class Radio
{
  public Radio(){}
}
```

Модификаторы доступа и вложенные типы

Как упоминалось в табл. 5.1, модификаторы доступа `private`, `protected` и `protected internal` могут применяться к *вложенному типу*. Вложение типов будет подробно рассматриваться в главе 6, а пока достаточно знать, что вложенный тип — это тип, объявленный прямо внутри области видимости класса или структуры. Для примера ниже приведено закрытое перечисление (по имени `CarColor`), вложенное в открытый класс (по имени `SportsCar`):

```
public class SportsCar
{
  // Нормально! Вложенные типы могут быть помечены как private.
  private enum CarColor
  {
    Red, Green, Blue
  }
}
```

Здесь допустимо применять модификатор доступа `private` к вложенному типу. Однако невложенные типы (вроде `SportsCar`) могут определяться только с модификатором `public` или `internal`. Таким образом, следующее определение класса незаконно:

```
// Ошибка! Невложенный тип не может быть помечен как private!
private class SportsCar
{}
```

Первый принцип ООП: службы инкапсуляции C#

Концепция инкапсуляции вращается вокруг идеи о том, что данные класса не должны быть напрямую доступными через его экземпляр. Наоборот, данные класса определяются как закрытые. Если пользователь объекта желает изменить его состояние, то должен делать это косвенно, используя открытые члены. Чтобы проиллюстрировать необходимость в службах инкапсуляции, предположим, что вы создали такое определение класса:

```
// Класс с единственным открытым полем.
class Book
{
  public int numberOfPages;
}
```

Проблема с открытыми данными заключается в том, что сами по себе они неспособны "понять", является ли присваиваемое значение допустимым с точки зрения текущих бизнес-правил системы. Как вам известно, верхний предел значений для типа `int` в C# довольно высок (2 147 483 647), поэтому компилятор разрешит следующее присваивание:

```
// Хм... Ничего себе мини-новелла!
static void Main(string[] args)
{
  Book miniNovel = new Book();
  miniNovel.numberOfPages = 30000000;
}
```

Хотя границы типа данных `int` не превышены, понятно, что мини-новелла на 30 миллионов страниц выглядит несколько неправдоподобно. Как видите, открытые поля не предоставляют способа ограничения значений верхними (или нижними) логически-

ми пределами. Если в системе установлено текущее бизнес-правило, которое регламентирует, что книга должна иметь от 1 до 1000 страниц, то совершенно неясно, как обеспечить его выполнение программным образом. Из-за этого открытым полям обычно нет места в определениях классов производственного уровня.

На заметку! Говоря точнее, члены класса, которые представляют состояние объекта, не должны помечаться как `public`. В то же время далее в главе вы увидите, что вполне нормально иметь открытые константы и открытые поля, допускающие только чтение.

Инкапсуляция предлагает способ предохранения целостности данных состояния для объекта. Вместо определения открытых полей (которые могут легко привести к повреждению данных) необходимо выработать у себя привычку определять *закрытые данные*, управление которыми осуществляется опосредованно с применением одного из двух главных приемов:

- определение пары открытых методов доступа и изменения;
- определение открытого свойства .NET.

Какой бы прием не был выбран, идея заключается в том, что хорошо инкапсулированный класс должен защищать свои данные и скрывать подробности своего функционирования от любопытных глаз из внешнего мира. Это часто называют *программированием в стиле черного ящика*. Преимущество такого подхода в том, что объект может свободно изменять внутреннюю реализацию любого метода. Работа существующего кода, который использует этот метод, не нарушается при условии, что параметры и возвращаемые значения методов остаются неизменными.

Инкапсуляция с использованием традиционных методов доступа и изменения

До конца этой главы будет построен довольно полный класс, моделирующий обычного сотрудника. Для начала создадим новый проект консольного приложения по имени `EmployeeApp` и добавим в него новый файл класса (`Employee.cs`) с помощью пункта меню Project⇨Add class (Проект⇨Добавить класс). Дополним класс `Employee` следующими полями, методами и конструкторами:

```
class Employee
{
  // Поля данных.
  private string empName;
  private int empID;
  private float currPay;

  // Конструкторы.
  public Employee() {}
  public Employee(string name, int id, float pay)
  {
    empName = name;
    empID = id;
    currPay = pay;
  }

  // Методы.
  public void GiveBonus(float amount)
  {
    currPay += amount;
  }
```

```
  public void DisplayStats()
  {
    Console.WriteLine("Name: {0}", empName);  // имя сотрудника
    Console.WriteLine("ID: {0}", empID); // идентификационный номер сотрудника
    Console.WriteLine("Pay: {0}", currPay);   // текущая выплата
   }
}
```

Обратите внимание, что поля класса `Employee` в текущий момент определены с применением ключевого слова `private`. Учитывая это, поля `empName`, `empID` и `currPay` не доступны напрямую через объектную переменную. Следовательно, показанная ниже логика в `Main()` приведет к ошибкам на этапе компиляции:

```
static void Main(string[] args)
{
  Employee emp = new Employee();
  // Ошибка! Невозможно напрямую обращаться к закрытым полям объекта!
  emp.empName = "Marv";
}
```

Если вы хотите, чтобы внешний мир взаимодействовал с полным именем сотрудника, то традиционный подход (который распространен в Java) предусматривает определение методов доступа (метод get) и изменения (метод set). Роль метода get заключается в возвращении вызывающему коду текущего значения лежащих в основе данных состояния. Метод set позволяет вызывающему коду изменять текущее значение лежащих в основе данных состояния при условии удовлетворения бизнес-правил.

В целях иллюстрации давайте инкапсулируем поле `empName`. Для этого к существующему классу `Employee` необходимо добавить показанные ниже открытые методы. Обратите внимание, что метод `SetName()` выполняет проверку входных данных, чтобы удостовериться, что строка имеет длину 15 символов или меньше. Если это не так, то на консоль выводится сообщение об ошибке и происходит возврат без внесения изменений в поле `empName`.

На заметку! В случае класса производственного уровня проверку длины строки с именем сотрудника следовало бы предусмотреть также и внутри логики конструктора. Мы пока проигнорируем эту деталь, но улучшим код позже, во время исследований синтаксиса свойств .NET.

```
class Employee
{
  // Поля данных.
  private string empName;
  ...
  // Метод доступа (метод get).
  public string GetName()
  {
    return empName;
  }
  // Метод изменения (метод set).
  public void SetName(string name)
  {
    // Перед присваиванием проверить входное значение.
    if (name.Length > 15)
      Console.WriteLine("Error! Name length exceeds 15 characters!");
                    // Ошибка! Длина имени превышает 15 символов!
```

```
    else
      empName = name;
  }
}
```

Такой подход требует наличия двух уникально именованных методов для управления единственным элементом данных. Чтобы протестировать новые методы, модифицируем метод `Main()` следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Encapsulation *****\n");
  Employee emp = new Employee("Marvin", 456, 30000);
  emp.GiveBonus(1000);
  emp.DisplayStats();

  // Использовать методы get/set для взаимодействия с именем сотрудника,
  // представленного объектом.
  emp.SetName("Marv");
  Console.WriteLine("Employee is named: {0}", emp.GetName());
  Console.ReadLine();
}
```

Благодаря коду в методе `SetName()` попытка указать для имени строку, содержащую более 15 символов (как показано ниже), приводит к выводу на консоль жестко закодированного сообщения об ошибке:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Encapsulation *****\n");
  ...
  // Длиннее 15 символов! На консоль выводится сообщение об ошибке.
  Employee emp2 = new Employee();
  emp2.SetName("Xena the warrior princess");

  Console.ReadLine();
}
```

Пока все идет хорошо. Мы инкапсулировали закрытое поле `empName` с использованием двух открытых методов с именами `GetName()` и `SetName()`. Для дальнейшей инкапсуляции данных в классе `Employee` понадобится добавить разнообразные дополнительные методы (такие как `GetID()`, `SetID()`, `GetCurrentPay()`, `SetCurrentPay()`). В каждом методе, изменяющем данные, может содержаться несколько строк кода, в которых реализована проверка дополнительных бизнес-правил. Несмотря на то что это определенно достижимо, для инкапсуляции данных класса в языке C# имеется удобная альтернативная система записи.

Инкапсуляция с использованием свойств .NET

Хотя инкапсулировать поля данных можно с применением традиционной пары методов get и set, в языках .NET предпочтение отдается обеспечению инкапсуляции данных с использованием *свойств*. Прежде всего, имейте в виду, что свойства — это всего лишь упрощенное представление "настоящих" методов доступа и изменения. Следовательно, проектировщик класса по-прежнему может выполнить любую внутреннюю логику перед присваиванием значения (например, преобразовать в верхний регистр, избавиться от недопустимых символов, проверить вхождение внутрь границ и т.д.).

Ниже приведен измененный класс Employee, который теперь обеспечивает инкапсуляцию каждого поля с использованием синтаксиса свойств вместо традиционных методов get и set.

```
class Employee
{
  // Поля данных.
  private string empName;
  private int empID;
  private float currPay;
  // Свойства.
  public string Name
  {
    get { return empName; }
    set
    {
      if (value.Length > 15)
        Console.WriteLine("Error! Name length exceeds 15 characters!");
                        // Ошибка! Длина имени превышает 15 символов!
      else
        empName = value;
    }
  }
  // Можно было бы добавить дополнительные бизнес-правила для установки
  // этих свойств, но в настоящем примере в этом нет необходимости.
  public int ID
  {
    get { return empID; }
    set { empID = value; }
  }
  public float Pay
  {
    get { return currPay; }
    set { currPay = value; }
  }
  ...
}
```

Свойство C# состоит из определений областей get (метод доступа) и set (метод изменения) прямо внутри самого свойства. Обратите внимание, что свойство указывает тип инкапсулируемых им данных способом, который выглядит как возвращаемое значение. Кроме того, в отличие от метода, при определении свойства не применяются круглые скобки (даже пустые). Взгляните на следующий комментарий к текущему свойству ID:

```
// int представляет тип данных, инкапсулируемых этим свойством.
public int ID  // Обратите внимание на отсутствие круглых скобок.
{
  get { return empID; }
  set { empID = value; }
}
```

В области set свойства используется лексема value, которая представляет входное значение, присваиваемое свойству вызывающим кодом. Эта лексема *не* является настоящим ключевым словом C#, а представляет собой то, что называется *контекстным ключевым словом*. Когда лексема value находится внутри области set, она всегда обозначает значение, присваиваемое вызывающим кодом, и всегда имеет тип, совпадающий с типом самого свойства.

Таким образом, вот как свойство Name может проверить допустимую длину string:

```
public string Name
{
  get { return empName; }
  set
  {
    // Здесь value на самом деле имеет тип string.
    if (value.Length > 15)
       Console.WriteLine("Error! Name length exceeds 15 characters!");
                        // Ошибка! Длина имени превышает 15 символов!
    else
      empName = value;
  }
}
```

После определения этих свойств вызывающему коду кажется, что он имеет дело с *открытым элементом данных*; однако "за кулисами" при каждом обращении к ним вызывается корректный блок get или set, предохраняя инкапсуляцию:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Encapsulation *****\n");
  Employee emp = new Employee("Marvin", 456, 30000);
  emp.GiveBonus(1000);
  emp.DisplayStats();
  // Переустановка и затем получение свойства Name.
  emp.Name = "Marv";
  Console.WriteLine("Employee is named: {0}", emp.Name);
  Console.ReadLine();
}
```

Свойства (как противоположность методам доступа и изменения) также облегчают манипулирование типами, поскольку способны реагировать на внутренние операции C#. В целях иллюстрации предположим, что тип класса Employee имеет внутреннюю закрытую переменную-член, представляющую возраст сотрудника. Ниже показаны необходимые изменения (обратите внимание на применение цепочки вызовов конструкторов):

```
class Employee
{
  ...
  // Новое поле и свойство.
  private int empAge;
  public int Age
  {
    get { return empAge; }
    set { empAge = value; }
  }
  // Обновленные конструкторы.
  public Employee() {}
  public Employee(string name, int id, float pay)
    :this(name, 0, id, pay){}
  public Employee(string name, int age, int id, float pay)
  {
    empName = name;
    empID = id;
    empAge = age;
    currPay = pay;
  }
```

```
  // Обновленный метод DisplayStats() теперь учитывает возраст.
  public void DisplayStats()
  {
    Console.WriteLine("Name: {0}", empName);  // имя сотрудника
    Console.WriteLine("ID: {0}", empID); // идентификационный номер сотрудника
    Console.WriteLine("Age: {0}", empAge);    // возраст сотрудника
    Console.WriteLine("Pay: {0}", currPay);   // текущая выплата
  }
}
```

Теперь предположим, что создан объект `Employee` по имени `joe`. Необходимо сделать так, чтобы в день рождения сотрудника возраст увеличивался на 1 год. Используя традиционные методы set и get, пришлось бы написать приблизительно такой код:

```
Employee joe = new Employee();
joe.SetAge(joe.GetAge() + 1);
```

Тем не менее, если `empAge` инкапсулируется посредством свойства по имени `Age`, то код будет проще:

```
Employee joe = new Employee();
joe.Age++;
```

Использование свойств внутри определения класса

Свойства, а в особенности их часть `set` — это общепринятое место для размещения бизнес-правил класса. В настоящий момент класс `Employee` имеет свойство `Name`, которое гарантирует, что длина имени не превышает 15 символов. Остальные свойства (`ID`, `Pay` и `Age`) также могут быть обновлены соответствующей логикой.

Хотя все это хорошо, но необходимо также принимать во внимание и то, что обычно происходит внутри конструктора класса. Конструктор получает входные параметры, проверяет данные на предмет допустимости и затем присваивает значения внутренним закрытым полям. Пока что главный конструктор *не* проверяет входные строковые данные на вхождение в диапазон допустимых значений, поэтому его можно было бы изменить следующим образом:

```
public Employee(string name, int age, int id, float pay)
{
  // Это выглядит как проблема...
  if (name.Length > 15)
    Console.WriteLine("Error! Name length exceeds 15 characters!");
                    // Ошибка! Длина имени превышает 15 символов!
  else
    empName = name;
  empID = id;
  empAge = age;
  currPay = pay;
}
```

Наверняка вы заметили проблему, связанную с этим подходом. Свойство `Name` и главный конструктор выполняют одну и ту же проверку на предмет ошибок. Реализуя проверки для других элементов данных, есть реальный шанс столкнуться с дублированием кода. Стремясь рационализировать код и изолировать всю проверку, касающуюся ошибок, в каком-то центральном местоположении, вы добьетесь успеха, если для получения и установки значений внутри класса *всегда* будете применять свойства. Взгляните на показанный ниже модифицированный конструктор:

```
public Employee(string name, int age, int id, float pay)
{
  // Уже лучше! Используйте свойства для установки данных класса.
  // Это сократит количество дублированных проверок на предмет ошибок.
  Name = name;
  Age = age;
  ID = id;
  Pay = pay;
}
```

Помимо обновления конструкторов для применения свойств при присваивании значений рекомендуется использовать свойства повсюду в реализации класса, чтобы гарантировать неизменное соблюдение бизнес-правил. Во многих случаях прямая ссылка на лежащие в основе закрытые данные производится только внутри самого свойства. Имея все это в виду, модифицируем класс `Employee`:

```
class Employee
{
  // Поля данных.
  private string empName;
  private int empID;
  private float currPay;
  private int empAge;

  // Конструкторы.
  public Employee() { }
  public Employee(string name, int id, float pay)
    :this(name, 0, id, pay){}
  public Employee(string name, int age, int id, float pay)
  {
    Name = name;
    Age = age;
    ID = id;
    Pay = pay;
  }

  // Методы.
  public void GiveBonus(float amount)
  { Pay += amount; }

  public void DisplayStats()
  {
    Console.WriteLine("Name: {0}", Name);
    Console.WriteLine("ID: {0}", ID);
    Console.WriteLine("Age: {0}", Age);
    Console.WriteLine("Pay: {0}", Pay);
  }

  // Свойства остаются прежними...
  ...
}
```

Свойства, которые допускают только чтение и только запись

При инкапсуляции данных может возникнуть желание сконфигурировать *свойство, допускающее только чтение*. Для этого нужно просто опустить блок `set`. Аналогично, если необходимо *свойство, доступное только для записи*, то следует опустить блок `get`. Например, пусть имеется новое свойство по имени `SocialSecurityNumber`, которое ин-

капсулирует закрытую строковую переменную empSSN. Вот как превратить его в свойство, доступное только для чтения:

```
public string SocialSecurityNumber
{
  get { return empSSN; }
}
```

Теперь предположим, что конструктор класса принимает новый параметр, который дает возможность указывать в вызывающем коде номер карточки социального страхования для объекта сотрудника. Поскольку свойство SocialSecurityNumber допускает только чтение, устанавливать значение так, как показано ниже, нельзя:

```
public Employee(string name, int age, int id, float pay, string ssn)
{
  Name = name;
  Age = age;
  ID = id;
  Pay = pay;
  // Если свойство предназначено только для чтения, это больше невозможно!
  SocialSecurityNumber = ssn;
}
```

Если только вы не готовы переделать это свойство в поддерживающее чтение и запись, то единственным выбором будет применение лежащей в основе переменной-члена empSSN внутри логики конструктора:

```
public Employee(string name, int age, int id, float pay, string ssn)
{
  ...
  // Проверить должным образом входной параметр ssn и затем установить
значение.
  empSSN = ssn;
}
```

Исходный код. Проект EmployeeApp доступен в подкаталоге Chapter_5.

Еще раз о ключевом слове static: определение статических свойств

Ранее в этой главе рассказывалось о роли ключевого слова static. Теперь, когда вы научились использовать синтаксис свойств C#, мы можем формализовать статические свойства. В проекте StaticDataAndMembers класс SavingsAccount имел два открытых статических метода для получения и установки процентной ставки. Однако более стандартный подход предусматривает помещение этого элемента данных в статическое свойство. Ниже приведен пример (обратите внимание на применение ключевого слова static):

```
// Простой класс депозитного счета.
class SavingsAccount
{
  // Данные уровня экземпляра.
  public double currBalance;
  // Статический элемент данных.
  private static double currInterestRate = 0.04;
```

```
  // Статическое свойство.
  public static double InterestRate
  {
    get { return currInterestRate; }
    set { currInterestRate = value; }
  }
  ...
}
```

Если вы хотите использовать это свойство вместо предыдущих статических методов, то понадобится модифицировать метод `Main()`:

```
// Вывести текущую процентную ставку через свойство.
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.InterestRate);
```

Понятие автоматических свойств

При создании свойств для инкапсуляции данных часто обнаруживается, что области `set` содержат код для применения бизнес-правил программы. Тем не менее, в некоторых случаях нужна только простая логика извлечения или установки значения. В результате получается большой объем кода следующего вида:

```
// Тип Car, использующий стандартный синтаксис свойств.
class Car
{
  private string carName = "";
  public string PetName
  {
    get { return carName; }
    set { carName = value; }
  }
}
```

В таких случаях многократное определение закрытых поддерживающих полей и простых свойств может стать слишком громоздким. Например, при построении класса, которому нужно 9 закрытых элементов данных, в итоге получаются 9 связанных с ними свойств, которые представляют собой не более чем тонкие оболочки для служб инкапсуляции. Чтобы упростить процесс обеспечения простой инкапсуляции данных полей, можно использовать *синтаксис автоматических свойств*. Как следует из названия, это средство перекладывает работу по определению закрытых поддерживающих полей и связанных с ними свойств C# на компилятор за счет применения небольшого нововведения в синтаксисе. В целях иллюстрации создадим новый проект консольного приложения по имени `AutoProps`. Взгляните на переделанный класс `Car`, в котором используется этот синтаксис для быстрого создания трех свойств:

```
class Car
{
  // Автоматические свойства! Нет нужды в определении поддерживающих полей.
  public string PetName { get; set; }
  public int Speed { get; set; }
  public string Color { get; set; }
}
```

На заметку! Среда Visual Studio предлагает фрагмент кода `prop`. Если вы наберете слово `prop` и два раза нажмете клавишу <Tab>, то IDE-среда сгенерирует начальный код для нового автоматического свойства. Затем с помощью клавиши <Tab> можно циклически проходить по всем частям определения и заполнять необходимые детали. Испытайте такой прием.

При определении автоматического свойства вы просто указываете модификатор доступа, лежащий в основе тип данных, имя свойства и пустые области `get/set`. Во время компиляции тип будет оснащен автоматически сгенерированным поддерживающим полем и подходящей реализацией логики `get/set`.

На заметку! Имя автоматически сгенерированного закрытого поддерживающего поля не видно в кодовой базе C#. Просмотреть его можно только с помощью такого инструмента, как `ildasm.exe`.

В текущей версии языка C# теперь возможно определять "автоматическое свойство только для чтения", опуская область `set`. Тем не менее, определять свойство, предназначенное только для записи, не разрешено. Вот пример:

```
// Свойство только для чтения? Допустимо!
public int MyReadOnlyProp { get; }

// Свойство только для записи? Ошибка!
public int MyWriteOnlyProp { set; }
```

Взаимодействие с автоматическими свойствами

Поскольку компилятор будет определять закрытые поддерживающие поля на этапе компиляции (и учитывая, что эти поля в коде C# непосредственно не доступны), в классе, который имеет автоматические свойства, для установки и чтения лежащих в их основе значений всегда должен применяться синтаксис свойств. Это важно отметить, т.к. многие программисты напрямую используют закрытые поля *внутри* определения класса, что в данном случае невозможно. Например, если бы класс `Car` содержал метод `DisplayStats()`, то в его реализации применялись бы имена свойств:

```
class Car
{
  // Автоматические свойства!
  public string PetName { get; set; }
  public int Speed { get; set; }
  public string Color { get; set; }
  public void DisplayStats()
  {
    Console.WriteLine("Car Name: {0}", PetName);
    Console.WriteLine("Speed: {0}", Speed);
    Console.WriteLine("Color: {0}", Color);
  }
}
```

При использовании экземпляра класса, определенного с автоматическими свойствами, можно присваивать и получать значения с помощью вполне ожидаемого синтаксиса свойств:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Automatic Properties *****\n");
  Car c = new Car();
  c.PetName = "Frank";
  c.Speed = 55;
  c.Color = "Red";
  Console.WriteLine("Your car is named {0}? That's odd...", c.PetName);
  c.DisplayStats();
  Console.ReadLine();
}
```

Автоматические свойства и стандартные значения

Когда автоматические свойства применяются для инкапсуляции числовых и булевских данных, их можно использовать напрямую внутри кодовой базы, поскольку скрытым поддерживающим полям будут присвоены безопасные стандартные значения (`false` для булевских и `0` для числовых данных). Но имейте в виду, что если синтаксис автоматического свойства применяется для упаковки переменной другого класса, то скрытое поле ссылочного типа также будет установлено в стандартное значение `null` (и это может привести к проблеме, если не проявить должную осторожность).

Давайте добавим в текущий проект новый класс по имени `Garage` (представляющий гараж), в котором используются два автоматических свойства (разумеется, реальный класс гаража может поддерживать коллекцию объектов `Car`; однако в данный момент проигнорируем эту деталь):

```
class Garage
{
  // Скрытое поддерживающее поле int установлено в 0!
  public int NumberOfCars { get; set; }

  // Скрытое поддерживающее поле Car установлено в null!
  public Car MyAuto { get; set; }
}
```

Имея стандартные значения C# для полей данных, значение `NumberOfCars` можно вывести в том виде, как оно есть (поскольку ему автоматически присвоено значение 0), но если напрямую обратиться к `MyAuto`, то во время выполнения сгенерируется исключение ссылки на `null`, потому что лежащей в основе переменной-члену типа `Car` не был присвоен новый объект:

```
static void Main(string[] args)
{
  ...
  Garage g = new Garage();

  // Нормально, выводится стандартное значение 0.
  Console.WriteLine("Number of Cars: {0}", g.NumberOfCars);

  // Ошибка времени выполнения! Поддерживающее поле в данный момент равно null!
  Console.WriteLine(g.MyAuto.PetName);
  Console.ReadLine();
}
```

Чтобы решить эту проблему, можно модифицировать конструкторы класса, обеспечив безопасное создание объекта. Ниже показан пример:

```
class Garage
{
  // Скрытое поддерживающее поле установлено в 0!
  public int NumberOfCars { get; set; }

  // Скрытое поддерживающее поле установлено в null!
  public Car MyAuto { get; set; }

  // Для переопределения стандартных значений, присвоенных скрытым
  // поддерживающим полям, должны использоваться конструкторы.
  public Garage()
  {
    MyAuto = new Car();
    NumberOfCars = 1;
  }
```

```
  public Garage(Car car, int number)
  {
    MyAuto = car;
    NumberOfCars = number;
  }
}
```

После такого изменения объект Car теперь можно помещать в объект Garage:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Automatic Properties *****\n");
  // Создать объект автомобиля.
  Car c = new Car();
  c.PetName = "Frank";
  c.Speed = 55;
  c.Color = "Red";
  c.DisplayStats();
  // Поместить автомобиль в гараж.
  Garage g = new Garage();
  g.MyAuto = c;
  // Вывести количество автомобилей в гараже.
  Console.WriteLine("Number of Cars in garage: {0}", g.NumberOfCars);
  // Вывести название автомобиля.
  Console.WriteLine("Your car is named: {0}", g.MyAuto.PetName);
  Console.ReadLine();
}
```

Инициализация автоматических свойств

Наряду с тем, что предыдущий подход работает вполне нормально, в последней версии C# появилась новая языковая возможность, которая содействует упрощению способа присваивания автоматическим свойствам их начальных значений. Как упоминалось в начале главы, полю данных в классе можно напрямую присваивать начальное значение при его объявлении. Например:

```
class Car
{
  private int numberOfDoors = 2;
}
```

В подобной же манере теперь язык C# позволяет присваивать начальные значения лежащим в основе поддерживающим полям, которые генерируются компилятором. Это уменьшает сложности, связанные с добавлением дополнительных операторов кода в конструкторы класса, которые обеспечивают корректную установку данных свойств.

Ниже представлена модифицированная версия класса Garage с инициализацией автоматических свойств подходящими значениями. Обратите внимание, что больше нет необходимости в добавлении к стандартному конструктору класса дополнительной логики для выполнения безопасного присваивания. Здесь производится непосредственное присваивание нового объекта Car свойству MyAuto.

```
class Garage
{
  // Скрытое поддерживающее поле установлено в 1.
  public int NumberOfCars { get; set; } = 1;
  // Скрытое поддерживающее поле установлено в новый объект Car.
  public Car MyAuto { get; set; } = new Car();
```

```
  public Garage(){}
  public Garage(Car car, int number)
  {
    MyAuto = car;
    NumberOfCars = number;
  }
}
```

Вы наверняка согласитесь с тем, что автоматические свойства — это очень полезное средство языка программирования C#, т.к. отдельные свойства в классе можно определять с применением модернизированного синтаксиса. Конечно, если вы создаете свойство, которое помимо получения и установки закрытого поддерживающего поля требует дополнительного кода (такого как логика проверки достоверности, запись в журнал событий, взаимодействие с базой данных и т.д.), то его придется определять как "нормальное" свойство .NET вручную. Автоматические свойства C# не делают ничего кроме предоставления простой инкапсуляции для лежащей в основе порции (сгенерированных компилятором) закрытых данных.

Исходный код. Проект `AutoProps` доступен в подкаталоге `Chapter_5`.

Понятие синтаксиса инициализации объектов

Как можно было видеть повсеместно в этой главе, при создании нового объекта конструктор позволяет указывать начальные значения. Вдобавок свойства позволяют безопасным образом получать и устанавливать лежащие в основе данные. При работе со сторонними классами, включая классы из библиотеки базовых классов .NET, нередко можно обнаружить, что в них отсутствует конструктор, который бы позволил установить абсолютно все порции данных состояния. С учетом этого программист обычно вынужден выбирать наилучший конструктор из возможных, а затем присваивать остальные значения, используя предлагаемый набор свойств.

Для упрощения процесса создания и подготовки объекта в C# предлагается *синтаксис инициализации объектов*. Такой прием делает возможным создание новой объектной переменной и присваивание значений многочисленным свойствам и/или открытым полям в нескольких строках кода. Синтаксически инициализатор объекта выглядит как список разделенных запятыми значений, помещенный в фигурные скобки (`{}`). Каждый элемент в списке инициализации отображается на имя открытого поля или открытого свойства инициализируемого объекта.

Чтобы увидеть этот синтаксис в действии, создадим новый проект консольного приложения по имени `ObjectInitializers`. Ниже показан класс `Point`, в котором присутствуют автоматические свойства (для синтаксиса инициализации объектов они не обязательны, но помогают получить более лаконичный код):

```
class Point
{
  public int X { get; set; }
  public int Y { get; set; }

  public Point(int xVal, int yVal)
  {
    X = xVal;
    Y = yVal;
  }
  public Point() { }
```

```
  public void DisplayStats()
  {
    Console.WriteLine("[{0}, {1}]", X, Y);
  }
}
```

А теперь посмотрим, как создавать объекты `Point`, с применением любого из следующих подходов:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Init Syntax *****\n");
  // Создать объект Point, устанавливая каждое свойство вручную.
  Point firstPoint = new Point();
  firstPoint.X = 10;
  firstPoint.Y = 10;
  firstPoint.DisplayStats();
  // Создать объект Point посредством специального конструктора.
  Point anotherPoint = new Point(20, 20);
  anotherPoint.DisplayStats();
  // Создать объект Point, используя синтаксис инициализации объектов.
  Point finalPoint = new Point { X = 30, Y = 30 };
  finalPoint.DisplayStats();
  Console.ReadLine();
}
```

При создании последней переменной `Point` специальный конструктор не используется (как это делается традиционно), а взамен устанавливаются значения открытых свойств `X` и `Y`. "За кулисами" вызывается стандартный конструктор типа, за которым следует установка значений указанных свойств. В этом отношении синтаксис инициализации объектов представляет собой просто сокращение синтаксиса для создания переменной класса с применением стандартного конструктора и установки данных состояния свойство за свойством.

Вызов специальных конструкторов с помощью синтаксиса инициализации

В предшествующих примерах объекты типа `Point` инициализировались путем неявного вызова стандартного конструктора этого типа:

```
// Здесь стандартный конструктор вызывается неявно.
Point finalPoint = new Point { X = 30, Y = 30 };
```

При желании стандартный конструктор допускается вызывать и явно:

```
// Здесь стандартный конструктор вызывается явно.
Point finalPoint = new Point() { X = 30, Y = 30 };
```

Имейте в виду, что при конструировании объекта типа с использованием синтаксиса инициализации можно вызывать *любой* конструктор, определенный в классе. В настоящий момент в типе `Point` определен конструктор с двумя аргументами для установки позиции (*x*, *y*). Таким образом, следующее объявление переменной `Point` приведет к установке `X` в `100` и `Y` в `100` независимо от того факта, что в аргументах конструктора указаны значения `10` и `16`:

```
// Вызов специального конструктора.
Point pt = new Point(10, 16) { X = 100, Y = 100 };
```

Имея текущее определение типа `Point`, вызов специального конструктора с применением синтаксиса инициализации не особенно полезен (и излишне многословен). Тем

не менее, если тип Point предоставляет новый конструктор, который позволяет вызывающему коду устанавливать цвет (через специальное перечисление PointColor), то комбинация специальных конструкторов и синтаксиса инициализации объектов становится ясной. Модифицируем тип Point, как показано далее:

```
enum PointColor
{ LightBlue, BloodRed, Gold }
class Point
{
  public int X { get; set; }
  public int Y { get; set; }
  public PointColor Color{ get; set; }
  public Point(int xVal, int yVal)
  {
    X = xVal;
    Y = yVal;
    Color = PointColor.Gold;
  }
  public Point(PointColor ptColor)
  {
    Color = ptColor;
  }
  public Point()
    : this(PointColor.BloodRed){ }
  public void DisplayStats()
  {
    Console.WriteLine("[{0}, {1}]", X, Y);
    Console.WriteLine("Point is {0}", Color);
  }
}
```

Посредством этого нового конструктора теперь можно создавать точку золотистого цвета (в позиции (90, 20)):

```
// Вызов более интересного специального конструктора
// с помощью синтаксиса инициализации.
Point goldPoint = new Point(PointColor.Gold){ X = 90, Y = 20 };
goldPoint.DisplayStats();
```

Инициализация данных с помощью синтаксиса инициализации

Как кратко упоминалось ранее в этой главе (и будет подробно обсуждаться в главе 6), отношение "имеет" ("has-a") позволяет формировать новые классы, определяя переменные-члены существующих классов. Например, пусть имеется класс Rectangle, в котором для представления координат верхнего левого и нижнего правого углов используется тип Point. Так как автоматические свойства устанавливают все переменные с типами классов в null, новый класс будет реализован с применением "традиционного" синтаксиса свойств:

```
class Rectangle
{
  private Point topLeft = new Point();
  private Point bottomRight = new Point();
  public Point TopLeft
  {
    get { return topLeft; }
    set { topLeft = value; }
  }
```

```
  public Point BottomRight
  {
    get { return bottomRight; }
    set { bottomRight = value; }
  }
  public void DisplayStats()
  {
    Console.WriteLine("[TopLeft: {0}, {1}, {2} BottomRight: {3}, {4}, {5}]",
      topLeft.X, topLeft.Y, topLeft.Color,
      bottomRight.X, bottomRight.Y, bottomRight.Color);
  }
}
```

С помощью синтаксиса инициализации объектов можно было бы создать новую переменную `Rectangle` и установить внутренние объекты `Point` следующим образом:

```
// Создать и инициализировать объект Rectangle.
Rectangle myRect = new Rectangle
{
  TopLeft = new Point { X = 10, Y = 10 },
  BottomRight = new Point { X = 200, Y = 200}
};
```

Преимущество синтаксиса инициализации объектов в том, что он по существу сокращает объем вводимого кода (предполагая, что подходящий конструктор отсутствует). Вот как выглядит традиционный подход к созданию такого же экземпляра `Rectangle`:

```
// Традиционный подход.
Rectangle r = new Rectangle();
Point p1 = new Point();
p1.X = 10;
p1.Y = 10;
r.TopLeft = p1;
Point p2 = new Point();
p2.X = 200;
p2.Y = 200;
r.BottomRight = p2;
```

Поначалу синтаксис инициализации объектов может показаться несколько непривычным, но как только вы освоитесь с кодом, то будете приятно поражены тем, насколько быстро можно устанавливать состояние нового объекта с минимальными усилиями.

Исходный код. Проект `ObjectInitializers` доступен в подкаталоге `Chapter_5`.

Работа с данными константных полей

Язык C# предлагает ключевое слово `const` для определения константных данных, которые после начальной установки больше никогда не могут быть изменены. Как и можно было догадаться, оно полезно при определении набора известных значений, логически привязанных к заданному классу или структуре, для использования в приложениях.

Предположим, что вы строите обслуживающий класс по имени `MyMathClass`, в котором нужно определить значение числа π (для простоты будем считать его равным `3.14`). Начнем с создания нового проекта консольного приложения по имени `ConstData`. Учитывая, что вы не хотите давать возможность другим разработчикам изменять это значение в коде, число π можно смоделировать с помощью следующей константы:

```
namespace ConstData
{
  class MyMathClass
  {
    public const double PI = 3.14;
  }
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Fun with Const *****\n");
      Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
      // Ошибка! Константу изменять нельзя!
      // MyMathClass.PI = 3.1444;
      Console.ReadLine();
    }
  }
}
```

Обратите внимание, что ссылка на константные данные, определенные в классе `MyMathClass`, производится с применением префикса в виде имени класса (т.е. `MyMathClass.PI`). Причина в том, что константные поля класса являются неявно *статическими*. Однако допустимо определять локальные константные данные и обращаться к ним внутри области действия метода или свойства, например:

```
static void LocalConstStringVariable()
{
  // Доступ к локальным константным данным можно получать напрямую.
  const string fixedStr = "Fixed string Data";
  Console.WriteLine(fixedStr);
  // Ошибка!
  // fixedStr = "This will not work!";
}
```

Независимо от того, где вы определяете константную часть данных, всегда помните о том, что начальное значение, присваиваемое константе, должно быть указано в момент ее определения. Следовательно, если вы модифицируете класс `MyMathClass` так, чтобы значение `PI` присваивалось внутри конструктора класса, то получите ошибку на этапе компиляции:

```
class MyMathClass
{
  // Попытка установить PI в конструкторе?
  public const double PI;
  public MyMathClass()
  {
    // Невозможно - присваивание должно осуществляться в момент объявления.
    PI = 3.14;
  }
}
```

Такое ограничение связано с тем фактом, что значение константных данных должно быть известно на этапе компиляции. Как известно, конструкторы (или любые другие методы) вызываются во время выполнения.

Понятие полей, допускающих только чтение

С константными данными тесно связано понятие *полей данных, доступных только для чтения* (которое не следует путать со свойствами только для чтения). Подобно константе поле только для чтения не может быть изменено после первоначального присваивания. Тем не менее, в отличие от константы значение, присваиваемое такому полю, может быть определено во время выполнения и потому может на законном основании присваиваться внутри конструктора, но больше нигде.

Поле только для чтения полезно в ситуации, когда значение не известно вплоть до момента выполнения (возможно, из-за того, что для его получения необходимо прочитать внешний файл), но нужно гарантировать, что впоследствии оно не будет изменяться. В целях иллюстрации рассмотрим следующее изменение в классе `MyMathClass`:

```
class MyMathClass
{
  // Поля только для чтения могут присваиваться
  // в конструкторах, но больше нигде.
  public readonly double PI;
  public MyMathClass ()
  {
    PI = 3.14;
  }
}
```

Любая попытка выполнить присваивание полю, помеченному как `readonly`, за пределами конструктора приведет к ошибке на этапе компиляции:

```
class MyMathClass
{
  public readonly double PI;
  public MyMathClass ()
  {
    PI = 3.14;
  }
  // Ошибка!
  public void ChangePI()
  { PI = 3.14444;}
}
```

Статические поля, допускающие только чтение

В отличие от константных полей поля, допускающие только чтение, не являются неявно статическими. Таким образом, если необходимо предоставить доступ к `PI` на уровне класса, то придется явно использовать ключевое слово `static`. Если значение статического поля только для чтения известно на этапе компиляции, то начальное присваивание выглядит очень похожим на такое присваивание в случае константы (однако в этой ситуации проще применить ключевое слово `const`, потому что поле данных присваивается в момент его объявления):

```
class MyMathClass
{
  public static readonly double PI = 3.14;
}
class Program
{
  static void Main(string[] args)
  {
```

```
    Console.WriteLine("***** Fun with Const *****");
    Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
    Console.ReadLine();
  }
}
```

Тем не менее, если значение статического поля только для чтения не известно вплоть до времени выполнения, то должен использоваться статический конструктор, как было описано ранее в главе:

```
class MyMathClass
{
  public static readonly double PI;

  static MyMathClass()
  { PI = 3.14; }
}
```

Исходный код. Проект `ConstData` доступен в подкаталоге `Chapter_5`.

Понятие частичных классов

Последняя, но от этого не менее важная тема связана с пониманием роли ключевого слова `partial` языка C#. Класс производственного уровня вполне может насчитывать многие сотни (если не тысячи) строк кода внутри единственного файла `*.cs`. Как обнаруживается, при создании классов нередко порядочная часть стереотипного кода, будучи однажды обдуманной, может по существу игнорироваться. Например, поля данных, свойства и конструкторы обычно остаются неизменными во время эксплуатации, в то время как методы имеют тенденцию довольно часто пересматриваться из-за обновленных алгоритмов и т.п.

В языке C# одиночный класс можно разносить по нескольким файлам кода, чтобы изолировать стереотипный код от более полезных (и сложных) членов. Чтобы продемонстрировать ситуацию, когда частичные классы могут быть удобными, загрузим ранее созданный проект `EmployeeApp` в Visual Studio и откроем файл `Employee.cs` для редактирования. Как вы помните, этот единственный файл содержит код для всех аспектов класса:

```
class Employee
{
  // Поля данных
  // Конструкторы
  // Методы
  // Свойства
}
```

Применяя частичные классы, вы могли бы перенести (к примеру) свойства, конструкторы и поля данных в новый файл по имени `Employee.Core.cs` (имя файла к делу не относится). Первый шаг предусматривает добавление ключевого слова `partial` к текущему определению класса и вырезание кода, подлежащего помещению в новый файл:

```
// Employee.cs
partial class Employee
{
  // Методы
  // Свойства
}
```

Далее предположив, что к проекту был добавлен новый файл класса, в него можно переместить поля данных и конструкторы с помощью простой операции вырезания и вставки. Кроме того, вы *должны* добавить ключевое слово `partial` к этому аспекту определения класса. Вот пример:

```
// Employee.Core.cs
partial class Employee
{
  // Поля данных
  // Конструкторы
}
```

На заметку! Не забывайте, что каждый аспект определения частичного класса должен быть помечен ключевым словом `partial`!

После компиляции модифицированного проекта вы не должны заметить вообще никакой разницы. Вся идея, положенная в основу частичного класса, касается только этапа проектирования. Как только приложение скомпилировано, в сборке оказывается один целостный класс. Единственное требование при определении частичных классов связано с тем, что разные части должны иметь одно и то же имя класса и находиться внутри того же самого пространства имен .NET.

Сценарии использования для частичных классов?

Теперь, когда вы понимаете механизм определения частичного класса, вас может интересовать, когда (и почему) может понадобиться делать это. Откровенно говоря, определения частичных классов применяются не слишком часто. Однако среда Visual Studio постоянно использует их в фоновом режиме. Например, если вы строите графический пользовательский интерфейс с применением инфраструктуры Windows Presentation Foundation (WPF), то заметите, что Visual Studio помещает весь сгенерированный визуальным конструктором код в отдельный файл частичного класса, позволяя вам сосредоточиться на специальной программной логике (не отвлекаясь на код, сгенерированный конструктором).

Исходный код. Проект `EmployeeAppPartial` доступен в подкаталоге `Chapter_5`.

Резюме

Целью этой главы было ознакомление вас с ролью типа класса C#. Вы видели, что классы могут иметь любое количество *конструкторов*, которые позволяют пользователю объекта устанавливать состояние объекта при его создании. В главе также было проиллюстрировано несколько приемов проектирования классов (и связанных с ними ключевых слов). Вспомните, что ключевое слово `this` используется для получения доступа к текущему объекту, ключевое слово `static` дает возможность определять поля и члены, привязанные к уровню класса (в отличие от объекта), а ключевое слово `const` и модификатор `readonly` позволяют определять элементы данных, которые никогда не изменяются после первоначальной установки.

Большая часть главы была посвящена деталям первого принципа ООП — инкапсуляции. Вы узнали о модификаторах доступа C# и роли свойств типа, о синтаксисе инициализации объектов и о частичных классах. Теперь вы готовы перейти к следующей главе, в которой рассказывается о построении семейства взаимосвязанных классов с применением наследования и полиморфизма.

ГЛАВА 6

Наследование и полиморфизм

В главе 5 рассматривался первый основной принцип объектно-ориентированного программирования (ООП) — инкапсуляция. Вы узнали, как строить отдельный четко определенный тип класса с конструкторами и разнообразными членами (полями, свойствами, методами, константами и полями только для чтения). В настоящей главе мы сосредоточим внимание на оставшихся двух принципах ООП: наследовании и полиморфизме.

Прежде всего, вы научитесь строить семейства связанных классов с применением *наследования*. Как будет показано, эта форма многократного использования кода позволяет определять в родительском классе общую функциональность, которая может быть задействована, а возможно и модифицирована в дочерних классах. По ходу изложения вы узнаете, как устанавливать *полиморфный интерфейс* в иерархиях классов, используя виртуальные и абстрактные члены, а также о роли явного приведения.

Глава завершится исследованием роли изначального родительского класса в библиотеках базовых классов .NET — `System.Object`.

Базовый механизм наследования

Вспомните из главы 5, что *наследование* — это аспект ООП, упрощающий повторное использование кода. Говоря более точно, встречаются две разновидности повторного использования кода: наследование (отношение "является") и модель включения/делегации (отношение "имеет"). Давайте начнем эту главу с рассмотрения классической модели наследования, т.е. отношения "является".

Когда вы устанавливаете между классами отношение "является", то тем самым строите зависимость между двумя или более типами классов. Основная идея, лежащая в основе классического наследования, состоит в том, что новые классы могут создаваться с применением существующих классов в качестве отправной точки. В качестве простого примера создадим новый проект консольного приложения по имени `BasicInheritance`. Предположим, что спроектирован класс `Car`, который моделирует ряд базовых деталей автомобиля:

```
// Простой базовый класс.
class Car
{
  public readonly int maxSpeed;
  private int currSpeed;
  public Car(int max)
  {
    maxSpeed = max;
  }
```

```
  public Car()
  {
    maxSpeed = 55;
  }
  public int Speed
  {
    get { return currSpeed; }
    set
    {
      currSpeed = value;
      if (currSpeed > maxSpeed)
      {
        currSpeed = maxSpeed;
      }
    }
  }
}
```

Обратите внимание, что класс `Car` использует службы инкапсуляции для управления доступом к закрытому полю `currSpead` посредством открытого свойства по имени `Speed`. В настоящий момент с типом `Car` можно работать следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Basic Inheritance *****\n");

  // Создать объект Car и установить максимальную скорость.
  Car myCar = new Car(80);

  // Установить текущую скорость и вывести ее на консоль.
  myCar.Speed = 50;
  Console.WriteLine("My car is going {0} MPH", myCar.Speed);
  Console.ReadLine();
}
```

Указание родительского класса для существующего класса

Теперь предположим, что планируется построить новый класс по имени `MiniVan`. Подобно базовому классу `Car` вы хотите определить класс `MiniVan` так, чтобы он поддерживал данные для максимальной и текущей скоростей и свойство по имени `Speed`, которое позволило бы пользователю модифицировать состояние объекта. Очевидно, что классы `Car` и `MiniVan` взаимосвязаны; фактически можно сказать, что `MiniVan` *"является"* разновидностью `Car`. Отношение "является" (формально называемое *классическим наследованием*) позволяет строить новые определения классов, которые расширяют функциональность существующих классов.

Существующий класс, который будет служить основой для нового класса, называется *базовым классом*, *суперклассом* или *родительским классом*. Роль базового класса заключается в определении всех общих данных и членов для классов, которые его расширяют. Расширяющие классы формально называются *производными* или *дочерними* классами. В языке C# для установления между классами отношения "является" применяется операция двоеточия в определении класса. Пусть написан следующий новый класс `MiniVan`:

```
// MiniVan "является" Car.
class MiniVan : Car
{
}
```

В настоящее время в этом новом классе никакие члены не определены. Так чего же мы достигли за счет наследования `MiniVan` от базового класса `Car`? Выражаясь просто, объекты `MiniVan` теперь имеют доступ ко всем открытым членам, определенным внутри базового класса.

На заметку! Несмотря на то что конструкторы обычно определяются как открытые, производный класс никогда не наследует конструкторы родительского класса. Конструкторы используются для создания только экземпляра класса, внутри которого они определены.

Учитывая отношение между этими двумя типами классов, класс `MiniVan` можно использовать следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Basic Inheritance *****\n");
  ...
  // Создать объект MiniVan.
  MiniVan myVan = new MiniVan();
  myVan.Speed = 10;
  Console.WriteLine("My van is going {0} MPH",
    myVan.Speed);
  Console.ReadLine();
}
```

Обратите внимание, что хотя в класс `MiniVan` никакие члены не добавлялись, в нем есть прямой доступ к открытому свойству `Speed` родительского класса; тем самым обеспечивается повторное использование кода. Это намного лучше, чем создавать класс `MiniVan`, который имеет те же самые члены, что и класс `Car`, такие как свойство `Speed`. В случае дублирования кода в этих двух классах придется сопровождать два фрагмента кода, что очевидно будет непродуктивным расходом времени.

Всегда помните, что наследование предохраняет инкапсуляцию, а потому следующий код приведет к ошибке на этапе компиляции, т.к. закрытые члены не могут быть доступны через объектную ссылку:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Basic Inheritance *****\n");
  ...
  // Создать объект MiniVan.
  MiniVan myVan = new MiniVan();
    myVan.Speed = 10;
  Console.WriteLine("My van is going {0} MPH",
  myVan.Speed);
  // Ошибка! Доступ к закрытым членам невозможен!
  myVan.currSpeed = 55;
  Console.ReadLine();
}
```

В качестве связанного замечания: даже если класс `MiniVan` определяет собственный набор членов, он по-прежнему не будет располагать возможностью доступа к любым закрытым членам базового класса `Car`. Не забывайте, что закрытые члены доступны *только* внутри класса, в котором они определены. Например, показанный ниже метод в `MiniVan` вызовет ошибку на этапе компиляции:

```
// Класс MiniVan является производным от Car.
class MiniVan : Car
```

```
{
  public void TestMethod()
  {
    // Нормально! Доступ к открытым членам родительского
    // типа в производном типе возможен.
    Speed = 10;

    // Ошибка! Нельзя обращаться к закрытым членам
    // родительского типа из производного типа!
    currSpeed = 10;
  }
}
```

Замечание относительно множества базовых классов

Говоря о базовых классах, важно иметь в виду, что язык C# требует, чтобы отдельно взятый класс имел в точности *один* непосредственный базовый класс. Создать тип класса, который был бы производным напрямую от двух и более базовых классов, невозможно (этот прием, поддерживаемый в неуправляемом языке C++, известен как *множественное наследование*). Попытка создать класс, для которого указаны два непосредственных родительских класса, как продемонстрировано в следующем коде, приведет к ошибке на этапе компиляции:

```
// Недопустимо! Множественное наследование
// классов в языке C# не разрешено!
class WontWork
  : BaseClassOne, BaseClassTwo
{}
```

В главе 8 вы увидите, что платформа .NET позволяет классу или структуре реализовывать любое количество дискретных интерфейсов. Таким способом тип C# может поддерживать несколько линий поведения, одновременно избегая сложностей, которые связаны с множественным наследованием. К слову, в то время как класс может иметь только один непосредственный базовый класс, интерфейс разрешено наследовать от множества других интерфейсов. Применяя такой подход, можно строить развитые иерархии интерфейсов, которые моделируют сложные линии поведения (см. главу 8).

Ключевое слово `sealed`

Язык C# предлагает еще одно ключевое слово, `sealed`, которое предотвращает наследование. Когда класс помечен как `sealed` (запечатанный), компилятор не позволяет создавать классы, производные от него. Например, пусть вы приняли решение о том, что дальнейшее расширение класса `MiniVan` не имеет смысла:

```
// Класс Minivan не может быть расширен!
sealed class MiniVan : Car
{
}
```

Если вы (или ваш коллега) попытаетесь унаследовать от этого класса, то получите ошибку на этапе компиляции:

```
// Ошибка! Нельзя расширять класс, помеченный ключевым словом sealed!
class DeluxeMiniVan
  : MiniVan
{}
```

Чаще всего запечатывание класса имеет наибольший смысл при проектировании обслуживающего класса. Скажем, в пространстве имен `System` определены многочисленные запечатанные классы. В этом несложно убедиться, открыв браузер объектов в Visual Studio (через меню View (Вид)) и выбрав класс `String`, который определен в пространстве имен `System` внутри сборки `mscorlib.dll`. На рис. 6.1 обратите внимание на значок, используемый для обозначения класса `sealed`.

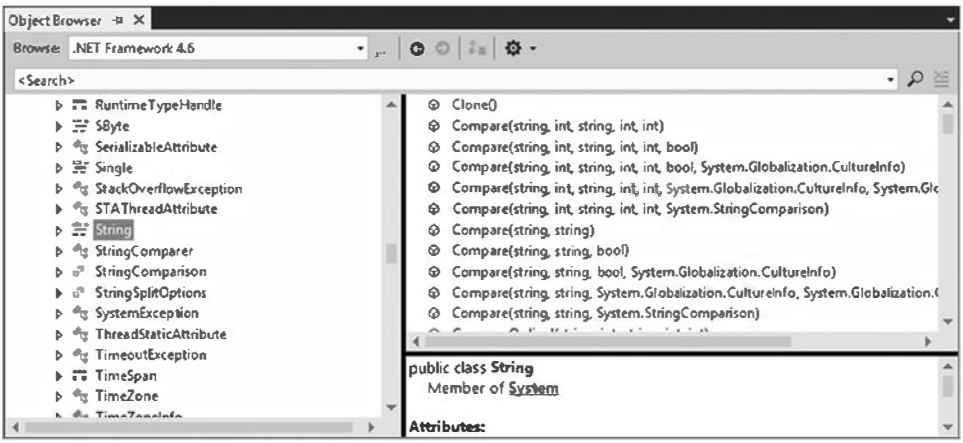


Рис. 6.1. В библиотеках базовых классов определено множество запечатанных классов, таких как `System.String`

Таким образом, как и в случае `MiniVan`, если вы попытаетесь построить новый класс, который расширял бы `System.String`, то получите ошибку на этапе компиляции:

```
// Еще одна ошибка! Нельзя расширять класс, помеченный как sealed!
class MyString
  : String
{}
```

На заметку! В главе 4 вы узнали о том, что структуры C# всегда неявно запечатаны (см. табл. 4.3). Следовательно, создать структуру, производную от другой структуры, класс, производный от структуры, или структуру, производную от класса, невозможно. Структуры могут применяться для моделирования только отдельных, атомарных, определяемых пользователем типов. Если вы хотите задействовать отношение "является", то должны использовать классы.

Как и можно было догадаться, существуют многие другие детали наследования, о которых вы узнаете в оставшемся материале главы. Пока просто примите к сведению, что операция двоеточия позволяет устанавливать отношения "базовый–производный" между классами, в то время как ключевое слово `sealed` предотвращает последующее наследование.

Корректировка диаграмм классов Visual Studio

В главе 2 кратко упоминалось о том, что среда Visual Studio позволяет устанавливать отношения "базовый–производный" между классами визуальным образом во время проектирования. Для работы с этим аспектом IDE-среды сначала понадобится добавить в текущий проект новый файл диаграммы классов. Выберите пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент) и щелкните на значке Class Diagram (Диаграмма классов); на рис. 6.2 этот файл был переименован с `ClassDiagram1.cd` на `Cars.cd`.

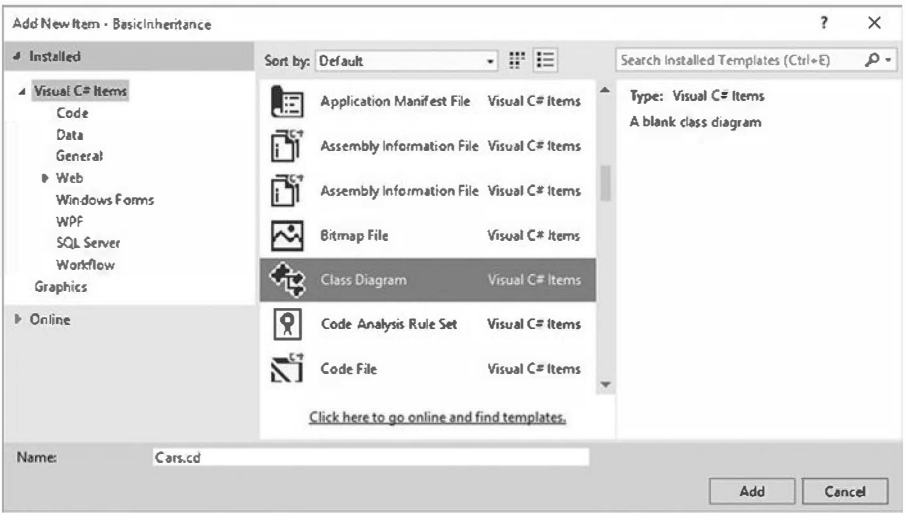


Рис. 6.2. Добавление новой диаграммы классов

После щелчка на кнопке Add (Добавить) отобразится пустая поверхность проектирования. Чтобы добавить типы в визуальный конструктор классов, просто перетащите каждый файл из окна Solution Explorer (Проводник решений) на эту поверхность. Также вспомните, что удаление элемента из визуального конструктора (путем его выбора и нажатия клавиши <Delete>) не приводит к уничтожению ассоциированного с ним исходного кода, а просто убирает элемент из поверхности конструктора. Текущая иерархия классов показана на рис. 6.3.

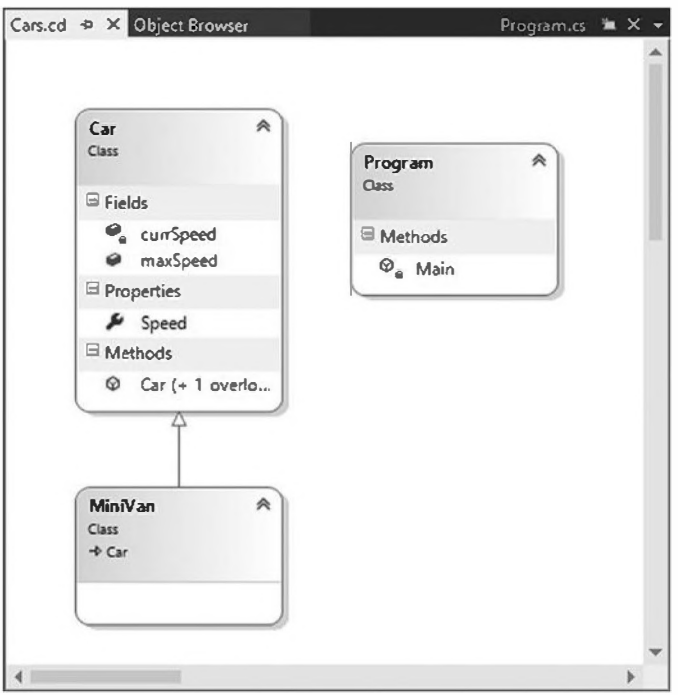


Рис. 6.3. Визуальный конструктор Visual Studio

Как говорилось в главе 2, помимо простого отображения отношений между типами внутри текущего приложения можно также создавать новые типы и наполнять их членами, применяя панель инструментов конструктора классов и окно Class Details (Детали класса).

Если вы хотите использовать указанные визуальные инструменты при проработке оставшихся глав книги, можете делать это. Однако всегда анализируйте сгенерированный код, чтобы четко понимать, что эти инструменты для вас сделали.

Исходный код. Проект `BasicInheritance` доступен в подкаталоге `Chapter_6`.

Второй принцип ООП: детали наследования

Теперь, когда вы видели базовый синтаксис наследования, давайте создадим более сложный пример и рассмотрим многочисленные детали построения иерархий классов. Для этого мы снова обратимся к классу `Employee`, который был спроектирован в главе 5. Первым делом создадим новый проект консольного приложения C# по имени `Employees`.

Далее выберите пункт меню Project⇨Add Existing Item (Проект⇨Добавить существующий элемент) и перейдите к местоположению файлов `Employee.cs` и `Employee.Core.cs`, созданных в примере `EmployeeApp` из главы 5. Выделите оба файла (щелчком при нажатой клавише <Ctrl>) и щелкните на кнопке Add (Добавить). Среда Visual Studio отреагирует копированием каждого файла в текущий проект (имея дело с копиями, вам не придется беспокоиться о том, что исходные файлы проекта из главы 5 случайно будут изменены).

Прежде чем приступить к построению каких-то производных классов, следует уделить внимание одной детали. Поскольку первоначальный класс `Employee` был создан в проекте по имени `EmployeeApp`, он находится внутри идентично названного пространства имен .NET. Пространства имен подробно рассматриваются в главе 14; тем не менее, ради простоты переименуйте текущее пространство имен (в обоих файлах) на `Employees`, чтобы оно совпадало с именем нового проекта:

```
// Не забудьте изменить название пространства имен в обоих файлах C#!
namespace Employees
{
  partial class Employee
  {...}
}
```

На заметку! В качестве проверки работоспособности скомпилируйте и запустите новый проект, нажав <Ctrl+F5>. Пока что программа ничего не делает, но это позволит удостовериться в отсутствии ошибок на этапе компиляции.

Нашей целью является создание семейства классов, которые моделируют разнообразные типы сотрудников в компании. Предположим, что необходимо задействовать функциональность класса `Employee` при создании двух новых классов (`SalesPerson` и `Manager`). Новый класс `SalesPerson` "является" `Employee` (как и `Manager`). Вспомните, что в модели классического наследования базовые классы (вроде `Employee`) обычно применяются для определения характеристик, общих для всех наследников. Подклассы (такие как `SalesPerson` и `Manager`) расширяют эту общую функциональность, добавляя к ней специфическую функциональность.

В настоящем примере мы будем считать, что класс Manager расширяет Employee, сохраняя количество фондовых опционов, тогда как класс SalesPerson поддерживает хранение количества продаж. Добавьте новый файл класса (Manager.cs), в котором определяется класс Manager со следующим автоматическим свойством:

```
// Менеджерам нужно знать количество их фондовых опционов.
class Manager : Employee
{
  public int StockOptions { get; set; }
}
```

Затем добавьте еще один новый файл класса (SalesPerson.cs), в котором определен класс SalesPerson с подходящим автоматическим свойством:

```
// Продавцам нужно знать количество продаж.
class SalesPerson : Employee
{
  public int SalesNumber { get; set; }
}
```

Теперь, когда отношение "является" установлено, классы SalesPerson и Manager автоматически наследуют все открытые члены базового класса Employee. В целях иллюстрации модифицируем метод Main(), как показано ниже:

```
// Создание объекта подкласса и доступ к функциональности базового класса.
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");
  SalesPerson fred = new SalesPerson();
  fred.Age = 31;
  fred.Name = "Fred";
  fred.SalesNumber = 50;
  Console.ReadLine();
}
```

Управление созданием объектов базового класса с помощью ключевого слова base

В настоящий момент объекты классов SalesPerson и Manager могут создаваться только с использованием "бесплатно полученного" стандартного конструктора (см. главу 5). Памятуя об этом, предположим, что в класс Manager добавлен новый конструктор с шестью аргументами, который вызывается следующим образом:

```
static void Main(string[] args)
{
  ...
  // Предположим, что у Manager есть конструктор с такой сигнатурой:
  // (string fullName, int age, int empID,
  // float currPay, string ssn, int numbOfOpts)
  Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
  Console.ReadLine();
}
```

Взглянув на список параметров, легко заметить, что большинство аргументов должно быть сохранено в переменных-членах, определенных в базовом классе Employee. Чтобы сделать это, в классе Manager можно было бы реализовать показанный ниже специальный конструктор:

```
public Manager(string fullName, int age, int empID,
               float currPay, string ssn, int numbOfOpts)
{
  // Это свойство определено в классе Manager.
  StockOptions = numbOfOpts;

  // Присвоить входные параметры, используя
  // унаследованные свойства родительского класса.
  ID = empID;
  Age = age;
  Name = fullName;
  Pay = currPay;

  // Если свойство SSN окажется доступным только для чтения,
  // то здесь возникнет ошибка на этапе компиляции!
  SocialSecurityNumber = ssn;
}
```

Первая проблема с этим подходом в том, что если любое свойство определено как допускающее только чтение (например, свойство `SocialSecurityNumber`), то присвоить значение входного параметра `string` данному полю не удастся, как можно видеть в финальном операторе специального конструктора.

Вторая проблема связана с тем, что был косвенно создан довольно неэффективный конструктор, учитывая тот факт, что в C#, если не указано иначе, стандартный конструктор базового класса вызывается автоматически перед выполнением логики конструктора производного класса. После этого момента текущая реализация имеет доступ к многочисленным открытым свойствам базового класса `Employee` для установки его состояния. Таким образом, во время создания объекта `Manager` на самом деле выполнялось семь действий (обращения к пяти унаследованным свойствам и двум конструкторам).

Для оптимизации создания объектов производного класса необходимо корректно реализовать конструкторы подкласса, чтобы они явно вызывали подходящий специальный конструктор базового класса вместо стандартного конструктора. Подобным образом можно сократить количество вызовов инициализации унаследованных членов (что уменьшит время обработки). Первым делом понадобится обеспечить в родительском классе `Employee` следующий конструктор с пятью параметрами:

```
// Добавить в базовый класс Employee.
public Employee(string name, int age, int id, float pay, string ssn)
:this(name, age, id, pay)
{
  empSSN = ssn;
}
```

Теперь давайте модифицируем специальный конструктор в классе `Manager`, применив ключевое слово `base`:

```
public Manager(string fullName, int age, int empID,
               float currPay, string ssn, int numbOfOpts)
  : base(fullName, age, empID, currPay, ssn)
{
  // Это свойство определено в классе Manager.
  StockOptions = numbOfOpts;
}
```

Здесь ключевое слово `base` ссылается на сигнатуру конструктора (подобно синтаксису, используемому для объединения конструкторов одиночного класса в цепочку посредством ключевого слова `this`, как обсуждалось в главе 5), что всегда указывает

производному конструктору на необходимость передачи данных конструктору непосредственного родительского класса. В этой ситуации явно вызывается конструктор с пятью параметрами, определенный в Employee, что избавляет от излишних обращений во время создания объекта дочернего класса. Специальный конструктор класса SalesPerson выглядит в основном идентично:

```
// В качестве общего правила запомните, что все подклассы должны
// явно вызывать подходящий конструктор базового класса.
public SalesPerson(string fullName, int age, int empID,
  float currPay, string ssn, int numbOfSales)
  : base(fullName, age, empID, currPay, ssn)
{
  // Это принадлежит нам!
  SalesNumber = numbOfSales;
}
```

На заметку! Ключевое слово base можно применять всякий раз, когда подкласс желает обратиться к открытому или защищенному члену, определенному в родительском классе. Использование этого ключевого слова не ограничивается логикой конструктора. Вы увидите примеры применения ключевого слова base в подобной манере позже в главе при рассмотрении полиморфизма.

И, наконец, вспомните, что после добавления к определению класса специального конструктора стандартный конструктор молча удаляется. Следовательно, не забудьте переопределить стандартный конструктор для классов SalesPerson и Manager. Вот пример:

```
// Аналогичным образом переопределите стандартный
// конструктор также и в классе Manager.
public SalesPerson() {}
```

Хранение секретов семейства: ключевое слово protected

Как вы уже знаете, открытые элементы напрямую доступны отовсюду, в то время как закрытые элементы могут быть доступны только в классе, где они определены. Вспомните из главы 5, что C# опережает многие другие современные объектные языки и предоставляет дополнительное ключевое слово для определения доступности членов — protected (защищенный).

Когда базовый класс определяет защищенные данные или защищенные члены, он устанавливает набор элементов, которые могут быть непосредственно доступны любому наследнику. Если вы хотите разрешить дочерним классам SalesPerson и Manager напрямую обращаться к разделу данных, который определен в Employee, измените исходный класс Employee, как показано ниже:

```
// Защищенные данные состояния.
partial class Employee
{
  // Производные классы теперь могут иметь прямой доступ к этой информации.
  protected string empName;
  protected int empID;
  protected float currPay;
  protected int empAge;
  protected string empSSN;
  ...
}
```

Преимущество определения защищенных членов в базовом классе заключается в том, что производным классам больше не придется обращаться к данным косвенно, используя открытые методы и свойства. Разумеется, подходу присущ и недостаток: когда производный класс имеет прямой доступ к внутренним данным своего родителя, есть вероятность непредумышленного обхода существующих бизнес-правил, которые реализованы внутри открытых свойств. Определяя защищенные члены, вы создаете уровень доверия между родительским классом и дочерним классом, т.к. компилятор не будет перехватывать какие-либо нарушения бизнес-правил, предусмотренных для типа.

Наконец, имейте в виду, что с точки зрения пользователя объекта защищенные данные расцениваются как закрытые (поскольку пользователь находится "снаружи" семейства). По этой причине следующий код недопустим:

```
static void Main(string[] args)
{
  // Ошибка! Доступ к защищенным данным из клиентского кода невозможен!
  Employee emp = new Employee();
  emp.empName = "Fred";
}
```

На заметку! Несмотря на то что защищенные поля данных могут нарушить инкапсуляцию, определять защищенные методы вполне безопасно (и полезно). При построении иерархий классов обычно приходится определять набор методов, которые предназначены для применения только производными типами, но не внешним миром.

Добавление запечатанного класса

Вспомните, что *запечатанный* класс не может быть расширен другими классами. Как уже упоминалось, такой прием чаще всего используется при проектировании обслуживающих классов. Тем не менее, при построении иерархий классов вы можете обнаружить, что определенная ветвь в цепочке наследования нуждается в "отсечении", т.к. дальнейшее ее расширение не имеет смысла. Для примера предположим, что вы добавили в приложение еще один класс (`PTSalesPerson`), который расширяет существующий тип `SalesPerson`. На рис. 6.4 показано текущее обновление.

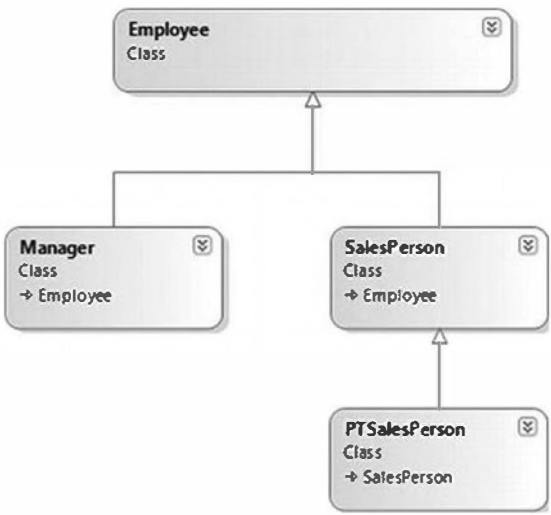


Рис. 6.4. Класс `PTSalesPerson`

Класс PTSalesPerson представляет продавца, который работает на условиях частичной занятости. В качестве варианта скажем, что нужно гарантировать отсутствие возможности создания подкласса PTSalesPerson. (В конце концов, какой смысл в дополнительной "частичной занятости" от имеющейся "частичной занятости"?) Чтобы предотвратить наследование от класса, необходимо применить ключевое слово sealed:

```
sealed class PTSalesPerson : SalesPerson
{
  public PTSalesPerson(string fullName, int age, int empID,
                       float currPay, string ssn, int numbOfSales)
    :base (fullName, age, empID, currPay, ssn, numbOfSales)
  {
  }
  // Остальные члены класса...
}
```

Реализация модели включения/делегации

Вы уже знаете, что повторное использование кода встречается в двух видах. Только что было продемонстрировано классическое отношение "является". Перед тем, как мы начнем исследование третьего принципа ООП (полиморфизма), давайте взглянем на отношение "имеет" (также известное как *модель включения/делегации* или *агрегация*). Предположим, что создан новый класс, который моделирует пакет льгот для сотрудников:

```
// Этот новый тип будет функционировать как включаемый класс.
class BenefitPackage
{
  // Предположим, что есть другие члены, представляющие
  // медицинские/стоматологические программы и т.д.
  public double ComputePayDeduction()
  {
    return 125.0;
  }
}
```

Очевидно, что было бы довольно странно устанавливать отношение "является" между классом BenefitPackage и типами сотрудников. (Разве Employee "является" BenefitPackage? Вряд ли.) Однако должно быть ясно, что какое-то отношение между ними должно быть установлено. Короче говоря, нужно выразить идею о том, что каждый сотрудник "имеет" BenefitPackage. Для этого можно модифицировать определение класса Employee следующим образом:

```
// Теперь сотрудники имеют льготы.
partial class Employee
{
  // Содержит объект BenefitPackage.
  protected BenefitPackage empBenefits = new BenefitPackage();
  ...
}
```

На этой стадии вы имеете объект, который благополучно содержит в себе другой объект. Тем не менее, открытие доступа к функциональности содержащегося объекта внешнему миру требует делегации. *Делегация* — это просто действие по добавлению во включающий класс открытых членов, которые работают с функциональностью содержащегося внутри объекта.

Например, можно было бы изменить класс Employee так, чтобы он открывал доступ к включенному объекту empBenefits с применением специального свойства, а также использовать его функциональность внутренне посредством нового метода по имени GetBenefitCost():

```
partial class Employee
{
  // Содержит объект BenefitPackage.
  protected BenefitPackage empBenefits = new BenefitPackage();

  // Открывает доступ к некоторому поведению, связанному со льготами.
  public double GetBenefitCost()
  { return empBenefits.ComputePayDeduction(); }

  // Открывает доступ к объекту через специальное свойство.
  public BenefitPackage Benefits
  {
    get { return empBenefits; }
    set { empBenefits = value; }
  }
  ...
}
```

В следующем обновленном методе Main() обратите внимание на взаимодействие с внутренним типом BenefitsPackage, который определен в типе Employee:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");
  ...
  Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
  double cost = chucky.GetBenefitCost();
  Console.ReadLine();
}
```

Определения вложенных типов

В главе 5 кратко упоминалась концепция вложенных типов, которая является развитием только что рассмотренного отношения "имеет". В C# (а также в других языках .NET) допускается определять тип (перечисление, класс, интерфейс, структуру или делегат) прямо внутри области действия класса либо структуры. В таком случае вложенный (или "внутренний") тип считается членом охватывающего (или "внешнего") типа, и в глазах исполняющей системы им можно манипулировать как любым другим членом (полем, свойством, методом и событием). Синтаксис, применяемый для вложения типа, достаточно прост:

```
public class OuterClass
{
  // Открытый вложенный тип может использоваться кем угодно.
  public class PublicInnerClass {}

  // Закрытый вложенный тип может использоваться
  // только членами включающего класса.
  private class PrivateInnerClass {}
}
```

Хотя синтаксис довольно ясен, ситуации, в которых это может понадобиться, не настолько очевидны. Для того чтобы понять данный прием, рассмотрим характерные черты вложенных типов.

- Вложенные типы позволяют получить полный контроль над уровнем доступа внутреннего типа, потому что они могут быть объявлены как закрытые (вспомните, что невложенные классы нельзя объявлять с ключевым словом `private`).
- Поскольку вложенный тип является членом включающего класса, он может иметь доступ к закрытым членам этого включающего класса.
- Часто вложенный тип полезен только как вспомогательный для внешнего класса, и не предназначен для использования во внешнем мире.

Когда тип включает в себя другой тип класса, он может создавать переменные-члены этого типа, как он бы делал это для любого другого элемента данных. Однако если планируется работать с вложенным типом за пределами включающего типа, то он должен быть уточнен именем включающего типа. Взгляните на такой код:

```
static void Main(string[] args)
{
  // Создать и использовать объект открытого вложенного класса. Нормально!
  OuterClass.PublicInnerClass inner;
  inner = new OuterClass.PublicInnerClass();

  // Ошибка на этапе компиляции! Доступ к закрытому вложенному классу невозможен!
  OuterClass.PrivateInnerClass inner2;
  inner2 = new OuterClass.PrivateInnerClass();
}
```

Чтобы применить эту концепцию в примере с сотрудниками, предположим, что определение `BenefitPackage` теперь вложено непосредственно в класс `Employee`:

```
partial class Employee
{
  public class BenefitPackage
  {
    // Предположим, что есть другие члены, представляющие
    // медицинские/стоматологические программы и т.д.
    public double ComputePayDeduction()
    {
      return 125.0;
    }
  }
  ...
}
```

Процесс вложения может распространяться настолько "глубоко", насколько это требуется. Например, пусть необходимо создать перечисление по имени `BenefitPackageLevel`, документирующее разнообразные уровни льгот, из которых может выбирать сотрудник. Чтобы программно обеспечить тесную связь между типами `Employee`, `BenefitPackage` и `BenefitPackageLevel`, перечисление можно вложить следующим образом:

```
// В класс Employee вложен класс BenefitPackage.
public partial class Employee
{
  // В класс BenefitPackage вложено перечисление BenefitPackageLevel.
  public class BenefitPackage
  {
    public enum BenefitPackageLevel
    {
      Standard, Gold, Platinum
    }
```

```
    public double ComputePayDeduction()
    {
      return 125.0;
    }
  }
  ...
}
```

Взгляните, как приходится использовать это перечисление из-за отношений вложения:

```
static void Main(string[] args)
{
  ...
  // Определить уровень льгот.
  Employee.BenefitPackage.BenefitPackageLevel myBenefitLevel =
    Employee.BenefitPackage.BenefitPackageLevel.Platinum;
  Console.ReadLine();
}
```

Итак, к настоящему моменту вы ознакомились с несколькими ключевыми словами (и концепциями), которые позволяют строить иерархии типов, связанных посредством классического наследования, включения и вложения. Не беспокойтесь, если ясны еще не все детали. На протяжении оставшихся глав книги будет построено еще немало иерархий. А теперь давайте перейдем к исследованию последнего принципа ООП — полиморфизма.

Третий принцип ООП: поддержка полиморфизма в C#

Вспомните, что в базовом классе `Employee` определен метод по имени `GiveBonus()`, который первоначально был реализован так:

```
public partial class Employee
{
  public void GiveBonus(float amount)
  {
    Pay += amount;
  }
  ...
}
```

Поскольку этот метод был определен с ключевым словом `public`, теперь можно раздавать бонусы продавцам и менеджерам (а также продавцам с частичной занятостью):

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");

  // Выдать каждому сотруднику бонус?
  Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
  chucky.GiveBonus(300);
  chucky.DisplayStats();
  Console.WriteLine();

  SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
  fran.GiveBonus(200);
  fran.DisplayStats();
  Console.ReadLine();
}
```

Проблема с текущим проектным решением заключается в том, что открыто унаследованный метод `GiveBonus()` функционирует идентично для всех подклассов. В идеале при подсчете бонуса для штатного продавца и частично занятого продавца должно приниматься во внимание количество продаж. Возможно, менеджеры должны получать дополнительные фондовые опционы вместе с денежным вознаграждением. Учитывая это, вы однажды столкнетесь с интересным вопросом: "Как сделать так, чтобы связанные типы реагировали по-разному на один и тот же запрос?". Попробуем найти на него ответ.

Ключевые слова `virtual` и `override`

Полиморфизм предоставляет подклассу способ определения собственной версии метода, определенного в его базовом классе, с применением процесса, который называется *переопределением метода*. Чтобы модернизировать текущее проектное решение, необходимо понимать смысл ключевых слов `virtual` и `override`. Если базовый класс желает определить метод, который *может быть* (но не обязательно) переопределен в подклассе, то он должен пометить его ключевым словом `virtual`:

```
partial class Employee
{
  // Теперь этот метод может быть переопределен в производном классе.
  public virtual void GiveBonus(float amount)
  {
    Pay += amount;
  }
  ...
}
```

На заметку! Методы, помеченные ключевым словом `virtual`, называются *виртуальными методами*.

Когда подкласс желает изменить реализацию деталей виртуального метода, он делает это с помощью ключевого слова `override`. Например, классы `SalesPerson` и `Manager` могли бы переопределять метод `GiveBonus()`, как показано ниже (предположим, что класс `PTSalesPerson` не будет переопределять `GiveBonus()`, а потому просто наследует версию, определенную в `SalesPerson`):

```
class SalesPerson : Employee
{
  ...
  // Бонус продавца зависит от количества продаж.
  public override void GiveBonus(float amount)
  {
    int salesBonus = 0;
    if (SalesNumber >= 0 && SalesNumber <= 100)
      salesBonus = 10;
    else
    {
      if (SalesNumber >= 101 && SalesNumber <= 200)
        salesBonus = 15;
      else
        salesBonus = 20;
    }
    base.GiveBonus(amount * salesBonus);
  }
}
```

```
class Manager : Employee
{
  ...
  public override void GiveBonus(float amount)
  {
    base.GiveBonus(amount);
    Random r = new Random();
    StockOptions += r.Next(500);
  }
}
```

Обратите внимание, что каждый переопределенный метод может задействовать стандартное поведение посредством ключевого слова `base`.

Таким образом, полностью повторять реализацию логики метода `GiveBonus()` вовсе не обязательно, а взамен можно повторно использовать (и расширять) стандартное поведение родительского класса.

Также предположим, что текущий метод `DisplayStatus()` класса `Employee` объявлен виртуальным:

```
public virtual void DisplayStats()
{
  Console.WriteLine("Name: {0}", Name);
  Console.WriteLine("ID: {0}", ID);
  Console.WriteLine("Age: {0}", Age);
  Console.WriteLine("Pay: {0}", Pay);
  Console.WriteLine("SSN: {0}", SocialSecurityNumber);
}
```

Тогда каждый подкласс может переопределять этот метод с целью отображения количества продаж (для продавцов) и текущих фондовых опционов (для менеджеров). Например, рассмотрим версию метода `DisplayStatus()` из класса `Manager` (класс `SalesPerson` реализовывал бы метод `DisplayStatus()` в похожей манере, выводя на консоль количество продаж):

```
public override void DisplayStats()
{
  base.DisplayStats();
  Console.WriteLine("Number of Stock Options: {0}", StockOptions);
}
```

Теперь, когда каждый подкласс может истолковывать эти виртуальные методы значащим для него образом, их экземпляры ведут себя как более независимые сущности:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");

  // Лучшая система бонусов!
  Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
  chucky.GiveBonus(300);
  chucky.DisplayStats();
  Console.WriteLine();

  SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
  fran.GiveBonus(200);
  fran.DisplayStats();
  Console.ReadLine();
}
```

Вот результат тестового запуска приложения в его текущем виде:

```
***** The Employee Class Hierarchy *****

Name: Chucky
ID: 92
Age: 50
Pay: 100300
SSN: 333-23-2322
Number of Stock Options: 9337

Name: Fran
ID: 93
Age: 43
Pay: 5000
SSN: 932-32-3232
Number of Sales: 31
```

Переопределение виртуальных членов в IDE-среде Visual Studio

Вы наверняка заметили, что при переопределении члена класса потребуется вспомнить тип каждого параметра, не говоря уже об имени метода и соглашениях по передаче параметров (`ref`, `out` и `params`). В Visual Studio доступно полезное средство IntelliSense, к которому можно обращаться при переопределении виртуального члена. Если вы наберете слово `override` внутри области действия типа класса (и затем нажмете клавишу пробела), то IntelliSense автоматически отобразит список всех допускающих переопределение членов родительского класса (рис. 6.5).

Если вы выберете член и нажмете клавишу <Enter>, то IDE-среда отреагирует автоматическим заполнением заглушки метода. Обратите внимание, что вы также получаете оператор кода, который вызывает родительскую версию виртуального члена (можете удалить эту строку, если она не нужна).

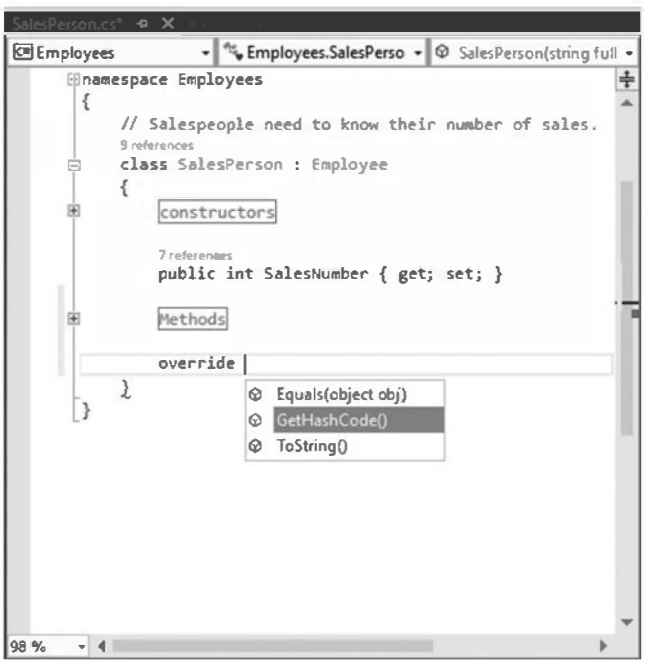


Рис. 6.5. Быстрый просмотр переопределяемых методов в Visual Studio

Например, используя этот прием при переопределении метода DisplayStatus(), вы обнаружите следующий автоматически сгенерированный код:

```
public override void DisplayStats()
{
  base.DisplayStats();
}
```

Запечатывание виртуальных членов

Вспомните, что к типу класса можно применять ключевое слово sealed, чтобы предотвратить расширение его поведения другими типами через наследование. Ранее класс PTSalesPerson был запечатан на основе предположения о том, что разработчикам не имеет смысла дальше расширять эту линию наследования.

Следует отметить, что временами желательно не запечатывать класс целиком, а просто предотвратить переопределение некоторых виртуальных методов в производных типах. Для примера предположим, что вы не хотите, чтобы продавцы с частичной занятостью получали специальные бонусы. Предотвратить переопределение виртуального метода GiveBonus() в классе PTSalesPerson можно, запечатав этот метод в классе SalesPerson:

```
// Класс SalesPerson запечатал метод GiveBonus()!
class SalesPerson : Employee
{
  ...
  public override sealed void GiveBonus(float amount)
  {
    ...
  }
}
```

Здесь класс SalesPerson на самом деле переопределяет виртуальный метод GiveBonus(), определенный в Employee, но явно помечает его как sealed. Таким образом, попытка переопределения этого метода в классе PTSalesPerson приведет к ошибке на этапе компиляции:

```
sealed class PTSalesPerson : SalesPerson
{
  public PTSalesPerson(string fullName, int age, int empID,
                       float currPay, string ssn, int numbOfSales)
    :base (fullName, age, empID, currPay, ssn, numbOfSales)
  {
  }

  // Ошибка на этапе компиляции! Переопределять этот метод
  // в классе PTSalesPerson нельзя, т.к. он был запечатан.
  public override void GiveBonus(float amount)
  {
  }
}
```

Абстрактные классы

В настоящий момент базовый класс Employee спроектирован так, что поставляет различные данные-члены своим наследникам, а также предлагает два виртуальных метода (GiveBonus() и DisplayStatus()), которые могут быть переопределены в наследниках. Хотя все это замечательно, у такого проектного решения имеется один весьма

странный побочный эффект: можно напрямую создавать экземпляры базового класса Employee:

```
// Что это будет означать?
Employee X = new Employee();
```

В нашем примере базовый класс Employee служит единственной цели — определять общие члены для всех подклассов. По всем признакам мы не намерены позволять кому-либо создавать непосредственные экземпляры этого класса, т.к. сам тип Employee является концептуально чересчур общим. Например, если кто-то скажет: "Я сотрудник!", то тут же возникнет вопрос: "Сотрудник какого *рода*?" (консультант, инструктор, административный работник, редактор, советник в правительстве и т.п.).

Учитывая, что многие базовые классы имеют тенденцию быть довольно расплывчатыми сущностями, намного более эффективным проектным решением для данного примера будет предотвращение возможности непосредственного создания в коде нового объекта Employee. В C# этого можно добиться за счет использования ключевого слова abstract в определении класса, создавая в итоге *абстрактный базовый класс*:

```
// Превращение класса Employee в абстрактный для
// предотвращения прямого создания его экземпляров.
abstract partial class Employee
{
  ...
}
```

Теперь попытка создания экземпляра класса Employee приводит к ошибке на этапе компиляции:

```
// Ошибка! Нельзя создавать экземпляр абстрактного класса!
Employee X = new Employee();
```

Определение класса, экземпляры которого нельзя создавать напрямую, на первый взгляд может показаться странным. Однако вспомните, что базовые классы (абстрактные или нет) полезны тем, что содержат все общие данные и функциональность для производных типов. Такая форма абстракции дает возможность считать, что "идея" сотрудника является полностью допустимой, просто это не конкретная сущность. Кроме того, необходимо понимать, что хотя *непосредственно* создавать экземпляры абстрактного класса невозможно, они все равно появляются в памяти при создании экземпляров производных классов. Таким образом, для абстрактных классов вполне нормально (и общепринято) определять любое количество конструкторов, которые вызываются *косвенно*, когда выделяется память под экземпляры производных классов.

На этой стадии у нас есть довольно интересная иерархия сотрудников. Мы добавим чуть больше функциональности к этому приложению позже, при рассмотрении правил приведения типов C#. А пока на рис. 6.6 представлено текущее проектное решение.

Исходный код. Проект Employees доступен в подкаталоге Chapter_6.

Полиморфные интерфейсы

Когда класс определен как абстрактный базовый (посредством ключевого слова abstract), в нем может определяться любое число *абстрактных членов*. Абстрактные члены могут применяться везде, где требуется определить член, который *не* предоставляет стандартной реализации, но *должен* приниматься во внимание каждым производным классом. Тем самым вы навязываете *полиморфный интерфейс* каждому наследнику, оставляя им задачу реализации конкретных деталей абстрактных методов.

Выражаясь упрощенно, полиморфный интерфейс абстрактного базового класса просто ссылается на его набор виртуальных и абстрактных методов. На самом деле это намного интереснее, чем может показаться на первый взгляд, поскольку данная характерная черта ООП позволяет строить легко расширяемые и гибкие приложения. В целях иллюстрации мы реализуем (и слегка модифицируем) иерархию фигур, кратко описанную в главе 5 во время обзора основных принципов ООП. Для начала создадим новый проект консольного приложения C# по имени `Shapes`.

На рис. 6.7 обратите внимание на то, что типы `Hexagon` и `Circle` расширяют базовый класс `Shape`. Как и любой базовый класс, `Shape` определяет набор членов (в этом случае свойство `PetName` и метод `Draw()`), общих для всех наследников.

Во многом подобно иерархии классов сотрудников вы должны иметь возможность запретить создание экземпляров класса `Shape` напрямую, потому что он представляет слишком абстрактную концепцию. Чтобы предотвратить непосредственное создание экземпляров класса `Shape`, его можно определить как абстрактный класс. К тому же, учитывая, что производные типы должны уникальным образом реагировать на вызов метода `Draw()`, давайте пометим его как `virtual` и определим стандартную реализацию.

```
// Абстрактный базовый класс иерархии.
abstract class Shape
{
  public Shape(string name = "NoName")
  { PetName = name; }

  public string PetName { get; set; }

  // Единственный виртуальный метод.
  public virtual void Draw()
  {
    Console.WriteLine(
        "Inside Shape.Draw()");
  }
}
```

Обратите внимание, что виртуальный метод `Draw()` предоставляет стандартную реализацию, которая просто выводит на

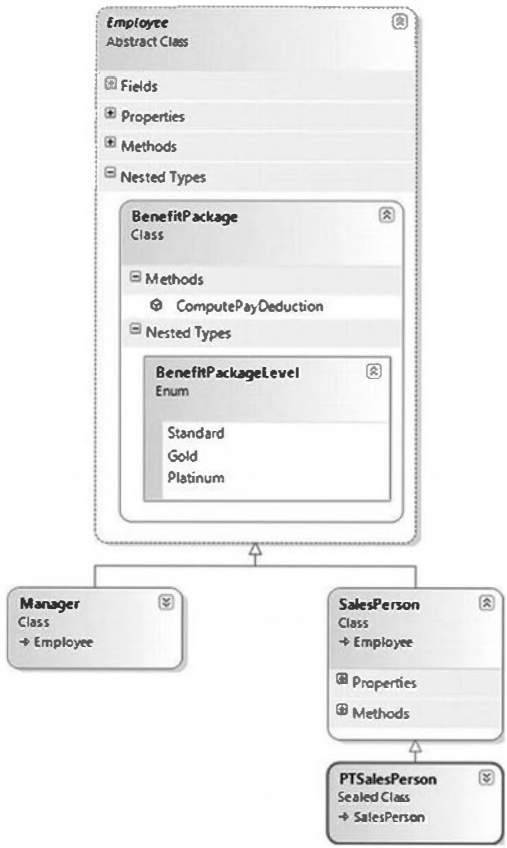


Рис. 6.6. Иерархия классов `Employee`

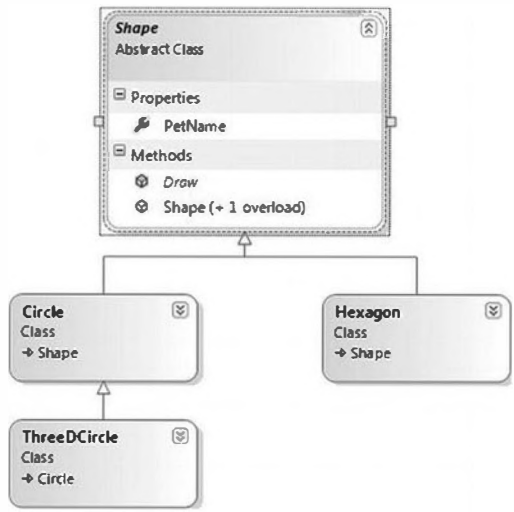


Рис. 6.7. Иерархия классов фигур

консоль сообщение, информирующее о том, что вызван метод `Draw()` из базового класса `Shape`. Теперь вспомните, что когда метод помечен ключевым словом `virtual`, он поддерживает стандартную реализацию, которую автоматически наследуют все производные типы. Если дочерний класс так решит, то он *может* переопределить такой метод, но он не *обязан* это делать. Рассмотрим показанную ниже реализацию типов `Circle` и `Hexagon`:

```
// Класс Circle не переопределяет метод Draw().
class Circle : Shape
{
  public Circle() {}
  public Circle(string name) : base(name){}
}

// Класс Hexagon переопределяет метод Draw().
class Hexagon : Shape
{
  public Hexagon() {}
  public Hexagon(string name) : base(name){}
  public override void Draw()
  {
    Console.WriteLine("Drawing {0} the Hexagon", PetName);
  }
}
```

Полезность абстрактных методов становится совершенно ясной, как только вы снова вспомните, что подклассы *никогда не обязаны* переопределять виртуальные методы (как в случае `Circle`). Следовательно, если создать экземпляры типов `Hexagon` и `Circle`, то обнаружится, что `Hexagon` знает, как правильно "рисовать" себя (или, по крайней мере, выводить на консоль подходящее сообщение). Тем не менее, реакция `Circle` порядком сбивает с толку.

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Polymorphism *****\n");

  Hexagon hex = new Hexagon("Beth");
  hex.Draw();
  Circle cir = new Circle("Cindy");
  // Вызывает реализацию базового класса!
  cir.Draw();
  Console.ReadLine();
}
```

Взгляните на вывод этого метода `Main()`:

```
***** Fun with Polymorphism *****

Drawing Beth the Hexagon
Inside Shape.Draw()
```

Очевидно, что это не самое разумное проектное решение для текущей иерархии. Чтобы вынудить каждый дочерний класс переопределять метод `Draw()`, его можно определить как абстрактный метод класса `Shape`, а это значит, что какая-либо стандартная реализация вообще не предоставляется. Для пометки метода как абстрактного в C# используется ключевое слово `abstract`. Обратите внимание, что абстрактные методы не предоставляют никакой реализации:

```
abstract class Shape
{
  // Вынудить все дочерние классы определять способ своей визуализации.
  public abstract void Draw();
  ...
}
```

На заметку! Абстрактные методы могут быть определены только в абстрактных классах, иначе возникнет ошибка на этапе компиляции.

Методы, помеченные как `abstract`, являются чистым протоколом. Они просто определяют имя, возвращаемый тип (если есть) и набор параметров (при необходимости). Здесь абстрактный класс `Shape` информирует производные типы о том, что у него имеется метод по имени `Draw()`, который не принимает аргументов и ничего не возвращает. О необходимых деталях должен позаботиться производный класс.

С учетом этого метод `Draw()` в классе `Circle` теперь должен быть обязательно переопределен. В противном случае `Circle` также должен быть абстрактным классом и декорироваться ключевым словом `abstract` (что очевидно не подходит в настоящем примере). Вот изменения в коде:

```
// Если не реализовать здесь абстрактный метод Draw(), то Circle
// также должен считаться абстрактным и быть помечен как abstract!
class Circle : Shape
{
  public Circle() {}
  public Circle(string name) : base(name) {}
  public override void Draw()
  {
    Console.WriteLine("Drawing {0} the Circle", PetName);
  }
}
```

Итак, теперь можно предполагать, что любой класс, производный от `Shape`, действительно имеет уникальную версию метода `Draw()`. Для демонстрации полной картины полиморфизма рассмотрим следующий код:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Polymorphism *****\n");
  // Создать массив совместимых с Shape объектов.
  Shape[] myShapes = {new Hexagon(), new Circle(), new Hexagon("Mick"),
    new Circle("Beth"), new Hexagon("Linda")};
  // Пройти в цикле по всем элементам и взаимодействовать
  // с полиморфным интерфейсом.
  foreach (Shape s in myShapes)
  {
    s.Draw();
  }
  Console.ReadLine();
}
```

Ниже показан вывод из этого метода `Main()`:

```
***** Fun with Polymorphism *****

Drawing NoName the Hexagon
Drawing NoName the Circle
Drawing Mick the Hexagon
Drawing Beth the Circle
Drawing Linda the Hexagon
```

Данный метод `Main()` иллюстрирует полиморфизм в чистом виде. Хотя *напрямую* создавать экземпляры абстрактного базового класса (`Shape`) невозможно, с помощью абстрактной базовой переменной допускается хранить ссылки на объекты любого подкласса. Таким образом, созданный массив объектов `Shape` способен хранить объекты классов, производных от базового класса `Shape` (попытка поместить в массив объекты, несовместимые с `Shape`, приведет к ошибке на этапе компиляции).

С учетом того, что все элементы в массиве `myShapes` на самом деле являются производными от `Shape`, вы знаете, что все они поддерживают один и тот же "полиморфный интерфейс" (или, говоря проще, все они имеют метод `Draw()`). Во время итерации по массиву ссылок `Shape` исполняющая система самостоятельно определяет лежащий в основе тип элемента. В этот момент и вызывается корректная версия метода `Draw()`.

Такой прием также делает простым безопасное расширение текущей иерархии. Например, пусть вы унаследовали от абстрактного базового класса `Shape` дополнительные классы (`Triangle`, `Square` и т.д.). Благодаря полиморфному интерфейсу код внутри цикла `foreach` не потребует никаких изменений, т.к. компилятор обеспечивает помещение внутрь массива `myShapes` только совместимых с `Shape` типов.

Сокрытие членов

Язык C# предоставляет возможность, которая логически противоположна переопределению методов и называется *сокрытием*. Говоря формально, если производный класс определяет член, который идентичен члену, определенному в базовом классе, то производный класс *скрывает* версию члена из родительского класса. В реальном мире такая ситуация чаще всего возникает, когда вы создаете подкласс от класса, который разрабатывали не вы (или ваша команда); например, такой класс может входить в состав пакета программного обеспечения .NET, приобретенного у независимого поставщика.

В целях иллюстрации предположим, что вы получили от коллеги на доработку класс по имени `ThreeDCircle`, в котором определен метод `Draw()`, не принимающий аргументов:

```
class ThreeDCircle
{
  public void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

Вы полагаете, что `ThreeDCircle` "является" `Circle`, поэтому решаете унаследовать его от своего существующего типа `Circle`:

```
class ThreeDCircle : Circle
{
  public void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

После перекомпиляции вы обнаруживаете следующее предупреждение:

```
'ThreeDCircle.Draw()' hides inherited member 'Circle.Draw()'. To make the
current member override that implementation, add the override keyword.
Otherwise add the new keyword.
```

Shapes.ThreeDCircle.Draw() скрывает унаследованный член Shapes.Circle.Draw(). Чтобы текущий член переопределял эту реализацию, добавьте ключевое слово override. В противном случае добавьте ключевое слово new.

Проблема здесь в том, что у вас есть производный класс (ThreeDCircle), который содержит метод, идентичный унаследованному методу. Решить эту проблему можно несколькими способами. Вы могли бы просто модифицировать версию метода Draw() из родительского класса, добавив ключевое слово override (как предлагает компилятор). При таком подходе у типа ThreeDCircle появляется возможность расширять стандартное поведение родительского типа, как и требовалось. Однако если у вас нет доступа к файлу кода с определением базового класса (частый случай, когда приходится работать с библиотеками от независимых поставщиков), то нет и возможности изменить метод Draw(), превратив его в виртуальный член.

В качестве альтернативы вы можете добавить ключевое слово new к определению проблемного члена Draw() своего производного типа (ThreeDCircle). Делая это, вы явно утверждаете, что реализация производного типа намеренно спроектирована так, чтобы фактически игнорировать версию члена из родительского типа (опять-таки, в реальном мире это может оказаться полезным, если внешнее программное обеспечение .NET каким-то образом конфликтует с вашим программным обеспечением).

```
// Этот класс расширяет Circle и скрывает унаследованный метод Draw().
class ThreeDCircle : Circle
{
  // Скрыть любую реализацию Draw(), находящуюся выше в иерархии.
  public new void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

Вы можете также применить ключевое слово new к любому члену типа, который унаследован от базового класса (полю, константе, статическому члену или свойству). Продолжая пример, предположим, что в классе ThreeDCircle необходимо скрыть унаследованное свойство PetName:

```
class ThreeDCircle : Circle
{
  // Скрыть свойство PetName, определенное выше в иерархии.
  public new string PetName { get; set; }

  // Скрыть любую реализацию Draw(), находящуюся выше в иерархии.
  public new void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

Наконец, имейте в виду, вы по-прежнему можете обратиться к реализации скрытого члена из базового класса, используя явное приведение, как описано в следующем разделе. Например, это демонстрируется в следующем коде:

```
static void Main(string[] args)
{
  ...
  // Здесь вызывается метод Draw(), определенный в классе ThreeDCircle.
  ThreeDCircle o = new ThreeDCircle();
  o.Draw();

  // Здесь вызывается метод Draw(), определенный в родительском классе!
  ((Circle)o).Draw();
  Console.ReadLine();
}
```

Исходный код. Проект `Shapes` доступен в подкаталоге `Chapter_6`.

Правила приведения для базовых и производных классов

Теперь, когда вы умеете строить семейства взаимосвязанных типов классов, нужно изучить правила, которым подчиняются *операции приведения классов*. Давайте возвратимся к иерархии классов сотрудников, созданной ранее в главе, и добавим несколько новых методов в класс `Program` (если вы прорабатываете примеры, откройте проект `Employees` в Visual Studio). Как описано в последнем разделе настоящей главы, изначальным базовым классом в системе является `System.Object`. По этой причине любой класс "является" `Object` и может трактоваться как таковой. Таким образом, внутри переменной типа `object` допускается хранить экземпляр любого типа:

```
static void CastingExamples()
{
  // Manager "является" System.Object, поэтому в переменной
  // типа object можно сохранять ссылку на Manager.
  object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
}
```

В примере `Employees` классы `Manager`, `SalesPerson` и `PTSalesPerson` расширяют класс `Employee`, поэтому допустимая ссылка на базовый класс может хранить любой из объектов указанных классов. Следовательно, приведенный далее код также законен:

```
static void CastingExamples()
{
  // Manager "является" System.Object, поэтому в переменной
  // типа object можно сохранять ссылку на Manager.
  object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);

  // Manager также "является" Employee.
  Employee moonUnit =
     new Manager("MoonUnit Zappa", 2, 3001, 20000, "101-11-1321", 1);

  // PTSalesPerson "является" SalesPerson.
  SalesPerson jill =
     new PTSalesPerson("Jill", 834, 3002, 100000, "111-12-1119", 90);
}
```

Первое правило приведения между типами классов гласит, что когда два класса связаны отношением "является", то всегда можно безопасно сохранить объект производного типа в ссылке базового класса. Формально это называется *неявным приведением*, поскольку оно "просто работает" в соответствии с законами наследования. В результате появляется возможность строить некоторые мощные программные конструкции. Например, предположим, что в текущем классе `Program` определен новый метод:

```
static void GivePromotion(Employee emp)
{
  // Повысить зарплату...
  // Предоставить место на парковке компании...

  Console.WriteLine("{0} was promoted!", emp.Name);
}
```

Из-за того, что данный метод принимает единственный параметр типа Employee, в сущности, ему можно передавать объект любого унаследованного от Employee класса, учитывая наличие отношения "является":

```
static void CastingExamples()
{
  // Manager "является" System.Object, поэтому в переменной
  // типа object можно сохранять ссылку на Manager.
  object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);

  // Manager также "является" Employee.
  Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000,
                                  "101-11-1321", 1);
  GivePromotion(moonUnit);

  // PTSalesPerson "является" SalesPerson.
  SalesPerson jill = new PTSalesPerson("Jill", 834, 3002, 100000,
                                       "111-12-1119", 90);
  GivePromotion(jill);
}
```

Предыдущий код компилируется благодаря неявному приведению от типа базового класса (Employee) к производному классу. Но что, если вы хотите также вызвать метод GivePromotion() с объектом frank (хранящимся в общей ссылке System.Object)? Если вы передадите объект frank методу GivePromotion() напрямую, как показано ниже, то получите ошибку на этапе компиляции:

```
object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
// Ошибка!
GivePromotion(frank);
```

Проблема в том, что вы пытаетесь передать переменную, которая объявлена как принадлежащая не к типу Employee, а к более общему типу System.Object. Учитывая, что в цепочке наследования он находится выше, чем Employee, компилятор не разрешит неявное приведение, стараясь сохранить ваш код насколько возможно безопасным в отношении типов.

Несмотря на то что сами вы можете выяснить, что ссылка object указывает в памяти на объект совместимого с Employee класса, компилятор сделать подобное не в состоянии, поскольку это не будет известно вплоть до времени выполнения. Чтобы удовлетворить компилятор, понадобится применить *явное приведение*. Это и есть второе правило приведения: в таких случаях вы можете явно приводить "вниз", используя операцию приведения C#. Базовый шаблон, которому нужно следовать при выполнении явного приведения, выглядит так:

```
(класс_к_которому_нужно_привести) существующая_ссылка
```

Таким образом, чтобы передать переменную типа object методу GivePromotion(), потребуется написать следующий код:

```
// Правильно!
GivePromotion((Manager)frank);
```

Ключевое слово as

Имейте в виду, что явное приведение оценивается *во время выполнения*, а не на этапе компиляции. Ради иллюстрации предположим, что проект Employees содержит копию класса Hexagon, созданного ранее в главе. Для простоты мы могли бы добавить в текущий проект такой класс:

```
class Hexagon
{
  public void Draw() { Console.WriteLine("Drawing a hexagon!"); }
}
```

Хотя приведение объекта сотрудника к объекту фигуры абсолютно лишено смысла, код вроде показанного ниже скомпилируется без ошибок:

```
// Привести объект frank к типу Hexagon невозможно,
// но этот код нормально скомпилируется!
object frank = new Manager();
Hexagon hex = (Hexagon)frank;
```

Тем не менее, вы получите ошибку времени выполнения, или, более формально — *исключение времени выполнения*. В главе 7 будут рассматриваться подробности структурированной обработки исключений, а пока полезно отметить, что при явном приведении можно перехватывать возможные ошибки с применением ключевых слов `try` и `catch`:

```
// Перехват возможной ошибки приведения.
object frank = new Manager();
Hexagon hex;
try
{
  hex = (Hexagon)frank;
}
catch (InvalidCastException ex)
{
  Console.WriteLine(ex.Message);
}
```

Очевидно, что показанный пример надуман; в такой ситуации вас никогда не будет беспокоить приведение между указанными типами. Однако предположим, что есть массив элементов `System.Object`, среди которых лишь малая толика содержит объекты, совместимые с `Employee`. В этом случае первым делом желательно определить, совместим ли элемент массива с типом `Employee`, и если да, тогда выполнить приведение.

Для быстрого определения совместимости одного типа с другим во время выполнения в C# предусмотрено ключевое слово `as`. С помощью ключевого слова `as` можно определить совместимость, проверив возвращаемое значение на предмет `null`. Взгляните на следующий код:

```
// Использование ключевого слова as для проверки совместимости.
object[] things = new object[4];
things[0] = new Hexagon();
things[1] = false;
things[2] = new Manager();
things[3] = "Last thing";

foreach (object item in things)
{
  Hexagon h = item as Hexagon;
  if (h == null)
    Console.WriteLine("Item is not a hexagon");
  else
  {
    h.Draw();
  }
}
```

Здесь производится проход в цикле по всем элементам в массиве объектов и проверка каждого из них на совместимость с классом Hexagon. Если (и только если) обнаруживается объект, совместимый с Hexagon, то вызывается метод Draw(). В противном случае выводится сообщение о том, что элемент является несовместимым.

Ключевое слово is

В дополнение к ключевому слову as язык C# предлагает ключевое слово is, предназначенное для определения совместимости типов двух элементов. Тем не менее, в отличие от ключевого слова as, если типы не совместимы, то ключевое слово is возвращает false, а не ссылку null. Другими словами, ключевое слово is не выполняет какое-либо приведение; оно просто проверяет на совместимость. Если элементы совместимы, тогда можно проводить безопасное приведение.

В настоящий момент метод GivePromotion() спроектирован для приема любого возможного типа, производного от Employee. Взгляните на следующую его модификацию, в которой теперь осуществляется проверка, какой конкретно "тип сотрудника" был передан:

```
static void GivePromotion(Employee emp)
{
  Console.WriteLine("{0} was promoted!", emp.Name);
  if (emp is SalesPerson)
  {
    Console.WriteLine("{0} made {1} sale(s)!", emp.Name,
      ((SalesPerson)emp).SalesNumber);
    Console.WriteLine();
  }
  if (emp is Manager)
  {
    Console.WriteLine("{0} had {1} stock options...", emp.Name,
      ((Manager)emp).StockOptions);
    Console.WriteLine();
  }
}
```

Здесь во время выполнения производится проверка с целью выяснения, на что именно в памяти указывает входная ссылка типа базового класса. После определения, принят ли объект типа SalesPerson или Manager, можно применить явное приведение, чтобы получить доступ к специализированным членам этого типа. Также обратите внимание, что помещать операции приведения внутрь конструкции try/catch не обязательно, т.к. внутри раздела if, выполнившего проверку условия, уже известно, что приведение является безопасным.

Главный родительский класс System.Object

В завершение этой главы мы подробно рассмотрим главный родительский класс в рамках платформы .NET — Object. При чтении предыдущих разделов вы могли заметить, что базовые классы во всех иерархиях (Car, Shape, Employee) никогда явно не указывали свои родительские классы:

```
// Какой класс является родительским для Car?
class Car
{...}
```

В мире .NET каждый тип в конечном итоге является производным от базового класса по имени System.Object, который в языке C# может быть представлен с помощью ключевого слова object. Класс Object определяет набор общих членов для каждого типа внутри платформы. По сути, когда вы строите класс, в котором явно не указан родительский класс, компилятор автоматически делает его производным от Object. Если вы хотите прояснить свои намерения, то можете определять классы, производные от Object, следующим образом (однако вы не обязаны поступать так):

```
// Явное наследование класса от System.Object.
class Car : object
{...}
```

Подобно любому классу в System.Object определен набор членов. В показанном ниже формальном определении C# обратите внимание, что некоторые члены объявлены как virtual, указывая на возможность их переопределения в подклассах, тогда как другие помечены ключевым словом static (и поэтому вызываются на уровне класса):

```
public class Object
{
  // Виртуальные члены.
  public virtual bool Equals(object obj);
  protected virtual void Finalize();
  public virtual int GetHashCode();
  public virtual string ToString();

  // Невиртуальные члены уровня экземпляра.
  public Type GetType();
  protected object MemberwiseClone();

  // Статические члены.
  public static bool Equals(object objA, object objB);
  public static bool ReferenceEquals(object objA, object objB);
}
```

В табл. 6.1 приведен обзор функциональности, предоставляемой некоторыми часто используемыми методами System.Object.

Таблица 6.1. Основные методы System.Object

Метод экземпляра	Описание
Equals()	По умолчанию этот метод возвращает true, только если сравниваемые элементы ссылаются на один и тот же объект в памяти. Таким образом, Equals() применяется для сравнения объектных ссылок, а не состояния объектов. Обычно данный метод переопределяется, чтобы возвращать true, когда сравниваемые объекты имеют одинаковые значения внутреннего состояния (семантика, основанная на значениях). Имейте в виду, что если вы переопределяете Equals(), то должны также переопределить GetHashCode(), т.к. эти методы используются внутренне типами Hashtable для извлечения подобъектов из контейнера. Также вспомните из главы 4, что в классе ValueType этот метод переопределен для всех структур, чтобы выполнять сравнение на основе значений
Finalize()	На данный момент можно считать, что этот метод (когда он переопределен) вызывается для освобождения любых выделенных ресурсов перед уничтожением объекта. Сборка мусора в среде CLR подробно рассматривается в главе 9
GetHashCode()	Этот метод возвращает значение int, которое идентифицирует конкретный объект

Окончание табл. 6.1

Метод экземпляра	Описание
`ToString()`	Этот метод возвращает строковое представление объекта в формате `<пространство_имен>.<имя_типа>` (называемое *полностью заданным именем*). Данный метод будет часто переопределяться в подклассе, чтобы вместо полностью заданного имени возвращать строку, которая состоит из пар "имя-значение", представляющих внутреннее состояние объекта
`GetType()`	Этот метод возвращает объект `Type`, полностью описывающий объект, на который в данный момент производится ссылка. Другими словами, это метод идентификации типов во время выполнения (Runtime Type Identification — RTTI), доступный всем объектам (он подробно обсуждается в главе 15)
`MemberwiseClone()`	Этот метод возвращает почленную копию текущего объекта и часто применяется для клонирования объектов (см. главу 8)

Чтобы проиллюстрировать стандартное поведение, обеспечиваемое базовым классом `Object`, создадим новый проект консольного приложения C# по имени `ObjectOverrides`. Добавим в проект новый тип класса C#, содержащий следующее пустое определение типа `Person`:

```
// Не забывайте, что класс Person расширяет Object.
class Person {}
```

Теперь обновим метод `Main()` для взаимодействия с унаследованными членами `System.Object`:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with System.Object *****\n");
    Person p1 = new Person();
    // Использовать унаследованные члены System.Object.
    Console.WriteLine("ToString: {0}", p1.ToString());
    Console.WriteLine("Hash code: {0}", p1.GetHashCode());
    Console.WriteLine("Type: {0}", p1.GetType());
    // Создать другие ссылки на p1.
    Person p2 = p1;
    object o = p2;
    // Указывают ли ссылки на один и тот же объект в памяти?
    if (o.Equals(p1) && p2.Equals(o))
    {
      Console.WriteLine("Same instance!");
    }
    Console.ReadLine();
  }
}
```

Вот вывод, получаемый из этого метода `Main()`:

```
***** Fun with System.Object *****

ToString: ObjectOverrides.Person
Hash code: 46104728
Type: ObjectOverrides.Person
Same instance!
```

Прежде всего, обратите внимание на то, что стандартная реализация `ToString()` возвращает полностью заданное имя текущего типа (`ObjectOverrides.Person`). Как будет показано в главе 14, где исследуется построение специальных пространств имен, каждый проект C# определяет "корневое пространство имен", название которого совпадает с именем проекта. Здесь мы создали проект по имени `ObjectOverrides`, поэтому тип `Person` и класс `Program` помещены внутрь пространства имен `ObjectOverrides`.

Стандартное поведение метода `Equals()` заключается в проверке, указывают ли две переменные на один и тот же объект в памяти. В коде мы создаем новую переменную `Person` по имени `p1`. В этот момент новый объект `Person` помещается в управляемую кучу. Переменная `p2` также относится к типу `Person`. Тем не менее, вместо создания *нового* экземпляра переменной `p2` присваивается ссылка `p1`. Таким образом, переменные `p1` и `p2` указывают на один и тот же объект в памяти, как и переменная `o` (типа `object`). Учитывая, что `p1`, `p2` и `o` указывают на одно и то же местоположение в памяти, проверка эквивалентности дает положительный результат.

Хотя готовое поведение `System.Object` в некоторых случаях может удовлетворять всем потребностям, довольно часто в специальных типах часть этих унаследованных методов переопределяется. В целях иллюстрации модифицируем класс `Person`, добавив свойства, которые представляют имя, фамилию и возраст лица; все они могут быть установлены с помощью специального конструктора:

```
// Не забывайте, что класс Person расширяет Object.
class Person
{
  public string FirstName { get; set; } = "";
  public string LastName { get; set; } = "";
  public int Age { get; set; }

  public Person(string fName, string lName, int personAge)
  {
    FirstName = fName;
    LastName = lName;
    Age = personAge;
  }
  public Person(){}
}
```

Переопределение метода `System.Object.ToString()`

Многие создаваемые классы (и структуры) могут извлечь преимущества от переопределения метода `ToString()` для возвращения строки с текстовым представлением текущего состояния экземпляра типа. Помимо прочего, это полезно при отладке. То, как вы решите конструировать эту строку — дело личных предпочтений; однако рекомендуемый подход предусматривает отделение пар "имя-значение" друг от друга двоеточиями и помещение всей строки в квадратные скобки (такому принципу следуют многие типы из библиотек базовых классов .NET). Взгляните на следующую переопределенную версию `ToString()` для класса `Person`:

```
public override string ToString()
{
  string myState;
  myState = string.Format("[First Name: {0}; Last Name: {1}; Age: {2}]",
    FirstName, LastName, Age);
  return myState;
}
```

Приведенная реализация метода `ToString()` довольно прямолинейна, потому что класс `Person` содержит всего три порции данных состояния. Тем не менее, всегда помните о том, что правильное переопределение `ToString()` должно также учитывать любые данные, определенные *выше в цепочке наследования*.

При переопределении метода `ToString()` для класса, расширяющего специальный базовый класс, первым делом необходимо получить возвращаемое значение `ToString()` из родительского класса, используя ключевое слово `base`. После получения строковых данных родительского класса их можно дополнить специальной информацией производного класса.

Переопределение метода `System.Object.Equals()`

Давайте также переопределим поведение метода `Object.Equals()`, чтобы работать с *семантикой на основе значений*. Вспомните, что по умолчанию `Equals()` возвращает `true`, только если два сравниваемых объекта ссылаются на один и тот же экземпляр объекта в памяти. Для класса `Person` может оказаться полезной такая реализация `Equals()`, которая возвращает `true`, если две сравниваемые переменные содержат те же самые значения состояния (например, фамилию, имя и возраст).

Прежде всего, обратите внимание, что входной аргумент метода `Equals()` имеет общий тип `System.Object`. В связи с этим первым делом необходимо удостовериться в том, что вызывающий код действительно передал экземпляр типа `Person`, и для дополнительной подстраховки проверить, что входной параметр не является ссылкой `null`.

После того, как вы установите, что вызывающий код передал выделенный экземпляр `Person`, один из подходов заключается в реализации метода `Equals()` для сравнения поле за полем данных входного объекта с данными текущего объекта:

```
public override bool Equals(object obj)
{
  if (obj is Person && obj != null)
  {
    Person temp;
    temp = (Person)obj;
    if (temp.FirstName == this.FirstName
      && temp.LastName == this.LastName
      && temp.Age == this.Age)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  return false;
}
```

Здесь производится сравнение значений входного объекта с внутренними значениями текущего объекта (обратите внимание на применение ключевого слова `this`). Если имя, фамилия и возраст в двух объектах идентичны, то эти два объекта имеют одинаковые данные состояния и возвращается значение `true`. Любые другие результаты приводят к возвращению `false`.

Хотя такой подход действительно работает, вы определенно в состоянии представить, насколько трудоемкой была бы реализация специального метода `Equals()` для нетривиальных типов, которые могут содержать десятки полей данных. Распространенное со-

кращение предусматривает использование собственной реализации метода ToString(). Если класс располагает подходящей реализацией ToString(), в которой учитываются все поля данных вверх по цепочке наследования, то можно просто сравнивать строковые данные объектов:

```
public override bool Equals(object obj)
{
  // Больше нет необходимости приводить obj к типу Person,
  // т.к. у всех типов имеется метод ToString().
  return obj.ToString() == this.ToString();
}
```

Обратите внимание, что в этом случае нет необходимости проверять входной аргумент на принадлежность к корректному типу (Person в нашем примере), поскольку метод ToString() поддерживают все типы .NET. Еще лучше то, что больше не требуется выполнять проверку на предмет равенства свойство за свойством, т.к. теперь просто проверяются значения, возвращаемые методом ToString().

Переопределение метода System.Object.GetHashCode()

В случае переопределения в классе метода Equals() вы также должны переопределить стандартную реализацию метода GetHashCode(). Выражаясь упрощенно, *хеш-код* — это числовое значение, которое представляет объект как специфическое состояние. Например, если вы создадите две переменные типа string, хранящие значение Hello, то они должны давать один и тот же хеш-код. Однако если одна из них хранит строку в нижнем регистре (hello), то должны получаться разные хеш-коды.

Для выдачи хеш-значения метод System.Object.GetHashCode() по умолчанию применяет адрес текущей ячейки памяти, где расположен объект. Тем не менее, если вы строите специальный тип, подлежащий хранению в экземпляре типа Hashtable (из пространства имен System.Collections), то должны всегда переопределять этот член, потому что для извлечения объекта тип Hashtable будет вызывать методы Equals() и GetHashCode().

На заметку! Говоря точнее, класс System.Collections.Hashtable внутренне вызывает метод GetHashCode(), чтобы получить общее представление о местоположении объекта, а с помощью последующего (внутреннего) вызова метода Equals() определяет его точно.

Хотя мы не собираемся помещать объекты Person в System.Collections.Hashtable, ради полноты давайте переопределим метод GetHashCode(). Существует много алгоритмов, которые можно применять для создания хеш-кода, как весьма изощренных, так и не очень. В большинстве ситуаций есть возможность генерировать значение хеш-кода, полагаясь на реализацию метода GetHashCode() из класса System.String.

Учитывая, что класс String уже имеет эффективный алгоритм хеширования, использующий для вычисления хеш-значения символьные данные объекта String, вы можете просто вызвать метод GetHashCode() с той частью полей данных, которая должна быть уникальной во всех экземплярах (вроде номера карточки социального страхования), если ее удается идентифицировать. Таким образом, если определить в классе Person свойство SSN, то переопределить метод GetHashCode() можно было бы следующим образом:

```
// Предположим, что имеется свойство SSN.
class Person
{
```

```
  public string SSN {get; set;} = "";

  // Возвратить хеш-код на основе уникальных строковых данных.
  public override int GetHashCode()
  {
    return SSN.GetHashCode();
  }
}
```

Если выбрать часть уникальных строковых данных не удается, но есть переопределенный метод `ToString()`**, то можете вызвать** `GetHashCode()` **на собственном строковом представлении:**

```
// Возвратить хеш-код на основе значения, возвращаемого методом ToString()
// для объекта Person.
public override int GetHashCode()
{
  return this.ToString().GetHashCode();
}
```

Тестирование модифицированного класса `Person`

Теперь, когда виртуальные члены класса `Object` **переопределены, обновим** `Main()` **для тестирования внесенных изменений:**

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with System.Object *****\n");

  // ПРИМЕЧАНИЕ: эти объекты идентичны и предназначены
  // для тестирования методов Equals() и GetHashCode().
  Person p1 = new Person("Homer", "Simpson", 50);
  Person p2 = new Person("Homer", "Simpson", 50);

  // Получить строковые версии объектов.
  Console.WriteLine("p1.ToString() = {0}", p1.ToString());
  Console.WriteLine("p2.ToString() = {0}", p2.ToString());

  // Протестировать переопределенный метод Equals().
  Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));

  // Проверить хеш-коды.
  Console.WriteLine("Same hash codes?: {0}",
    p1.GetHashCode() == p2.GetHashCode());
  Console.WriteLine();

  // Изменить возраст p2 и протестировать снова.
  p2.Age = 45;
  Console.WriteLine("p1.ToString() = {0}", p1.ToString());
  Console.WriteLine("p2.ToString() = {0}", p2.ToString());
  Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));
  Console.WriteLine("Same hash codes?: {0}",
    p1.GetHashCode() == p2.GetHashCode());
  Console.ReadLine();
}
```

Ниже показан вывод:

```
***** Fun with System.Object *****

p1.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p2.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p1 = p2?: True
```

```
Same hash codes?: True
p1.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p2.ToString() = [First Name: Homer; Last Name: Simpson; Age: 45]
p1 = p2?: False
Same hash codes?: False
```

Статические члены класса System.Object

В дополнение к только что рассмотренным членам уровня экземпляра класс `System.Object` определяет два (очень полезных) статических члена, которые также проверяют эквивалентность на основе значений или на основе ссылок. Взгляните на следующий код:

```
static void StaticMembersOfObject()
{
  // Статические члены System.Object.
  Person p3 = new Person("Sally", "Jones", 4);
  Person p4 = new Person("Sally", "Jones", 4);
  Console.WriteLine("P3 and P4 have same state: {0}", object.Equals(p3, p4));
  Console.WriteLine("P3 and P4 are pointing to same object: {0}",
    object.ReferenceEquals(p3, p4));
}
```

Здесь можно просто передать методу `ReferenceEquals()` два объекта (любого типа) и позволить классу `System.Object` выяснить их эквивалентность.

Исходный код. Проект `ObjectOverrides` доступен в подкаталоге `Chapter_6`.

Резюме

В этой главе объяснялась роль и детали наследования и полиморфизма. В ней были представлены многочисленные новые ключевые слова и лексемы для поддержки каждого приема. Например, вспомните, что двоеточие применяется для указания родительского класса для создаваемого типа. Родительские типы способны определять любое количество виртуальных и/или абстрактных членов для установления полиморфного интерфейса. Производные типы переопределяют эти члены, используя ключевое слово `override`.

Вдобавок к построению множества иерархий классов в главе также было исследовано явное приведение между базовыми и производными типами. В завершение главы рассматривались особенности главного родительского класса в библиотеках базовых классов .NET — `System.Object`.

ГЛАВА 7

Структурированная обработка исключений

В этой главе вы узнаете о том, как обрабатывать аномалии, возникающие во время выполнения кода C#, с использованием *структурированной обработки исключений*. Будут описаны не только ключевые слова C#, предназначенные для этих целей (`try`, `catch`, `throw`, `finally`, `when`), но и разница между исключениями уровня приложения и уровня системы, а также роль базового класса `System.Exception`. Кроме того, будет показано, как создавать специальные исключения, и рассмотрены некоторые инструменты отладки в Visual Studio, связанные с исключениями.

Ода ошибкам, дефектам и исключениям

Что бы ни нашептывало наше (порой раздутое) самолюбие, идеальных программистов не существует. Разработка программного обеспечения является сложным делом, и из-за этой сложности довольно часто даже самые лучшие программы поставляются с разнообразными *проблемами*. В одних случаях проблема возникает из-за "плохо написанного" кода (например, по причине выхода за границы массива), а в других — из-за ввода пользователем некорректных данных, которые не были учтены в кодовой базе приложения (скажем, кода в поле для телефонного номера вводится значение `Chucky`). Вне зависимости от причин проблемы в конечном итоге приложение не работает так, как ожидалось. Чтобы подготовить почву для предстоящего обсуждения структурированной обработки исключений, рассмотрим три распространенных термина, которые применяются для описания аномалий.

- *Дефекты*. Выражаясь просто, это ошибки, которые допустил программист. В качестве примера предположим, что вы программируете на неуправляемом C++. Если вы забудете освободить динамически выделенную память, вызывая утечку памяти, то получите дефект.
- *Пользовательские ошибки*. С другой стороны, пользовательские ошибки обычно возникают из-за тех, кто запускает приложение, а не тех, кто его создает. Например, ввод конечным пользователем в текстовое поле неправильно сформированной строки с высокой вероятностью может привести к генерации ошибки, если в коде не была предусмотрена обработка такого некорректного ввода.
- *Исключения*. Исключениями обычно считаются аномалии во время выполнения, которые трудно (а то и невозможно) учесть на стадии программирования приложения. Примерами исключений могут быть попытка соединения с базой данных, которая больше не существует, открытие недостоверного XML-файла или попытка

установления связи с машиной, которая в текущий момент находится в автономном режиме. В каждом из упомянутых случаев программист (или конечный пользователь) обладает довольно низким контролем над такими "исключительными" обстоятельствами.

С учетом приведенных определений должно быть понятно, что структурированная обработка *исключений* в .NET — это прием работы с *исключительными ситуациями* во время выполнения. Тем не менее, даже для дефектов и пользовательских ошибок, которые ускользнули от глаз программиста, среда CLR будет часто генерировать соответствующее исключение, идентифицирующее возникшую проблему. Скажем, в библиотеках базовых классов .NET определены многочисленные исключения, такие как `FormatException`, `IndexOutOfRangeException`, `FileNotFoundException`, `ArgumentOutOfRangeException` и т.д.

В рамках терминологии .NET *исключение* вызывается дефектами, некорректным пользовательским вводом и ошибками времени выполнения, даже если программисты могут трактовать каждую аномалию как отдельную проблему. Однако прежде чем погружаться в детали, давайте формализуем роль структурированной обработки исключений и посмотрим, чем она отличается от традиционных приемов обработки ошибок.

На заметку! Чтобы сделать примеры кода максимально ясными, мы не будем перехватывать абсолютно все исключения, которые может выдавать заданный метод из библиотеки базовых классов. Разумеется, в своих проектах производственного уровня вы должны весьма активно пользоваться приемами, описанными в этой главе.

Роль обработки исключений .NET

До появления платформы .NET обработка ошибок в среде операционной системы Windows представляла собой запутанную смесь технологий. Многие программисты внедряли собственную логику обработки ошибок в контекст разрабатываемого приложения. Например, команда разработчиков могла определять набор числовых констант для представления известных условий возникновения ошибок и затем применять эти константы как возвращаемые значения методов. Взгляните на следующий фрагмент кода на языке C:

```
/* Типичный механизм перехвата ошибок в стиле C. */
#define E_FILENOTFOUND 1000

int UseFileSystem()
{
  // Предполагается, что в этой функции происходит нечто
  // такое, что приводит к возврату следующего значения.
  return E_FILENOTFOUND;
}

void main()
{
  int retVal = UseFileSystem();
  if(retVal == E_FILENOTFOUND)
    printf("Cannot find file...");  // Не удалось найти файл
}
```

Такой подход далеко не идеален, учитывая тот факт, что константа `E_FILENOTFOUND` — всего лишь числовое значение, которое мало что говорит о том, каким образом решить возникшую проблему. В идеале желательно, чтобы название ошибки, описательное сообщение и другая полезная информация об условиях возникновения ошибки были

помещены в единственный четко определенный пакет (что как раз и происходит при структурированной обработке исключений). В дополнение к специальным приемам, к которым прибегают разработчики, внутри API-интерфейса Windows определены сотни кодов ошибок, которые поступают в виде `#define` и `HRESULT`, а также очень многих вариаций простых булевских значений (`bool`, `BOOL`, `VARIANT_BOOL` и т.д.).

Очевидной проблемой, присущей таким старым приемам, является полное отсутствие симметрии. Каждый подход более или менее подгоняется под заданную технологию, заданный язык и возможно даже заданный проекта. Чтобы положить конец этому безумству, платформа .NET предложила стандартную методику для генерации и перехвата ошибок времени выполнения — структурированную обработку исключений. Достоинство этой методики в том, что разработчики теперь имеют унифицированный подход к обработке ошибок, который является общим для всех языков, ориентированных на .NET. Следовательно, способ обработки ошибок, используемый программистом на C#, синтаксически подобен способу, который применяет программист на VB или программист на C++, имеющий дело с C++/CLI.

Дополнительное преимущество связано с тем, что синтаксис, используемый для генерации и отлавливания исключений за пределами границ сборок и машины, идентичен. Скажем, если вы применяете язык C# при построении службы Windows Communication Foundation (WCF), то можете сгенерировать исключение SOAP для удаленного вызывающего кода, используя те же самые ключевые слова, которые позволяют генерировать исключения внутри методов в одном приложении.

Еще одно преимущество исключений .NET состоит в том, что в отличие от загадочных числовых значений они представляют собой объекты, в которых содержится читабельное описание проблемы, а также детальный снимок стека вызовов на момент первоначального возникновения исключения. Более того, конечному пользователю можно предоставить справочную ссылку, которая указывает на URL-адрес с подробностями об ошибке, а также специальные данные, определенные программистом.

Строительные блоки обработки исключений в .NET

Программирование со структурированной обработкой исключений предусматривает применение четырех взаимосвязанных сущностей:

- тип класса, который представляет детали исключения;
- член, способный генерировать экземпляр класса исключения в вызывающем коде при соответствующих обстоятельствах;
- блок кода на вызывающей стороне, который обращается к члену, предрасположенному к возникновению исключения;
- блок кода на вызывающей стороне, который будет обрабатывать (или перехватывать) исключение, если оно произойдет.

Язык программирования C# предлагает пять ключевых слов (`try`, `catch`, `throw`, `finally` и `when`), которые позволяют генерировать и обрабатывать исключения. Объект, представляющий текущую проблему, относится к классу, который расширяет класс `System.Exception` (или производный от него класс). С учетом этого давайте исследуем роль данного базового класса, касающегося исключений.

Базовый класс `System.Exception`

Все исключения в конечном итоге порождены от базового класса `System.Exception`, который, в свою очередь, является производным от `System.Object`. Ниже показана основная часть этого класса (обратите внимание, что некоторые его члены являются виртуальными и, следовательно, могут быть переопределены в производных классах):

```
public class Exception : ISerializable, _Exception
{
  // Открытые конструкторы.
  public Exception(string message, Exception innerException);
  public Exception(string message);
  public Exception();
  ...
  // Методы.
  public virtual Exception GetBaseException();
  public virtual void GetObjectData(SerializationInfo info,
    StreamingContext context);
  // Свойства.
  public virtual IDictionary Data { get; }
  public virtual string HelpLink { get; set; }
  public Exception InnerException { get; }
  public virtual string Message { get; }
  public virtual string Source { get; set; }
  public virtual string StackTrace { get; }
  public MethodBase TargetSite { get; }
  ...
}
```

Как видите, многие свойства, определенные в классе `System.Exception`, по своей природе допускают только чтение. Причина в том, что стандартные значения для каждого из них обычно будут предоставляться производными типами. Например, стандартное сообщение типа `IndexOutOfRangeException` выглядит так: "Index was outside the bounds of the array" ("Индекс вышел за границы массива").

На заметку! Класс `Exception` реализует два интерфейса .NET. Хотя интерфейсы вы пока еще не изучали (см. главу 8), просто имейте в виду, что интерфейс `_Exception` позволяет исключению .NET обрабатываться неуправляемой кодовой базой (такой как приложение COM), в то время как интерфейс `ISerializable` дает возможность объекту исключения пересекать определенные границы (скажем, границы машины).

В табл. 7.1 описаны наиболее важные члены класса `System.Exception`.

Таблица 7.1. Основные члены типа `System.Exception`

Свойство `System.Exception`	Описание
`Data`	Это свойство только для чтения позволяет извлекать коллекцию пар "ключ-значение" (представленную объектом, реализующим `IDictionary`), которая предоставляет дополнительную определяемую программистом информацию об исключении. По умолчанию эта коллекция пуста
`HelpLink`	Это свойство позволяет получать или устанавливать URL для доступа к справочному файлу или веб-сайту с подробным описанием ошибки
`InnerException`	Это свойство только для чтения может использоваться для получения информации о предыдущем исключении или исключениях, которые послужили причиной возникновения текущего исключения. Запись предыдущих исключений осуществляется путем их передачи конструктору самого последнего сгенерированного исключения

Окончание табл. 7.1

Свойство System.Exception	Описание
Message	Это свойство только для чтения возвращает текстовое описание заданной ошибки. Само сообщение об ошибке устанавливается как параметр конструктора
Source	Это свойство позволяет получать либо устанавливать имя сборки или объекта, который привел к генерации исключения
StackTrace	Это свойство только для чтения содержит строку, идентифицирующую последовательность вызовов, которая привела к возникновению исключения. Как нетрудно догадаться, данное свойство очень полезно во время отладки или для сохранения информации об ошибке во внешнем журнале ошибок
TargetSite	Это свойство только для чтения возвращает объект MethodBase с описанием многочисленных деталей о методе, который привел к генерации исключения (вызов ToString() будет идентифицировать этот метод по имени)

Простейший пример

Чтобы продемонстрировать полезность структурированной обработки исключений, мы должны создать класс, который будет генерировать исключение в надлежащих (или, можно сказать, *исключительных*) обстоятельствах. Создадим новый проект консольного приложения C# по имени SimpleException и определим в нем два класса (Car (автомобиль) и Radio (радиоприемник)), связав их между собой отношением "имеет". В классе Radio определен единственный метод, который отвечает за включение и выключение радиоприемника:

```
class Radio
{
  public void TurnOn(bool on)
  {
    if(on)
      Console.WriteLine("Jamming...");      // включен
    else
      Console.WriteLine("Quiet time...");   // выключен
  }
}
```

Помимо использования класса Radio через включение/делегацию класс Car (его код показан ниже) определен так, что если пользователь превышает предопределенную максимальную скорость (заданную с помощью константного члена MaxSpeed), то двигатель выходит из строя, приводя объект Car в нерабочее состояние (отражается закрытой переменной-членом типа bool по имени carIsDead).

Кроме того, класс Car имеет несколько свойств для представления текущей скорости и указанного пользователем "дружественного названия" автомобиля, а также различные конструкторы для установки состояния нового объекта Car. Ниже приведено полное определение Car вместе с поясняющими комментариями.

```
class Car
{
  // Константа для представления максимальной скорости.
  public const int MaxSpeed = 100;
```

```
// Свойства автомобиля.
public int CurrentSpeed {get; set;} = 0;
public string PetName {get; set;} = "";

// Не вышел ли автомобиль из строя?
private bool carIsDead;

// В автомобиле имеется радиоприемник.
private Radio theMusicBox = new Radio();

// Конструкторы.
public Car() {}
public Car(string name, int speed)
{
  CurrentSpeed = speed;
  PetName = name;
}

public void CrankTunes(bool state)
{
  // Делегировать запрос внутреннему объекту.
  theMusicBox.TurnOn(state);
}

// Проверить, не перегрелся ли автомобиль.
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", PetName);
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed > MaxSpeed)
    {
      Console.WriteLine("{0} has overheated!", PetName);
      CurrentSpeed = 0;
      carIsDead = true;
    }
    else
      Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
  }
 }
}
```

Теперь реализуем метод `Main()`, в котором объект `Car` будет превышать заранее за данную максимальную скорость (установленную в `100` внутри класса `Car`):

```
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Exception Example *****");
  Console.WriteLine("=> Creating a car and stepping on it!");
  Car myCar = new Car("Zippy", 20);
  myCar.CrankTunes(true);

  for (int i = 0; i < 10; i++)
    myCar.Accelerate(10);
  Console.ReadLine();
}
```

Вывод будет выглядеть следующим образом:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
=> CurrentSpeed = 100
Zippy has overheated!
Zippy is out of order...
```

Генерация общего исключения

Теперь, имея функциональный класс `Car`, рассмотрим простейший способ генерации исключения. Текущая реализация метода `Accelerate()` просто отображает сообщение об ошибке, если вызывающий код пытается разогнать автомобиль до скорости, превышающей верхний предел.

Чтобы модернизировать этот метод для генерации исключения, когда пользователь пытается разогнать автомобиль до скорости, которая превышает установленный предел, потребуется создать и сконфигурировать новый экземпляр класса `System.Exception`, установив значение доступного только для чтения свойства `Message` через конструктор класса. Для отправки объекта ошибки обратно вызывающему коду применяется ключевое слово `throw` языка C#. Ниже приведен обновленный код метода `Accelerate()`:

```
// На этот раз генерировать исключение, если пользователь
// превышает предел, указанный в MaxSpeed.
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", PetName);
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      carIsDead = true;
      CurrentSpeed = 0;

      // Использовать ключевое слово throw для генерации исключения.
      throw new Exception(string.Format("{0} has overheated!", PetName));
    }
    else
      Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
  }
}
```

Прежде чем выяснять, каким образом вызывающий код будет перехватывать данное исключение, необходимо отметить несколько интересных моментов. Для начала, если вы генерируете исключение, то всегда самостоятельно решаете, как вводится в действие ошибка и когда должно генерироваться исключение. Здесь мы предполагаем, что при попытке увеличить скорость объекта `Car` за пределы максимума должен быть сгенерирован объект `System.Exception` для уведомления о невозможности продолжить выполнение метода `Accelerate()` (в зависимости от создаваемого приложения такое предположение может быть как допустимым, так и нет).

В качестве альтернативы метод `Accelerate()` можно было бы реализовать так, чтобы он производил автоматическое восстановление, не генерируя перед этим исключение. По большому счету, исключения должны генерироваться только при возникновении более критичного условия (например, отсутствие нужного файла, невозможность подключения к базе данных и т.п.). Принятие решения о том, что должно служить причиной генерации исключения, требует серьезного обдумывания и поиска веских оснований на стадии проектирования. Для преследуемых сейчас целей будем считать, что попытка увеличить скорость автомобиля выше максимально допустимой является вполне оправданной причиной для выдачи исключения.

В любом случае, если вы снова запустите приложение с показанной выше логикой в методе `Main()`, то исключение, в конце концов, будет сгенерировано. В следующем выводе видно, что результат отсутствия обработки этой ошибки нельзя назвать идеальным, учитывая получение многословного сообщения об ошибке, за которым следует прекращение работы программы:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
Unhandled Exception: System.Exception: Zippy has overheated!
 at SimpleException.Car.Accelerate(Int32 delta) in C:\MyBooks\C# Book (7th ed)
\Code\Chapter_7\SimpleException\Car.cs:line 62
 at SimpleException.Program.Main(String[] args) in C:\MyBooks\C# Book (7th ed)
\Code\Chapter_7\SimpleException\Program.cs:line 20
Press any key to continue . . .
```

Перехват исключений

На заметку! Те, кто пришел в .NET из мира Java, должны помнить о том, что члены типа не прототипируются набором исключений, которые они могут генерировать (другими словами, платформа .NET не поддерживает проверяемые исключения). Лучше это или хуже, но вы не обязаны обрабатывать каждое исключение, генерируемое отдельно взятым членом.

Поскольку метод `Accelerate()` теперь генерирует исключение, вызывающий код должен быть готов обработать его, если оно возникнет. При вызове метода, который может сгенерировать исключение, должен использоваться блок `try/catch`. После перехвата объекта исключения можно обращаться к различным его членам и извлекать детальную информацию о проблеме.

Дальнейшие действия с этими данными в значительной степени зависят от вас. Вы можете зафиксировать их в файле отчета, записать в журнал событий Windows, отправить по электронной почте системному администратору или отобразить конечному пользователю сообщение о проблеме. Здесь мы просто выводим детали исключения в окно консоли:

```
// Обработка сгенерированного исключения.
static void Main(string[] args)
{
```

```
  Console.WriteLine("***** Simple Exception Example *****");
  Console.WriteLine("=> Creating a car and stepping on it!");
  Car myCar = new Car("Zippy", 20);
  myCar.CrankTunes(true);

  // Разогнаться до скорости, превышающей максимальный
  // предел автомобиля, с целью выдачи исключения.
  try
  {
    for(int i = 0; i < 10; i++)
      myCar. Accelerate(10);
  }
  catch(Exception e)
  {
    Console.WriteLine("\n*** Error! ***");           // ошибка
    Console.WriteLine("Method: {0}", e.TargetSite);  // метод
    Console.WriteLine("Message: {0}", e.Message);    // сообщение
    Console.WriteLine("Source: {0}", e.Source);      // источник
  }

  // Ошибка была обработана, продолжается выполнение следующего оператора.
  Console.WriteLine("\n***** Out of exception logic *****");
  Console.ReadLine();
}
```

По существу блок `try` представляет собой раздел операторов, которые в ходе выполнения могут генерировать исключение. Если исключение обнаруживается, то управление переходит к соответствующему блоку `catch`. С другой стороны, если код внутри блока `try` исключение не сгенерировал, то блок `catch` полностью пропускается, и выполнение проходит обычным образом. Ниже представлен вывод, полученный в результате тестового запуска данной программы:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90

*** Error! ***
Method: Void Accelerate(Int32)
Message: Zippy has overheated!
Source: SimpleException

***** Out of exception logic *****
```

Как видите, после обработки исключения приложение может продолжать свое функционирование с оператора, находящегося после блока `catch`. В некоторых обстоятельствах исключение может оказаться достаточно критическим для того, чтобы служить основанием завершения работы приложения. Тем не менее, во многих случаях логика внутри обработчика исключений позволяет приложению спокойно продолжить выполнение (хотя, может быть, с несколько меньшим объемом функциональности, например, без возможности подключения к удаленному источнику данных).

Конфигурирование состояния исключения

В настоящий момент объект `System.Exception`, сконфигурированный внутри метода `Accelerate()`, просто устанавливает значение, доступное через свойство `Message` (посредством параметра конструктора). Как было показано ранее в табл. 7.1, класс `Exception` также предлагает несколько дополнительных членов (`TargetSite`, `StackTrace`, `HelpLink` и `Data`), которые полезны для дальнейшего уточнения природы возникшей проблемы. Чтобы усовершенствовать текущий пример, давайте по очереди рассмотрим возможности этих членов.

Свойство `TargetSite`

Свойство `System.Exception.TargetSite` позволяет выяснить разнообразные детали о методе, который сгенерировал заданное исключение. Как было показано в предыдущем методе `Main()`, вывод значения свойства `TargetSite` приводит к отображению возвращаемого значения, имени и типов параметров метода, который сгенерировал исключение. Однако свойство `TargetSite` возвращает не простую строку, а строго типизированный объект `System.Reflection.MethodBase`. Данный тип можно применять для сбора многочисленных деталей, касающихся проблемного метода, а также класса, в котором этот метод определен. В целях иллюстрации изменим предыдущую логику в блоке `catch` следующим образом:

```
static void Main(string[] args)
{
  ...
 // Свойство TargetSite в действительности возвращает объект MethodBase.
 catch(Exception e)
 {
  Console.WriteLine("\n*** Error! ***");
  Console.WriteLine("Member name: {0}", e.TargetSite);
                    // имя члена
  Console.WriteLine("Class defining member: {0}", e.TargetSite.DeclaringType);
                  // класс, определяющий член
  Console.WriteLine("Member type: {0}", e.TargetSite.MemberType);
                    // тип члена
  Console.WriteLine("Message: {0}", e.Message);
                    // сообщение
  Console.WriteLine("Source: {0}", e.Source);
                    // источник
 }
 Console.WriteLine("\n***** Out of exception logic *****");
 Console.ReadLine();
}
```

На этот раз в коде используется свойство `MethodBase.DeclaringType` для выяснения полностью заданного имени класса, сгенерировавшего ошибку (в данном случае `SimpleException.Car`), а также свойство `MemberType` объекта `MethodBase` для идентификации типа члена (скажем, свойство или метод), в котором возникло исключение. Ниже показано, как теперь будет выглядеть вывод в результате выполнения логики в блоке `catch`:

```
*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
```

Свойство StackTrace

Свойство System.Exception.StackTrace позволяет определить последовательность вызовов, которая в результате привела к генерации исключения. Значение данного свойства никогда не устанавливается вручную — это делается автоматически во время создания объекта исключения. Чтобы подтвердить сказанное, модифицируем логику в блоке catch:

```
catch(Exception e)
{
  ...
  Console.WriteLine("Stack: {0}", e.StackTrace);
}
```

Если снова запустить программу, то в окне консоли можно будет обнаружить следующие данные трассировки стека (естественно, номера строк и пути к файлам у вас могут отличаться):

```
Stack: at SimpleException.Car.Accelerate(Int32 delta)
in c:\MyApps\SimpleException\car.cs:line 65 at SimpleException.Program.Main()
in c:\MyApps\SimpleException\Program.cs:line 21
```

Значение string, возвращаемое свойством StackTrace, отражает последовательность вызовов, которая привела к генерации этого исключения. Обратите внимание, что самый нижний номер строки в string указывает на место возникновения первого вызова в последовательности, а самый верхний — на место, где точно находится проблемный член. Очевидно, что такая информация очень полезна во время отладки или при ведении журнала для конкретного приложения, т.к. дает возможность отследить путь к источнику ошибки.

Свойство HelpLink

Хотя свойства TargetSite и StackTrace позволяют программистам выяснить, почему возникло конкретное исключение, эта информация не особенно полезна для пользователей. Как уже было показано, с помощью свойства System.Exception.Message можно извлечь читабельную информацию и отобразить ее конечному пользователю. Вдобавок можно установить свойство HelpLink для указания на специальный URL или стандартный справочный файл Windows, где приводятся более подробные сведения о проблеме.

По умолчанию значением свойства HelpLink является пустая строка. Если вы хотите присвоить данному свойству какое-то более интересное значение, то должны делать это перед генерацией исключения System.Exception. Изменим код метода Car.Accelerate():

```
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", PetName);
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      carIsDead = true;
      CurrentSpeed = 0;
```

```
      // Мы хотим обращаться к свойству HelpLink, поэтому необходимо
      // создать локальную переменную перед генерацией объекта Exception.
      Exception ex =
        new Exception(string.Format("{0} has overheated!", PetName));
      ex.HelpLink = "http://www.CarsRUs.com";
      throw ex;
    }
    else
      Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
                    // Вывод текущей скорости
  }
}
```

Теперь можно обновить логику в блоке `catch` **для вывода на консоль информации из свойства** `HelpLink`:

```
catch(Exception e)
{
  ...
  Console.WriteLine("Help Link: {0}", e.HelpLink);
}
```

Свойство `Data`

Свойство `Data` **класса** `System.Exception` **позволяет заполнить объект исключения подходящей вспомогательной информацией (такой как метка времени). Свойство** `Data` **возвращает объект, который реализует интерфейс по имени** `IDictionary`**, определенный в пространстве имен** `System.Collections`**. В главе 8 исследуется роль программирования на основе интерфейсов, а также рассматривается пространство имен** `System.Collections`**. На данный момент важно понимать лишь то, что словарные коллекции позволяют создавать наборы значений, извлекаемых по ключу. Взгляните на очередное изменение метода** `Car.Accelerate()`:

```
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", PetName);
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      carIsDead = true;
      CurrentSpeed = 0;
      // Мы хотим обращаться к свойству HelpLink, поэтому необходимо
      // создать локальную переменную перед генерацией объекта Exception.
      Exception ex =
        new Exception(string.Format("{0} has overheated!", PetName));
      ex.HelpLink = "http://www.CarsRUs.com";
      // Предоставить специальные данные, касающиеся ошибки.
      ex.Data.Add("TimeStamp",
        string.Format("The car exploded at {0}", DateTime.Now));
      ex.Data.Add("Cause", "You have a lead foot.");
      throw ex;
    }
    else
      Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
  }
}
```

Для успешного перечисления пар "ключ-значение" нужно сначала указать директиву `using` для пространства имен `System.Collections`, т.к. в файле с классом, реализующим метод `Main()`, будет применяться тип `DictionaryEntry`:

```
using System.Collections;
```

Затем потребуется обновить логику в блоке `catch`, чтобы обеспечить проверку значения, возвращаемого из свойства `Data`, на равенство `null` (т.е. стандартному значению). После этого свойства `Key` и `Value` типа `DictionaryEntry` используются для вывода специальных данных на консоль:

```
catch (Exception e)
{
...
  Console.WriteLine("\n-> Custom Data:");
  foreach (DictionaryEntry de in e.Data)
    Console.WriteLine("-> {0}: {1}", de.Key, de.Value);
}
```

Вот как теперь выглядит финальный вывод программы:

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90

*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
Stack: at SimpleException.Car.Accelerate(Int32 delta)
       at SimpleException.Program.Main(String[] args)
Help Link: http://www.CarsRUs.com

-> Custom Data:
-> TimeStamp: The car exploded at 9/12/2015 9:02:12 PM
-> Cause: You have a lead foot.

***** Out of exception logic *****
```

Свойство `Data` удобно тем, что позволяет упаковывать специальную информацию об ошибке, не требуя построения нового типа класса для расширения базового класса `Exception`. Тем не менее, каким бы полезным ни было свойство `Data`, разработчики приложений .NET все равно обычно строят строго типизированные классы исключений, которые поддерживают специальные данные, применяя строго типизированные свойства.

Такой подход позволяет вызывающему коду перехватывать конкретный производный от `Exception` тип, а не углубляться в коллекцию данных для получения дополнительных деталей. Чтобы понять, как это работает, необходимо разобраться с разницей между исключениями уровня системы и уровня приложения.

Исходный код. Проект `SimpleException` доступен в подкаталоге `Chapter_7`.

Исключения уровня системы (System.SystemException)

В библиотеках базовых классов .NET определено много классов, которые в конечном итоге являются производными от System.Exception. Например, в пространстве имен System определены основные объекты исключений, такие как ArgumentOutOfRangeException, IndexOutOfRangeException, StackOverflowException и т.п. В других пространствах имен есть исключения, которые отражают поведение этих пространств имен. Например, в System.Drawing.Printing определены исключения, связанные с выводом на печать, в System.IO — исключения, возникающие во время ввода-вывода, в System.Data — исключения, специфичные для баз данных, и т.д.

Исключения, которые генерируются самой платформой .NET, называются *системными исключениями*. Такие исключения в общем случае рассматриваются как неисправимые фатальные ошибки. Системные исключения унаследованы прямо от базового класса System.SystemException, который, в свою очередь, порожден от System.Exception (а тот — от класса System.Object):

```
public class SystemException : Exception
{
  // Разнообразные конструкторы.
}
```

Учитывая, что тип System.SystemException не добавляет никакой дополнительной функциональности кроме набора специальных конструкторов, вас может интересовать, по какой причине он вообще существует. Попросту говоря, когда тип исключения является производным от System.SystemException, то есть возможность выяснить, что исключение сгенерировала исполняющая среда .NET, а не кодовая база выполняющегося приложения. Это довольно легко проверить, используя ключевое слово is:

```
// Верно! NullReferenceException является SystemException.
NullReferenceException nullRefEx = new NullReferenceException();
Console.WriteLine("NullReferenceException is-a SystemException? : {0}",
                  nullRefEx is SystemException);
```

Исключения уровня приложения (System.ApplicationException)

Поскольку все исключения .NET — это типы классов, вы можете создавать собственные исключения, специфичные для приложения. Однако из-за того, что базовый класс System.SystemException представляет исключения, генерируемые средой CLR, может сложиться впечатление о том, что вы должны порождать свои специальные исключения от типа System.Exception. Конечно, можно поступать и так, но взамен их лучше наследовать от класса System.ApplicationException:

```
public class ApplicationException : Exception
{
  // Разнообразные конструкторы.
}
```

Как и SystemException, кроме набора конструкторов никаких дополнительных членов в классе ApplicationException не определено. С точки зрения функциональности единственная цель класса System.ApplicationException состоит в идентификации источника ошибки. При обработке исключения, производного от System.ApplicationException, можно предполагать, что исключение было сгенериро-

вано кодовой базой выполняющегося приложения, а не библиотеками базовых классов .NET либо исполняющей средой .NET.

На заметку! На практике лишь немногие разработчики приложений .NET строят специальные исключения, которые расширяют класс `ApplicationException`. Вместо этого они чаще создают подкласс `System.Exception`, хотя формально допустимы оба подхода.

Построение специальных исключений, способ первый

В то время как для сигнализации об ошибке времени выполнения можно всегда генерировать экземпляры `System.Exception` (это было показано в первом примере), иногда предпочтительнее создавать *строго типизированное исключение*, которое представляет уникальные детали, связанные с текущей проблемой. Например, предположим, что вы хотите построить специальное исключение (по имени `CarIsDeadException`) для представления ошибки, которая возникает из-за увеличения скорости неисправного автомобиля. Первым делом создается новый класс, унаследованный от `System.Exception`/`System.ApplicationException` (по соглашению имена всех классов исключений заканчиваются суффиксом `Exception`; в действительности это установившаяся практика в .NET).

На заметку! Как правило, все специальные классы исключений должны быть определены как открытые (вспомните, что стандартным модификатором доступа для невложенных типов является `internal`). Причина в том, что исключения часто передаются за границы сборок, поэтому они должны быть доступны вызывающей кодовой базе.

Создадим новый проект консольного приложения по имени `CustomException` и скопируем в него предыдущие файлы `Car.cs` и `Radio.cs`, выбрав пункт меню Project⇨Add Existing Item (Проект⇨Добавить существующий элемент) и для ясности изменив название пространства имен, в котором определены типы `Car` и `Radio`, с `SimpleException` на `CustomException`. Затем добавим в него следующее определение класса:

```
// Это специальное исключение описывает детали условия выхода автомобиля из строя.
// (Не забывайте, что можно также просто расширить класс Exception.)
public class CarIsDeadException : ApplicationException
{}
```

Как и с любым классом, вы можете создавать произвольное количество специальных членов, к которым можно обращаться внутри блока `catch` в вызывающем коде. Кроме того, вы можете также переопределять любые виртуальные члены, определенные в родительских классах. Например, вы могли бы реализовать `CarIsDeadException`, переопределив виртуальное свойство `Message`.

Вместо заполнения словаря данных (через свойство `Data`) при генерировании исключения конструктор позволяет указывать метку времени и причину возникновения ошибки. Наконец, метка времени и причина возникновения ошибки могут быть получены с применением строго типизированных свойств:

```
public class CarIsDeadException : ApplicationException
{
  private string messageDetails = String.Empty;
  public DateTime ErrorTimeStamp {get; set;}
  public string CauseOfError {get; set;}

  public CarIsDeadException(){}
```

```
  public CarIsDeadException(string message, string cause, DateTime time)
  {
    messageDetails = message;
    CauseOfError = cause;
    ErrorTimeStamp = time;
  }
  // Переопределение свойства Exception.Message.
  public override string Message
  {
    get
    {
      return string.Format("Car Error Message: {0}", messageDetails);
    }
  }
}
```

Здесь класс `CarIsDeadException` **поддерживает закрытое поле (**`messageDetails`**), которое представляет данные, касающиеся текущего исключения; его можно устанавливать с использованием специального конструктора. Сгенерировать это исключение в методе** `Accelerate()` **несложно. Понадобится просто создать, сконфигурировать и сгенерировать объект** `CarIsDeadException`**, а не** `System.Exception` **(обратите внимание, что в данном случае заполнять коллекцию данных вручную больше не требуется):**

```
// Сгенерировать специальное исключение CarIsDeadException.
public void Accelerate(int delta)
{
  ...
  CarIsDeadException ex =
   new CarIsDeadException (string.Format("{0} has overheated!", PetName),
                           "You have a lead foot", DateTime.Now);
  ex.HelpLink = "http://www.CarsRUs.com";
  throw ex;
  ...
}
```

Для перехвата такого входного исключения блок `catch` **теперь можно модифицировать, чтобы в нем перехватывался конкретный тип** `CarIsDeadException` **(тем не менее, учитывая, что** `System.CarIsDeadException` **"является"** `System.Exception`**, по-прежнему допустимо перехватывать** `System.Exception`**):**

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    // Отслеживание исключения.
    myCar.Accelerate(50);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
    Console.WriteLine(e.ErrorTimeStamp);
    Console.WriteLine(e.CauseOfError);
  }
  Console.ReadLine();
}
```

Итак, с обретением понимания базового процесса построения специального исключения, вас наверняка интересует, когда может понадобиться поступать подобным образом. Обычно вам придется создавать специальные исключения только в случаях, если ошибка тесно связана с выдающим ее классом. В качестве примеров можно назвать специальный класс для работы с файлами, который генерирует набор ошибок, относящихся к файлам, класс `Car`, который генерирует ошибки, связанные с автомобилем, объект доступа к данным, который генерирует ошибки, имеющие отношение к определенной таблице базы данных, и т.д. Создавая специальные исключения, вы предоставляете вызывающему коду возможность обрабатывать многочисленные исключения по отдельности на дескриптивной основе.

Построение специальных исключений, способ второй

Текущий класс `CarIsDeadException` переопределяет виртуальное свойство `System.Exception.Message`, чтобы сконфигурировать специальное сообщение об ошибке, и предоставляет два специальных свойства для учета дополнительных порций данных. Однако в реальности переопределять виртуальное свойство `Message` не обязательно, т.к. входное сообщение можно просто передать конструктору родительского класса:

```
public class CarIsDeadException : ApplicationException
{
  public DateTime ErrorTimeStamp { get; set; }
  public string CauseOfError { get; set; }

  public CarIsDeadException() { }

  // Передача сообщения конструктору родительского класса.
  public CarIsDeadException(string message, string cause, DateTime time)
    :base(message)
  {
    CauseOfError = cause;
    ErrorTimeStamp = time;
  }
}
```

Обратите внимание, что на этот раз *не* объявляется строковая переменная для представления сообщения и *не* переопределяется свойство `Message`. Взамен нужный параметр просто передается конструктору базового класса. При таком проектном решении специальный класс исключения является всего лишь уникально именованным классом, производным от `System.ApplicationException` (с дополнительными свойствами в случае необходимости), который не переопределяет какие-либо члены базового класса.

Не удивляйтесь, если большинство специальных классов исключений (а то и все) будет соответствовать такому простому шаблону. Во многих случаях роль специального исключения не обязательно связана с предоставлением дополнительной функциональности кроме унаследованной от базовых классов. В действительности преследуется цель обеспечить *строго именованный тип*, который четко идентифицирует природу ошибки, так что для разных типов исключений клиент может реализовать разную логику обработки.

Построение специальных исключений, способ третий

Если вы хотите создать по-настоящему интересный специальный класс исключения, то должны позаботиться о том, чтобы он следовал передовому опыту .NET. В частности, это предполагает, что класс обладает следующими характеристиками:

- является производным от класса `Exception`/`ApplicationException`;
- помечен атрибутом `[System.Serializable]`;
- определяет стандартный конструктор;
- определяет конструктор, который устанавливает значение унаследованного свойства `Message`;
- определяет конструктор для обработки "внутренних исключений";
- определяет конструктор для поддержки сериализации типа.

При наличии только базовых знаний платформы .NET роль атрибутов и сериализации объектов может быть не ясна, что вполне нормально. Эти темы будут подробно раскрыты далее в книге (в главе 15 объясняются атрибуты, а в главе 20 — службы сериализации). Тем не менее, чтобы завершить исследование специальных исключений, ниже приведена последняя версия класса `CarIsDeadException`, в которой реализованы все упомянутые выше специальные конструкторы (остальные специальные свойства и конструкторы выглядят так, как было показано в примере внутри раздела "Построение специальных исключений, способ второй"):

```
[Serializable]
public class CarIsDeadException : ApplicationException
{
  public CarIsDeadException() { }
  public CarIsDeadException(string message) : base( message ) { }
  public CarIsDeadException(string message, System.Exception inner)
    : base( message, inner ) { }
  protected CarIsDeadException(
    System.Runtime.Serialization.SerializationInfo info,
    System.Runtime.Serialization.StreamingContext context)
    : base( info, context ) { }
  // Любые дополнительные специальные свойства, конструкторы и члены данных...
}
```

Поскольку создаваемые специальные исключения, соответствующие установившейся практике в .NET, на самом деле отличаются только своими именами, полезно знать, что среда Visual Studio предлагает фрагмент кода под названием `Exception` (рис. 7.1), который автоматически генерирует новый класс исключения, отвечающий рекомендациям .NET. (Вспомните из главы 2, что для активизации фрагмента кода необходимо ввести его имя, которым в данном случае является `exception`, и два раза нажать клавишу <Tab>.)

Рис. 7.1. Фрагмент кода `Exception`

Исходный код. Проект `CustomException` находится в подкаталоге `Chapter_7`.

Обработка множества исключений

В своей простейшей форме блок `try` сопровождается единственным блоком `catch`. Однако в реальности часто приходится сталкиваться с ситуациями, когда операторы внутри блока `try` могут генерировать *многочисленные* исключения. Создадим новый проект консольного приложения на C# по имени `ProcessMultipleExpceptions`, добавим в него файлы `Car.cs`, `Radio.cs` и `CarIsDeadException.cs` из предыдущего проекта `CustomException` (посредством пункта меню Project⇨Add Existing Item) и соответствующим образом модифицируем название пространства имен.

Далее изменим метод `Accelerate()` класса `Car` так, чтобы он генерировал еще и предопределенное в библиотеках базовых классов исключение `ArgumentOutOfRangeException`, если передается недопустимый параметр (которым будет считаться любое значение меньше нуля). Обратите внимание, что конструктор этого класса исключения принимает имя проблемного аргумента в первом параметре `string`, за которым следует сообщение с описанием ошибки.

```
// Перед продолжением проверить аргумент на предмет допустимости.
public void Accelerate(int delta)
{
 if(delta < 0)
  throw new
   ArgumentOutOfRangeException("delta", "Speed must be greater than zero!");
                              // Значение скорости должно быть больше нуля!
  ...
}
```

Теперь логика в блоке `catch` может реагировать на каждый тип исключения специализированным образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    // Arg вызовет исключение выхода за пределы диапазона.
    myCar.Accelerate(-10);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

При написании множества блоков `catch` вы должны иметь в виду, что когда исключение сгенерировано, оно будет обрабатываться первым подходящим блоком `catch`. Чтобы проиллюстрировать, что означает "первый подходящий" блок `catch`, модифицируем предыдущий код, добавив еще один блок `catch`, который пытается обработать

все исключения помимо `CarIsDeadException` **и** `ArgumentOutOfRangeException` **путем перехвата общего типа** `System.Exception`:

```
// Этот код не скомпилируется!
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);

  try
  {
    // Вызвать исключение выхода за пределы диапазона аргумента.
    myCar.Accelerate(-10);
  }
  catch(Exception e)
  {
    // Обработать все остальные исключения?
    Console.WriteLine(e.Message);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

Такая логика обработки исключений приводит к ошибкам на этапе компиляции. Проблема в том, что первый блок `catch` **может обрабатывать** *любые* **исключения, производные от** `System.Exception` **(с учетом отношения "является"), в том числе** `CarIsDeadException` **и** `ArgumentOutOfRangeException`. **Следовательно, два последних блока** `catch` **в принципе недостижимы!**

Запомните эмпирическое правило: блоки `catch` **должны быть структурированы так, чтобы первый** `catch` **перехватывал наиболее специфическое исключение (т.е. производный тип, расположенный ниже всех в цепочке наследования типов исключений), а последний** `catch` **— самое общее исключение (т.е. базовый класс заданной цепочки наследования; в этом случае** `System.Exception`**).**

Таким образом, если вы хотите определить блок `catch`**, который будет обрабатывать любые исключения помимо** `CarIsDeadException` **и** `ArgumentOutOfRangeException`**, то можно было бы написать следующий код:**

```
// Этот код скомпилируется без проблем.
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    // Вызвать исключение выхода за пределы диапазона аргумента.
    myCar.Accelerate(-10);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
```

```
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  // Этот блок будет перехватывать любые исключения помимо
  // CarIsDeadException и ArgumentOutOfRangeException.
  catch (Exception e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

На заметку! Везде, где только возможно, отдавайте предпочтение перехвату специфичных классов исключений, а не общего класса `System.Exception`. Хотя может показаться, что это упрощает жизнь в краткосрочной перспективе (поскольку охватывает все исключения, которые пока не беспокоят), в долгосрочной перспективе могут возникать странные аварийные отказы во время выполнения, т.к. в коде не была предусмотрена непосредственная обработка более серьезной ошибки. Не забывайте, что финальный блок `catch`, который работает с `System.Exception`, на самом деле имеет тенденцию быть чрезвычайно общим.

Общие операторы `catch`

В языке C# также поддерживается "общий" контекст `catch`, который не получает явно объект исключения, сгенерированный заданным членом:

```
// Общий оператор catch.
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    myCar.Accelerate(90);
  }
  catch
  {
    Console.WriteLine("Something bad happened...");
                    // Произошло что-то плохое...
  }
  Console.ReadLine();
}
```

Очевидно, это не самый информативный способ обработки исключений, поскольку нет никакой возможности для получения содержательных данных о возникшей ошибке (таких как имя метода, стек вызовов или специальное сообщение). Тем не менее, в C# такая конструкция разрешена, потому что она может быть полезной, когда требуется обработать все ошибки в обобщенной манере.

Повторная генерация исключений

Когда вы перехватываете исключение, внутри блока `try` его разрешено *повторно сгенерировать* для передачи вверх по стеку вызовов предшествующему вызывающему коду. Для этого просто используйте ключевое слово `throw` в блоке `catch`.

В итоге исключение передается вверх по цепочке вызовов, что может оказаться полезным, если блок catch способен обработать текущую ошибку только частично:

```
// Передача ответственности.
static void Main(string[] args)
{
  ...
  try
  {
    // Логика увеличения скорости автомобиля...
  }
  catch(CarIsDeadException e)
  {
    // Выполнить частичную обработку этой ошибки и передать ответственность.
    throw;
  }
  ...
}
```

Имейте в виду, что в данном примере кода конечным получателем исключения CarIsDeadException будет среда CLR, т.к. метод Main() генерирует его повторно. По этой причине конечному пользователю будет отображаться системное диалоговое окно с информацией об ошибке. Обычно вы будете повторно генерировать частично обработанное исключение для передачи вызывающему коду, который имеет возможность обработать входное исключение более элегантным образом.

Также обратите внимание на неявную повторную генерацию объекта CarIsDeadException с помощью ключевого слова throw без аргументов. Дело в том, что здесь не создается новый объект исключения, а просто передается исходный объект исключения (со всей его исходной информацией). Это позволяет сохранить контекст первоначального целевого объекта.

Внутренние исключения

Как нетрудно догадаться, вполне возможно, что исключение сгенерируется во время обработки другого исключения. Например, предположим, что вы обрабатываете исключение CarIsDeadException внутри отдельного блока catch, и в ходе этого процесса пытаетесь записать данные трассировки стека в файл carErrors.txt на диске C: (для получения доступа к типам, связанным с вводом-выводом, потребуется указать директиву using с пространством имен System.IO):

```
catch(CarIsDeadException e)
{
  // Попытка открытия файла carErrors.txt, расположенного на диске C:.
  FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
  ...
}
```

Если указанный файл на диске C: отсутствует, то вызов метода File.Open() приведет к генерации исключения FileNotFoundException! Позже в книге, когда мы будем подробно рассматривать пространство имен System.IO, вы узнаете, как программно определить, существует ли файл на жестком диске, перед попыткой его открытия (тем самым вообще избегая исключения). Однако чтобы не отклоняться от темы исключений, скажем, что такое исключение было сгенерировано.

Когда во время обработки исключения вы сталкиваетесь с еще одним исключением, установившаяся практика предусматривает обязательное сохранение нового объекта исключения как "внутреннего исключения" в новом объекте того же типа, что и исход-

ное исключение. Причина, по которой необходимо создавать новый объект обрабатываемого исключения, связана с тем, что единственным способом документирования внутреннего исключения является применение параметра конструктора. Взгляните на следующий код:

```
catch (CarIsDeadException e)
{
  try
  {
    FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
    ...
  }
  catch (Exception e2)
  {
    // Сгенерировать исключение, которое записывает новое
    // исключение, а также сообщение из первого исключения.

    throw new CarIsDeadException(e.Message, e2);
  }
}
```

Обратите внимание, что в данном случае конструктору `CarIsDeadException` во втором параметре передается объект `FileNotFoundException`. После настройки этого нового объекта он передается вверх по стеку вызовов следующему вызывающему коду, которым в рассматриваемой ситуации будет метод `Main()`.

Поскольку после `Main()` нет "следующего вызывающего кода", который мог бы перехватить исключение, пользователю будет отображено системное диалоговое окно с сообщением об ошибке. Подобно повторной генерации исключения запись внутренних исключений обычно полезна, только если вызывающий код способен обработать исключение более элегантно. В таком случае внутри логики `catch` вызывающего кода можно использовать свойство `InnerException` для извлечения деталей внутреннего исключения.

Блок `finally`

В области действия `try/catch` можно также определять дополнительный блок `finally`. Целью блока `finally` является обеспечение того, что заданный набор операторов будет выполняться *всегда* независимо от того, возникло исключение (любого типа) или нет. Для иллюстрации предположим, что перед выходом из метода `Main()` радиоприемник в автомобиле должен всегда выключаться вне зависимости от обрабатываемого исключения:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  myCar.CrankTunes(true);
  try
  {
    // Логика, связанная с увеличением скорости автомобиля.
  }
  catch(CarIsDeadException e)
  {
    // Обработать объект CarIsDeadException.
  }
```

```
  catch(ArgumentOutOfRangeException e)
  {
    // Обработать объект ArgumentOutOfRangeException.
  }
  catch(Exception e)
  {
    // Обработать любой другой объект Exception.
  }
  finally
  {
    // Это код будет выполняться всегда независимо
    // от того, возникало исключение или нет.
    myCar.CrankTunes(false);
  }
  Console.ReadLine();
}
```

Если вы не определите блок `finally`, то в случае генерации исключения радиоприемник не выключится (что может быть или не быть проблемой). В более реалистичном сценарии, когда необходимо освободить объекты, закрыть файл либо отсоединиться от базы данных (или чего-то подобного), блок `finally` представляет собой подходящее место для выполнения надлежащей очистки.

Фильтры исключений

В текущем выпуске языка C# введена новая (и совершенно необязательная) конструкция, которая может быть помещена в блок `catch` посредством ключевого слова `when`. В случае ее добавления появляется возможность обеспечить выполнение операторов внутри блока `catch` только при удовлетворении некоторого условия в коде. Выражение условия должно давать в результате булевское значение (`true` или `false`) и может быть указано с применением простого выражения в самом определении `when` или за счет вызова дополнительного метода в коде. Коротко говоря, такой подход позволяет добавлять "фильтры" к логике исключения.

Для начала предположим, что в класс `CarIsDeadException` добавлено несколько специальных свойств:

```
public class CarIsDeadException : ApplicationException
{
  ...
  // Специальные члены для исключения.
  public DateTime ErrorTimeStamp { get; set; }
  public string CauseOfError { get; set; }

  public CarIsDeadException(string message,
     string cause, DateTime time)
    : base(message)
  {
    CauseOfError = cause;
    ErrorTimeStamp = time;
  }
}
```

Также предположим, что в методе `Accelerate()` для генерации исключения используется следующий новый конструктор:

```
CarIsDeadException ex =
  new CarIsDeadException(string.Format("{0} has overheated!", PetName),
                         "You have a lead foot", DateTime.Now);
```

А теперь взгляните на показанную ниже модифицированную логику исключения. Здесь к обработчику `CarIsDeadException` добавлена *конструкция when*, которая гарантирует, что данный блок `catch` никогда не будет выполняться в пятницу (конечно, этот пример надуман, но кто захочет разбирать автомобиль в выходные?). Обратите внимание, что одиночное булевское выражение в конструкции `when` должно быть помещено в круглые скобки (кроме того, теперь внутри этого блока выводится новое сообщение, что будет происходить, только когда условие `when` дает `true`).

```
catch (CarIsDeadException e) when (e.ErrorTimeStamp.DayOfWeek != DayOfWeek.Friday)
{
  // Выводится, только если выражение в конструкции when вычисляется как true.
  Console.WriteLine("Catching car is dead!");
  Console.WriteLine(e.Message);
}
```

Несмотря на то что вы, скорее всего, просто предусмотрите блок `catch` для перехвата заданной ошибки при любых условиях, как видите, новое ключевое слово `when` позволяет добиться большей степени детализации при реагировании на ошибки времени выполнения.

Отладка необработанных исключений с использованием Visual Studio

Имейте в виду, что среда Visual Studio предлагает набор инструментов, которые помогают отлаживать необработанные специальные исключения. В целях иллюстрации предположим, что мы увеличили скорость объекта `Car` до значения, превышающего максимум, но на этот раз не позаботились о помещении вызова внутрь блока `try`:

```
Car myCar = new Car("Rusty", 90);
myCar.Accelerate(2000);
```

Если вы запустите сеанс отладки в Visual Studio (выбрав пункт меню Debug⇨Start (Отладка⇨Начать)), то во время генерации необработанного исключения произойдет автоматический останов. Более того, откроется окно (рис. 7.2), отображающее значение свойства `Message`.

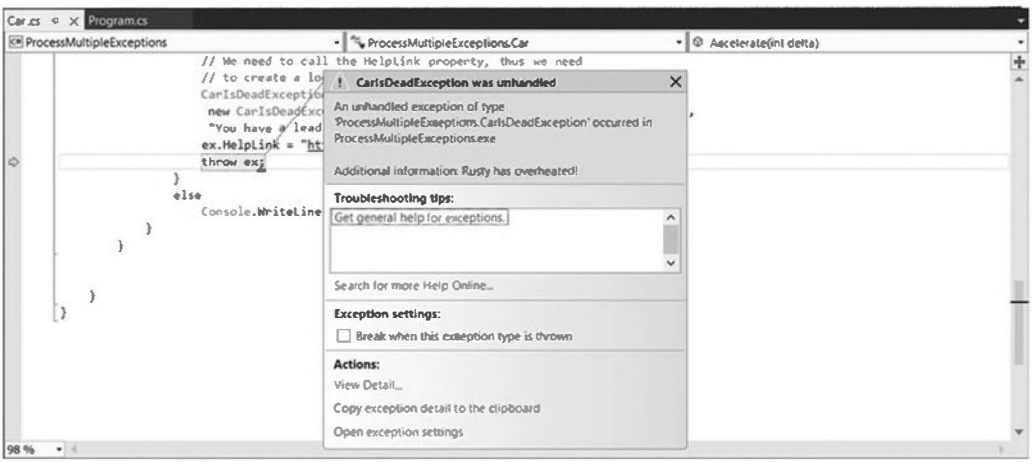


Рис. 7.2. Отладка необработанных специальных исключений в Visual Studio

На заметку! Если вы не обработали исключение, сгенерированное методом из библиотек базовых классов .NET, то отладчик Visual Studio остановит выполнение на операторе, который вызвал этот проблемный метод.

Щелкнув на ссылке View Detail (Показать подробности) в этом окне, вы обнаружите подробную информацию о состоянии объекта (рис. 7.3).

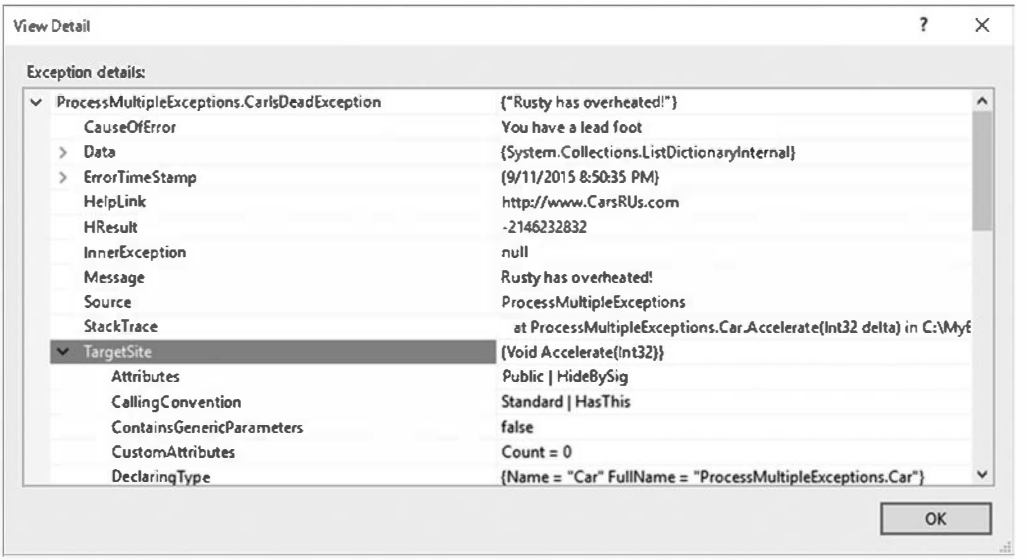


Рис. 7.3. Просмотр деталей исключения

Исходный код. Проект `ProcessMultipleExceptions` доступен в подкаталоге `Chapter_7`.

Резюме

В этой главе была раскрыта роль структурированной обработки исключений. Когда методу необходимо отправить объект ошибки вызывающему коду, он должен создать, сконфигурировать и сгенерировать специфичный объект производного от `System.Exception` типа посредством ключевого слова `throw` языка C#. Вызывающий код может обрабатывать любые входные исключения с применением ключевого слова `catch` и необязательного блока `finally`. Как было показано, версия C# 6.0 теперь поддерживает возможность создания фильтров исключений с использованием дополнительного ключевого слова `when`.

Когда вы строите собственные специальные исключения, в конечном итоге вы создаете класс, производный от класса `System.ApplicationException`, который обозначает исключение, генерируемое текущим выполняющимся приложением. В противоположность этому объекты ошибок, производные от класса `System.SystemException`, представляют критические (и фатальные) ошибки, генерируемые средой CLR. Наконец, в главе были продемонстрированы разнообразные инструменты среды Visual Studio, которые можно применять для создания специальных исключений (согласно установившейся практике .NET), а также для отладки необработанных исключений.

ГЛАВА 8

Работа с интерфейсами

Материал этой главы основан на ваших текущих знаниях объектно-ориентированной разработки и посвящен теме программирования на основе интерфейсов. Вы узнаете, как определять и реализовывать интерфейсы, а также ознакомитесь с преимуществами построения типов, которые поддерживают несколько линий поведения. В ходе изложения также обсуждаются связанные темы, такие как получение ссылок на интерфейсы, явная реализация интерфейсов и построение иерархий интерфейсов. Будет исследовано несколько стандартных интерфейсов, определенных внутри библиотек базовых классов .NET. Как вы увидите, специальные классы и структуры могут реализовывать эти предопределенные интерфейсы для поддержки ряда полезных аспектов поведения, включая клонирование, перечисление и сортировку объектов.

Понятие интерфейсных типов

Для начала давайте ознакомимся с формальным определением *интерфейсного типа*. Интерфейс представляет собой всего лишь именованный набор *абстрактных членов*. Вспомните из главы 6, что абстрактные методы являются чистым протоколом, поскольку они не предоставляют свои стандартные реализации. Специфичные члены, определяемые интерфейсом, зависят от того, какое точно поведение он моделирует. Другими словами, интерфейс выражает *поведение*, которое заданный класс или структура может избрать для поддержки. Более того, как вы увидите далее в главе, класс или структура может реализовывать столько интерфейсов, сколько необходимо, посредством чего поддерживая по существу множество линий поведения.

Как и можно было предположить, библиотеки базовых классов .NET поставляются с многочисленными предопределенными интерфейсными типами, которые реализуются разнообразными классами и структурами. Например, как будет показано в главе 21, инфраструктура ADO.NET содержит множество поставщиков данных, которые позволяют взаимодействовать с определенной системой управления базами данных. Таким образом, в ADO.NET на выбор доступен обширный набор классов подключений (`SqlConnection`, `OleDbConnection`, `OdbcConnection` и т.д.). Кроме того, независимые поставщики баз данных (а также многие проекты с открытым кодом) предоставляют библиотеки .NET для взаимодействия с большим числом других баз данных (MySQL, Oracle и т.д.), которые содержат объекты, реализующие упомянутые интерфейсы.

Невзирая на тот факт, что каждый класс подключения имеет уникальное имя, определен в отдельном пространстве имен и (в некоторых случаях) упакован в отдельную сборку, все они реализуют общий интерфейс под названием `IDbConnection`:

```
// Интерфейс IDbConnection определяет общий набор членов,
// поддерживаемых всеми классами подключения.
public interface IDbConnection : IDisposable
{
```

```
    // Методы.
    IDbTransaction BeginTransaction();
    IDbTransaction BeginTransaction(IsolationLevel il);
    void ChangeDatabase(string databaseName);
    void Close();
    IDbCommand CreateCommand();
    void Open();

    // Свойства.
    string ConnectionString { get; set;}
    int ConnectionTimeout { get; }
    string Database { get; }
    ConnectionState State { get; }
}
```

На заметку! По соглашению имена всех интерфейсов .NET снабжаются префиксом в виде заглавной буквы `I`. При создании собственных специальных интерфейсов рекомендуется также следовать этому соглашению.

В настоящий момент детали того, что на самом деле делают эти члены, не важны. Просто запомните, что интерфейс `IDbConnection` определяет набор членов, которые являются общими для всех классов подключений ADO.NET. В итоге каждый класс подключения гарантированно поддерживает такие члены, как `Open()`, `Close()`, `CreateCommand()` и т.д. Более того, поскольку методы этого интерфейса всегда абстрактные, в каждом классе подключения они могут быть реализованы уникальным образом.

В оставшихся главах книги вы встретите десятки интерфейсов, поставляемых в библиотеках базовых классов .NET. Вы увидите, что эти интерфейсы могут быть реализованы в собственных специальных классах и структурах для определения типов, которые тесно интегрированы с платформой .NET. Вдобавок, как только вы оцените полезность интерфейсных типов, вы определенно найдете причины для построения собственных таких типов.

Сравнение интерфейсных типов и абстрактных базовых классов

Учитывая материалы главы 6, интерфейсный тип может выглядеть кое в чем похожим на абстрактный базовый класс. Вспомните, что когда класс помечен как абстрактный, он *может* определять любое количество абстрактных членов для предоставления полиморфного интерфейса всем производным типам. Однако даже если класс действительно определяет набор абстрактных членов, он также может определять любое количество конструкторов, полей данных, неабстрактных членов (с реализацией) и т.д. С другой стороны, интерфейсы содержат *только* определения членов.

Полиморфный интерфейс, устанавливаемый абстрактным родительским классом, обладает одним серьезным ограничением: члены, определенные абстрактным родительским классом, поддерживаются *только производными типами*. Тем не менее, в крупных программных системах часто разрабатываются многочисленные иерархии классов, не имеющие общего родителя кроме `System.Object`. Учитывая, что абстрактные члены в абстрактном базовом классе применимы только к производным типам, не существует какого-то способа конфигурирования типов в разных иерархиях для поддержки одного и того же полиморфного интерфейса. В качестве примера предположим, что определен следующий абстрактный класс:

```
public abstract class CloneableType
{
  // Поддерживать этот "полиморфный интерфейс" могут только производные типы.
  // Классы в других иерархиях не имеют доступа к этому абстрактному члену.
  public abstract object Clone();
}
```

При таком определении поддерживать метод `Clone()` способны только классы, расширяющие `CloneableType`. Если создается новый набор классов, которые не расширяют данный базовый класс, то извлечь пользу от такого полиморфного интерфейса не удастся. К тому же вы можете вспомнить, что язык C# не поддерживает множественное наследование для классов. По этой причине, если вы хотите создать класс `MiniVan`, который является и `Car`, и `CloneableType`, то поступить так, как показано ниже, не получится:

```
// Недопустимо! Множественное наследование для классов в C# не поддерживается.
public class MiniVan : Car, CloneableType
{
}
```

Как и можно было догадаться, здесь на помощь приходят интерфейсные типы. После того как интерфейс определен, он может быть реализован любым классом либо структурой, в любой иерархии и внутри любого пространства имен или сборки (написанной на любом языке программирования .NET). Как видите, интерфейсы являются *чрезвычайно* полиморфными. Рассмотрим стандартный интерфейс .NET под названием `ICloneable`, определенный в пространстве имен `System`. В этом интерфейсе определен единственный метод по имени `Clone()`:

```
public interface ICloneable
{
  object Clone();
}
```

Заглянув в документацию .NET Framework 4.6 SDK, вы обнаружите, что интерфейс `ICloneable` реализован очень многими на вид несвязанными типами (`System.Array`, `System.Data.SqlClient.SqlConnection`, `System.OperatingSystem`, `System.String` и т.д.). Хотя эти типы не имеют общего родителя (кроме `System.Object`), их можно обрабатывать полиморфным образом посредством интерфейсного типа `ICloneable`.

Например, если есть метод по имени `CloneMe()`, принимающий параметр типа `ICloneable`, то такому методу можно передавать любой объект, который реализует указанный интерфейс. Рассмотрим следующий простой класс `Program`, определенный в проекте консольного приложения по имени `ICloneableExample`:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** A First Look at Interfaces *****\n");

    // Все эти классы поддерживают интерфейс ICloneable.
    string myStr = "Hello";
    OperatingSystem unixOS = new OperatingSystem(PlatformID.Unix, new Version());
    System.Data.SqlClient.SqlConnection sqlCnn =
      new System.Data.SqlClient.SqlConnection();

    // Следовательно, все они могут быть переданы методу,
    // принимающему параметр типа ICloneable.
    CloneMe(myStr);
```

```
    CloneMe(unixOS);
    CloneMe(sqlCnn);
    Console.ReadLine();
  }
  private static void CloneMe(ICloneable c)
  {
    // Клонировать то, что получено, и вывести его имя.
    object theClone = c.Clone();
    Console.WriteLine("Your clone is a: {0}",
      theClone.GetType().Name);
  }
}
```

После запуска приложения в окне консоли выводится имя каждого класса, полученное с помощью метода `GetType()`, который унаследован от `System.Object`. Как объясняется в главе 15, этот метод и службы рефлексии .NET позволяют разбирать строение любого типа во время выполнения. Ниже показан вывод предыдущей программы:

```
***** A First Look at Interfaces *****

Your clone is a: String
Your clone is a: OperatingSystem
Your clone is a: SqlConnection
```

Исходный код. Проект `ICloneableExample` доступен в подкаталоге `Chapter_8`.

Еще одно ограничение абстрактных базовых классов связано с тем, что *каждый производный тип* должен предоставлять реализацию для всего набора абстрактных членов. Чтобы увидеть, в чем заключается проблема, вспомним иерархию фигур, которая была определена в главе 6. Предположим, что в базовом классе `Shape` определен новый абстрактный метод по имени `GetNumberOfPoints()`, который позволяет производным типам возвращать количество вершин, требуемых для визуализации фигуры:

```
abstract class Shape
{
  ...
  // Каждый производный класс теперь должен поддерживать этот метод!
  public abstract byte GetNumberOfPoints();
}
```

Очевидно, что единственным классом, который в принципе имеет вершины, будет `Hexagon`. Однако теперь из-за внесенного обновления *каждый* производный класс (`Circle`, `Hexagon` и `ThreeDCircle`) обязан предоставить конкретную реализацию метода `GetNumberOfPoints()`, даже если в этом нет никакого смысла. И снова интерфейсный тип предлагает решение. Если вы определите интерфейс, который представляет поведение "наличия вершин", то можно будет просто подключить его к классу `Hexagon`, оставив классы `Circle` и `ThreeDCircle` незатронутыми.

Определение специальных интерфейсов

Теперь, когда вы лучше понимаете общую роль интерфейсов, давайте рассмотрим пример определения и реализации специальных интерфейсов. Для начала создадим новый проект консольного приложения по имени `CustomInterface`. С помощью пункта меню Project⇨Add Existing Item (Проект⇨Добавить существующий элемент) вставим в него файл (или файлы) с определениями типов фигур (`Shapes.cs` в примерах кода), которые были созданы в главе 6. После этого переименуем пространство имен, в котором

определяются типы, связанные с фигурами, на CustomInterface (просто чтобы избежать импортирования в новый проект определений пространства имен):

```
namespace CustomInterface
{
  // Здесь определяются типы фигур...
}
```

Теперь, выбрав пункт меню Project⇨Add Existing Item, вставим в проект новый интерфейс по имени IPointy (рис. 8.1).

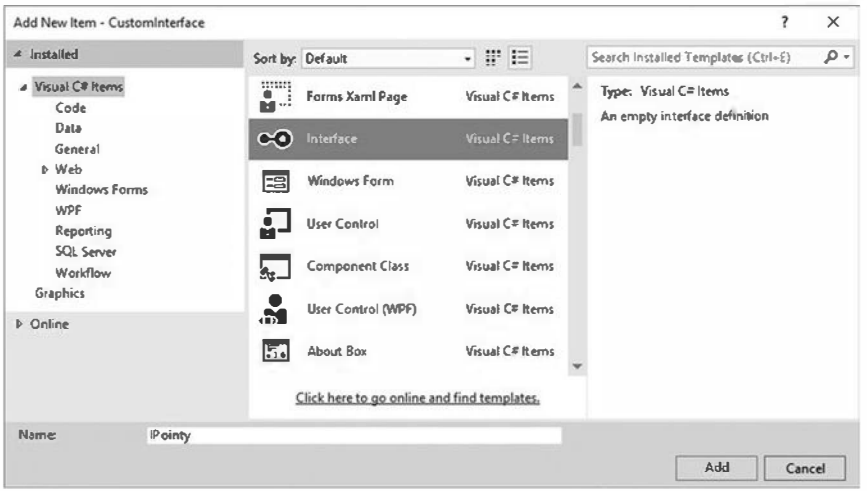


Рис. 8.1. Интерфейсы подобно классам могут определяться в файлах *.cs

На синтаксическом уровне интерфейс определяется с использованием ключевого слова interface языка C#. В отличие от классов для интерфейсов никогда не указывается базовый класс (даже System.Object; тем не менее, как будет показано позже в главе, можно задавать базовые интерфейсы). Кроме того, для членов интерфейса никогда не указываются модификаторы доступа (т.к. все члены интерфейса являются неявно открытыми и абстрактными). Ниже приведен пример определения специального интерфейса в C#:

```
// Этот интерфейс определяет поведение "наличия вершин".
public interface IPointy
{
  // Неявно открытый и абстрактный.
  byte GetNumberOfPoints();
}
```

Запомните, что при определении членов интерфейса область реализации для них не определяется. Интерфейсы — это чистый протокол, поэтому реализация для них никогда не предоставляется (за это отвечает поддерживающий класс или структура). Следовательно, показанная далее версия интерфейса IPointy дает в результате разнообразные ошибки на этапе компиляции:

```
// Внимание! В этом коде полно ошибок!
public interface IPointy
{
  // Ошибка! Интерфейсы не могут иметь поля данных!
  public int numbOfPoints;
```

```
  // Ошибка! Интерфейсы не могут иметь конструкторы!
  public IPointy() { numbOfPoints = 0;}

  // Ошибка! Интерфейсы не могут предоставлять реализацию своих членов!
  byte GetNumberOfPoints() { return numbOfPoints; }
}
```

В любом случае эта начальная версия интерфейса `IPointy` определяет единственный метод. Однако в интерфейсных типах .NET допускается также определять любое количество прототипов свойств. Например, модифицируем интерфейс `IPointy` так, чтобы в нем применялось свойство только для чтения, а не традиционный метод доступа:

```
// Поведение "наличия вершин" в виде свойства только для чтения.
public interface IPointy
{
  // Свойство, поддерживающее чтение и запись, в интерфейсе может выглядеть так:
  // retType PropName { get; set; }
  //
  // тогда как свойство только для записи - так:
  // retType PropName { set; }

  byte Points { get; }
}
```

На заметку! Интерфейсные типы также могут содержать определения событий (глава 10) и индексаторов (глава 11).

Сами по себе типы интерфейсов совершенно бесполезны, потому что они являются всего лишь именованными коллекциями абстрактных членов. Например, выделять память для типов интерфейсов, как это делается для классов или структур, невозможно:

```
// Внимание! Выделять память для типов недопустимо!
static void Main(string[] args)
{
  IPointy p = new IPointy(); // Ошибка на этапе компиляции!
}
```

Интерфейсы не привносят ничего особого до тех пор, пока не будут реализованы классом или структурой. Здесь `IPointy` представляет собой интерфейс, который выражает поведение "наличия вершин". Идея проста: одни классы в иерархии фигур (например, `Hexagon`) имеют вершины, в то время как другие (вроде `Circle`) — нет.

Реализация интерфейса

Когда решено расширить функциональность класса (или структуры) путем поддержки интерфейсов, к его определению добавляется список нужных интерфейсов, разделенных запятыми. Имейте в виду, что непосредственный базовый класс должен быть указан первым сразу после операции двоеточия. Если тип класса порождается напрямую от `System.Object`, то вы можете просто перечислить интерфейсы, поддерживаемые классом, т.к. компилятор C# будет считать, что типы расширяют `System.Object`, если не задано иначе. К слову, поскольку структуры всегда являются производными от класса `System.ValueType` (см. главу 4), достаточно указать список интерфейсов после определения структуры. Взгляните на приведенные ниже примеры:

```
// Этот класс является производными от System.Object и реализует
единственный интерфейс.
public class Pencil : IPointy
{...}
```

```
// Этот класс также является производными от System.Object
// и реализует единственный интерфейс.
public class SwitchBlade : object, IPointy
{...}

// Этот класс является производными от специального базового класса
// и реализует единственный интерфейс.
public class Fork : Utensil, IPointy
{...}

// Эта структура неявно является производной от System.ValueType
// и реализует два интерфейса.
public struct PitchFork : ICloneable, IPointy
{...}
```

Важно понимать, что реализация интерфейса работает по принципу "все или ничего". Поддерживающий тип не имеет возможности выборочно решать, какие члены он будет реализовывать. Учитывая, что интерфейс `IPointy` определяет единственное свойство только для чтения, накладные расходы невелики. Тем не менее, если вы реализуете интерфейс, который определяет десять членов (вроде показанного ранее `IDbConnection`), то тип отвечает за предоставление деталей для всех десяти абстрактных членов.

Вернемся к рассматриваемому примеру и добавим в проект новый тип класса по имени `Triangle`, который является `Shape` и поддерживает `IPointy`. Обратите внимание, что реализация доступного только для чтения свойства `Points` просто возвращает корректное количество вершин (3):

```
// Новый производный от Shape класс по имени Triangle.
class Triangle : Shape, IPointy
{
  public Triangle() { }
  public Triangle(string name) : base(name) { }
  public override void Draw()
  { Console.WriteLine("Drawing {0} the Triangle", PetName); }

  // Реализация IPointy.
  public byte Points
  {
    get { return 3; }
  }
}
```

Теперь модифицируем существующий тип `Hexagon`, чтобы он также поддерживал интерфейс `IPointy`:

```
// Hexagon теперь реализует IPointy.
class Hexagon : Shape, IPointy
{
  public Hexagon(){ }
  public Hexagon(string name) : base(name){ }
  public override void Draw()
  { Console.WriteLine("Drawing {0} the Hexagon", PetName); }

  // Реализация IPointy.
  public byte Points
  {
    get { return 6; }
  }
}
```

Чтобы подытожить все сделанное к этому моменту, на рис. 8.2 приведена диаграмма классов Visual Studio, где все совместимые с `IPointy` классы представлены с помощью популярной системы обозначений в виде "леденца на палочке". Еще раз обратите внимание, что `Circle` и `ThreeDCircle` не реализуют `IPointy`, поскольку такое поведение в данных классах не имеет смысла.

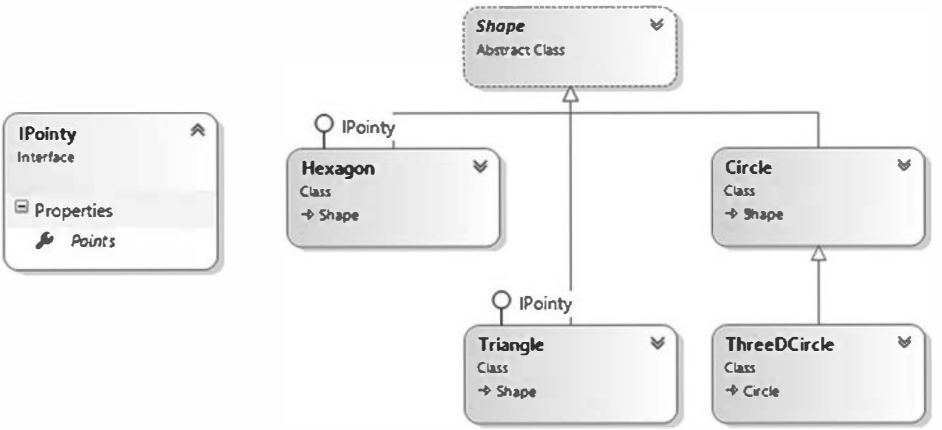


Рис. 8.2. Иерархия фигур, теперь с интерфейсами

На заметку! Чтобы скрыть или отобразить имена интерфейсов в визуальном конструкторе классов, щелкните правой кнопкой мыши на значке, представляющем интерфейс, и выберите в контекстном меню пункт Collapse (Свернуть) или Expand (Развернуть).

Обращение к членам интерфейса на уровне объектов

Теперь, имея несколько классов, которые поддерживают интерфейс `IPointy`, необходимо выяснить, каким образом взаимодействовать с новой функциональностью. Самый простой способ взаимодействия с функциональностью, предоставляемой заданным интерфейсом, заключается в обращении к его членам прямо на уровне объектов (при условии, что члены интерфейса не реализованы явно, о чем более подробно пойдет речь в разделе "Явная реализация интерфейсов" далее в главе). Например, рассмотрим следующий метод `Main()`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");

  // Обратиться к свойству Points, определенному в интерфейсе IPointy.
  Hexagon hex = new Hexagon();
  Console.WriteLine("Points: {0}", hex.Points);
  Console.ReadLine();
}
```

Такой подход нормально работает в этом конкретном случае, поскольку здесь точно известно, что тип `Hexagon` реализует упомянутый интерфейс и, следовательно, имеет свойство `Points`. Однако в других случаях определить, какие интерфейсы поддерживаются заданным типом, может быть нереально. Для примера предположим, что есть

массив, содержащий 50 объектов совместимых с Shape типов, причем только некоторые из них поддерживают интерфейс IPointy. Очевидно, что если вы попытаетесь обратиться к свойству Points для типа, который не реализует IPointy, то возникнет ошибка. Как же динамически определить, поддерживает ли класс или структура подходящий интерфейс?

Один из способов выяснить во время выполнения, поддерживает ли тип конкретный интерфейс, предусматривает использование явного приведения. Если тип не поддерживает запрашиваемый интерфейс, то генерируется исключение InvalidCastException. В случае подобного рода необходимо использовать структурированную обработку исключений:

```
static void Main(string[] args)
{
  ...
  // Перехватить возможное исключение InvalidCastException.
  Circle c = new Circle("Lisa");
  IPointy itfPt = null;
  try
  {
    itfPt = (IPointy)c;
    Console.WriteLine(itfPt.Points);
  }
  catch (InvalidCastException e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

Хотя можно было бы применить логику try/catch и надеяться на лучшее, в идеале хотелось бы определять, какие интерфейсы поддерживаются, до обращения к их членам. Давайте рассмотрим два способа, с помощью которых этого можно добиться.

Получение ссылок на интерфейсы: ключевое слово as

Для определения, поддерживает ли данный тип тот или иной интерфейс, можно использовать ключевое слово as, которое было представлено в главе 6. Если объект может трактоваться как указанный интерфейс, то возвращается ссылка на интересующий интерфейс, а если нет, то ссылка null. Таким образом, перед продолжением в коде необходимо реализовать проверку на предмет null:

```
static void Main(string[] args)
{
  ...
  // Можно ли hex2 трактовать как IPointy?
  Hexagon hex2 = new Hexagon("Peter");
  IPointy itfPt2 = hex2 as IPointy;

  if(itfPt2 != null)
    Console.WriteLine("Points: {0}", itfPt2.Points);
  else
    Console.WriteLine("OOPS! Not pointy...");
  Console.ReadLine();
}
```

Обратите внимание, что когда применяется ключевое слово `as`, отпадает необходимость в наличии логики `try/catch`, т.к. если ссылка не является `null`, то известно, что вызов происходит для действительной ссылки на интерфейс.

Получение ссылок на интерфейсы: ключевое слово `is`

Проверить, реализован ли нужный интерфейс, можно также с помощью ключевого слова `is` (которое впервые упоминалось в главе 6). Если интересующий объект *не* совместим с указанным интерфейсом, то возвращается значение `false`. С другой стороны, если тип совместим с интерфейсом, то можно безопасно обращаться к его членам без использования логики `try/catch`.

В целях иллюстрации предположим, что имеется массив типов `Shape`, в котором определенные элементы реализуют интерфейс `IPointy`. Вот как с помощью ключевого слова `is` выяснить, какие элементы в массиве поддерживают данный интерфейс:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");

  // Создать массив элементов Shape.
  Shape[] myShapes = { new Hexagon(), new Circle(),
                       new Triangle("Joe"), new Circle("JoJo")} ;

  for(int i = 0; i < myShapes.Length; i++)
  {
    // Вспомните, что базовый класс Shape определяет абстрактный
    // член Draw(), поэтому все фигуры знают, как себя рисовать.
    myShapes[i].Draw();

    // У каких фигур есть вершины?
    if(myShapes[i] is IPointy)
      Console.WriteLine("-> Points: {0}", ((IPointy) myShapes[i]).Points);
    else
      Console.WriteLine("-> {0}\'s not pointy!", myShapes[i].PetName);
     Console.WriteLine();
  }
  Console.ReadLine();
}
```

Вывод выглядит следующим образом:

```
***** Fun with Interfaces *****

Drawing NoName the Hexagon
-> Points: 6

Drawing NoName the Circle
-> NoName's not pointy!

Drawing Joe the Triangle
-> Points: 3

Drawing JoJo the Circle
-> JoJo's not pointy!
```

Использование интерфейсов в качестве параметров

Учитывая, что интерфейсы — допустимые типы .NET, можно строить методы, которые принимают интерфейсы в качестве параметров, как это было с показанным ранее методом `CloneMe()`.

Предположим, что в текущем примере определен еще один интерфейс по имени IDraw3D:

```
// Моделирует способность визуализации типа в трехмерном виде.
public interface IDraw3D
{
  void Draw3D();
}
```

Далее сконфигурируем две из трех наших фигур (Circle и Hexagon) с целью поддержки нового поведения:

```
// Circle поддерживает IDraw3D.
class ThreeDCircle : Circle, IDraw3D
{
  ...
  public void Draw3D()
  { Console.WriteLine("Drawing Circle in 3D!"); }
}
// Hexagon поддерживает IPointy и IDraw3D.
class Hexagon : Shape, IPointy, IDraw3D
{
  ...
  public void Draw3D()
  { Console.WriteLine("Drawing Hexagon in 3D!"); }
}
```

На рис. 8.3 показана обновленная диаграмма классов Visual Studio.

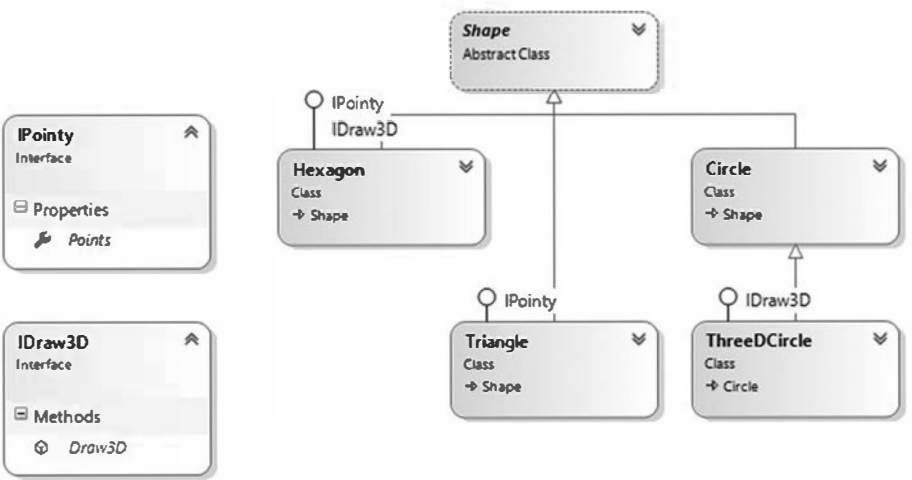


Рис. 8.3. Обновленная иерархия фигур

Если вы определите метод, принимающий интерфейс IDraw3D в качестве параметра, то ему можно будет передавать по существу любой объект, реализующий IDraw3D. (Попытка передачи типа, не поддерживающего необходимый интерфейс, приводит ошибке на этапе компиляции.) Взгляните на следующий метод, определенный в классе Program:

```
// Будет рисовать любую фигуру, поддерживающую IDraw3D.
static void DrawIn3D(IDraw3D itf3d)
{
  Console.WriteLine("-> Drawing IDraw3D compatible type");
  itf3d.Draw3D();
}
```

Теперь можно проверить, поддерживает ли элемент в массиве Shape новый интерфейс, и если это так, то передать его методу DrawIn3D() на обработку:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");
  Shape[] myShapes = { new Hexagon(), new Circle(),
                       new Triangle("Joe"), new Circle("JoJo") } ;
  for(int i = 0; i < myShapes.Length; i++)
  {
    ...
    // Можно ли нарисовать эту фигуру в трехмерном виде?
    if(myShapes[i] is IDraw3D)
      DrawIn3D((IDraw3D)myShapes[i]);
  }
}
```

Ниже представлен вывод, полученный из модифицированной версии приложения. Обратите внимание, что в трехмерном виде отображается только объект Hexagon, т.к. все остальные члены массива Shape не реализуют интерфейс IDraw3D:

```
***** Fun with Interfaces *****

Drawing NoName the Hexagon
-> Points: 6
-> Drawing IDraw3D compatible type
Drawing Hexagon in 3D!

Drawing NoName the Circle
-> NoName's not pointy!

Drawing Joe the Triangle
-> Points: 3

Drawing JoJo the Circle
-> JoJo's not pointy!
```

Использование интерфейсов в качестве возвращаемых значений

Интерфейсы могут также применяться в качестве типов возвращаемых значений методов. Например, можно было бы написать метод, который получает массив объектов Shape и возвращает ссылку на первый элемент, поддерживающий IPointy:

```
// Этот метод возвращает первый объект в массиве,
// который реализует интерфейс IPointy.
static IPointy FindFirstPointyShape(Shape[] shapes)
{
  foreach (Shape s in shapes)
  {
    if (s is IPointy)
      return s as IPointy;
  }
  return null;
}
```

Взаимодействовать с этим методом можно так:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");
```

```
  // Создать массив элементов Shape.
  Shape[] myShapes = { new Hexagon(), new Circle(),
                       new Triangle("Joe"), new Circle("JoJo")};
  // Получить первый элемент, имеющий вершины.
  // В целях безопасности не помешает проверить firstPointyItem на равенство null.
  IPointy firstPointyItem = FindFirstPointyShape(myShapes);
  Console.WriteLine("The item has {0} points", firstPointyItem.Points);
  ...
}
```

Массивы интерфейсных типов

Вспомните, что один и тот же интерфейс может быть реализован множеством типов, даже если они не находятся внутри той же самой иерархии классов и не имеют общего родительского класса помимо `System.Object`. Это позволяет формировать очень мощные программные конструкции. Например, пусть в текущем проекте разработаны три новых класса: два класса (`Knife` (нож) и `Fork` (вилка)) моделируют кухонные приборы, а третий (`PitchFork` (вилы)) — садовый инструмент (рис. 8.4).

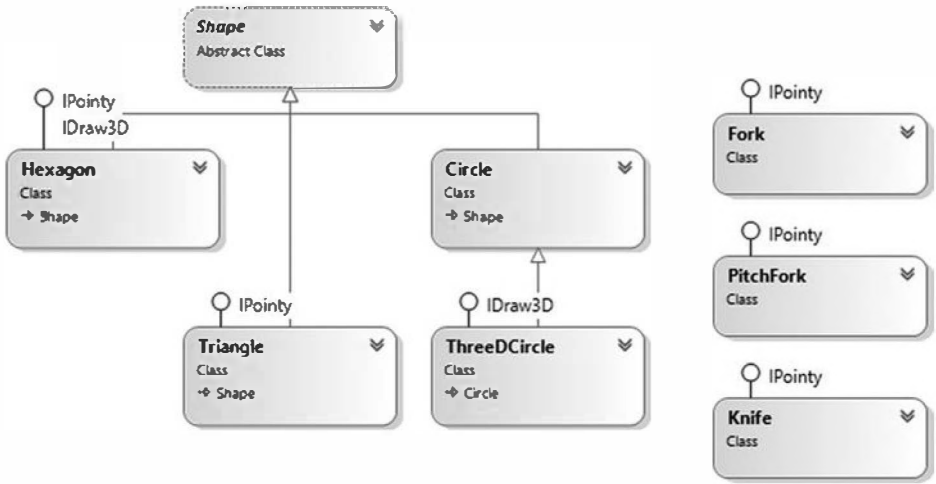


Рис. 8.4. Интерфейсы могут "подключаться" к любому типу внутри любой части иерархии классов

После того, как типы `PitchFork`, `Fork` и `Knife` определены, можно определить массив объектов, совместимых с `IPointy`. Поскольку все элементы поддерживают один и тот же интерфейс, можно выполнять проход по массиву и интерпретировать каждый его элемент как объект, совместимый с `IPointy`, несмотря на разнородность иерархий классов:

```
static void Main(string[] args)
{
  ...
  // Этот массив может содержать только типы,
  // которые реализуют интерфейс IPointy.
  IPointy[] myPointyObjects = {new Hexagon(), new Knife(),
    new Triangle(), new Fork(), new PitchFork()};
  foreach(IPointy i in myPointyObjects)
  Console.WriteLine("Object has {0} points.", i.Points);
  Console.ReadLine();
}
```

Просто чтобы подчеркнуть важность этого примера, запомните, что массив заданного интерфейсного типа может содержать элементы любых классов или структур, реализующих этот интерфейс.

Исходный код. Проект `CustomInterface` доступен в подкаталоге `Chapter_8`.

Реализация интерфейсов с использованием Visual Studio

Хотя программирование на основе интерфейсов является мощным приемом, реализация интерфейсов может быть сопряжена с довольно большим объемом клавиатурного ввода. Учитывая, что интерфейсы являются именованными наборами абстрактных членов, для *каждого* метода интерфейса в *каждом* типе, который поддерживает это поведение, потребуется вводить определение и реализацию. Следовательно, если вы хотите поддерживать интерфейс, который определяет пять методов и три свойства, то придется принять во внимание все восемь членов (иначе возникнут ошибки на этапе компиляции).

К счастью, в Visual Studio поддерживаются разнообразные инструменты, упрощающие задачу реализации интерфейсов. Для примера вставим в текущий проект еще один класс по имени `PointyTestClass`. Когда вы добавите к типу класса интерфейс, такой как `IPointy` (или любой другой подходящий интерфейс), то заметите, что по окончании ввода имени интерфейса (или при помещении на него курсора мыши в окне редактора кода) первая буква имени выделяется подчеркиванием (формально это называется *смарт-тегом*). Щелчок на смарт-теге приводит к отображению раскрывающегося списка, который позволяет реализовать этот интерфейс (рис. 8.5).

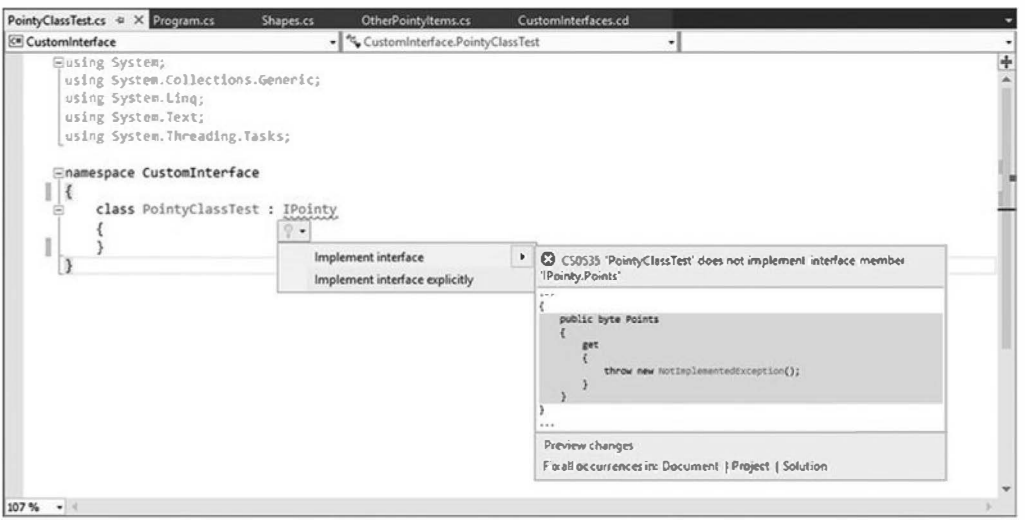


Рис. 8.5. Реализация интерфейсов в Visual Studio

Обратите внимание, что в этом списке предлагаются два пункта, из которых второй (явная реализация интерфейса) будет обсуждаться в следующем разделе. Для начала выберем первый пункт. Среда Visual Studio сгенерирует код заглушки, подлежащий обновлению (как видите, стандартная реализация генерирует исключение `System.NotImplementedException`, что очевидно можно удалить):

```
namespace CustomInterface
{
  class PointyTestClass : IPointy
  {
    public byte Points
    {
      get { throw new NotImplementedException(); }
    }
  }
}
```

На заметку! Среда Visual Studio также поддерживает рефакторинг в виде извлечения интерфейса (Extract Interface), доступный через пункт Extract Interface (Извлечь интерфейс) меню Quick Actions (Быстрые действия). Такой рефакторинг позволяет извлечь новое определение интерфейса из существующего определения класса. Например, вы можете находиться где-то на полпути к завершению написания класса, но вдруг вас осеняет, что данное поведение можно обобщить в виде интерфейса (открывая возможность для альтернативных реализаций).

Явная реализация интерфейсов

Как было показано ранее в главе, класс или структура может реализовывать любое количество интерфейсов. С учетом этого всегда существует возможность реализации интерфейсов, которые содержат члены с идентичными именами, из-за чего придется устранять конфликты имен. Чтобы проиллюстрировать разнообразные способы решения этой проблемы, создадим новый проект консольного приложения по имени `InterfaceNameClash` и добавим в него три специальных интерфейса, представляющих различные места, в которых реализующий их тип может визуализировать свой вывод:

```
// Вывести изображение на форме.
public interface IDrawToForm
{
  void Draw();
}

// Вывести изображение в буфер памяти.
public interface IDrawToMemory
{
  void Draw();
}

// Вывести изображение на принтер.
public interface IDrawToPrinter
{
  void Draw();
}
```

Обратите внимание, что в каждом интерфейсе определен метод по имени `Draw()` с идентичной сигнатурой (без аргументов). Если необходимо поддерживать все эти интерфейсы в одном классе `Octagon`, то компилятор разрешит следующее определение:

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
  public void Draw()
  {
    // Разделяемая логика вывода.
    Console.WriteLine("Drawing the Octagon...");
  }
}
```

Хотя компиляция этого кода пройдет гладко, здесь присутствует потенциальная проблема. Выражаясь просто, предоставление единственной реализации метода `Draw()` не позволяет предпринимать уникальные действия на основе того, какой интерфейс получен от объекта `Octagon`. Например, показанный ниже код будет приводить к вызову того же самого метода `Draw()` независимо от того, какой интерфейс получен:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
  // Все эти обращения приводят к вызову одного и того же метода Draw()!
  Octagon oct = new Octagon();

  IDrawToForm itfForm = (IDrawToForm)oct;
  itfForm.Draw();

  IDrawToPrinter itfPriner = (IDrawToPrinter)oct;
  itfPriner.Draw();

  IDrawToMemory itfMemory = (IDrawToMemory)oct;
  itfMemory.Draw();

  Console.ReadLine();
}
```

Очевидно, что код, требуемый для визуализации изображения в окне, довольно сильно отличается от кода, который необходим для вывода изображения на сетевой принтер или в область памяти. При реализации нескольких интерфейсов, имеющих идентичные члены, разрешить такой конфликт имен можно с применением синтаксиса *явной реализации интерфейсов*. Взгляните на следующую модификацию типа `Octagon`:

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
  // Явно привязать реализации Draw() к конкретным интерфейсам.
  void IDrawToForm.Draw()
  {
    Console.WriteLine("Drawing to form...");      // Вывод на форму
  }
  void IDrawToMemory.Draw()
  {
    Console.WriteLine("Drawing to memory...");    // Вывод в память
  }
  void IDrawToPrinter.Draw()
  {
    Console.WriteLine("Drawing to a printer..."); // Вывод на принтер
  }
}
```

Как видите, при явной реализации члена интерфейса общий шаблон выглядит следующим образом:

```
возвращаемыйТип ИмяИнтерфейса.ИмяМетода(параметры){}
```

Обратите внимание, что при использовании такого синтаксиса модификатор доступа не указывается; явно реализованные члены автоматически будут закрытыми. Например, этот синтаксис недопустим:

```
// Ошибка! Модификатор доступа не может быть указан!
public void IDrawToForm.Draw()
{
  Console.WriteLine("Drawing to form...");
}
```

Поскольку явно реализованные члены всегда неявно закрыты, они перестают быть доступными на уровне объектов. Фактически, если вы примените к типу `Octagon` операцию точки, то обнаружите, что средство IntelliSense не отображает члены `Draw()`. Как и следовало ожидать, для доступа к требуемой функциональности должно использоваться явное приведение. Ниже представлен пример:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
  Octagon oct = new Octagon();

  // Теперь для доступа к членам Draw() должно использоваться приведение.
  IDrawToForm itfForm = (IDrawToForm)oct;
  itfForm.Draw();

  // Сокращенная форма, если переменная интерфейса не нужна.
  ((IDrawToPrinter)oct).Draw();

  // Можно было бы также использовать ключевое слово as.
  if(oct is IDrawToMemory)
    ((IDrawToMemory)oct).Draw();

  Console.ReadLine();
}
```

Наряду с тем, что этот синтаксис действительно полезен, когда необходимо устранить конфликты имен, явную реализацию интерфейсов можно применять и просто для сокрытия более "сложных" членов на уровне объектов. В таком случае при использовании операции точки пользователь объекта будет видеть только подмножество всей функциональности типа. Тем не менее, те, кому требуется более сложное поведение, могут извлекать желаемый интерфейс через явное приведение.

Исходный код. Проект `InterfaceNameClash` доступен в подкаталоге `Chapter_8`.

Проектирование иерархий интерфейсов

Интерфейсы могут быть организованы в иерархии. Подобно иерархии классов, когда интерфейс расширяет существующий интерфейс, он наследует все абстрактные члены, определяемые родителем (или родителями). Конечно, в отличие от наследования на основе классов производный интерфейс никогда не наследует действительную реализацию. Взамен он просто расширяет собственное определение дополнительными абстрактными членами.

Иерархии интерфейсов могут быть удобны, когда нужно расширить функциональность имеющегося интерфейса, не нарушая работу существующих кодовых баз. В целях иллюстрации создадим новый проект консольного приложения по имени `InterfaceHierarchy`. Затем так спроектируем новый набор интерфейсов, связанных с визуализацией, чтобы `IDrawable` был корневым интерфейсом в дереве этого семейства:

```
public interface IDrawable
{
  void Draw();
}
```

Учитывая, что интерфейс `IDrawable` определяет базовое поведение рисования, можно создать производный интерфейс, который расширяет `IDrawable` возможностью визуализации в других форматах, например:

```
public interface IAdvancedDraw : IDrawable
{
  void DrawInBoundingBox(int top, int left, int bottom, int right);
  void DrawUpsideDown();
}
```

При таком проектном решении, если класс реализует интерфейс `IAdvancedDraw`, то ему потребуется реализовать все члены, определенные в цепочке наследования (в частности, методы `Draw()`, `DrawInBoundingBox()` и `DrawUpsideDown()`):

```
public class BitmapImage : IAdvancedDraw
{
  public void Draw()
  {
    Console.WriteLine("Drawing...");
  }
  public void DrawInBoundingBox(int top, int left, int bottom, int right)
  {
    Console.WriteLine("Drawing in a box...");
  }
  public void DrawUpsideDown()
  {
    Console.WriteLine("Drawing upside down!");
  }
}
```

Теперь, когда применяется класс `BitmapImage`, появляется возможность вызова каждого метода на уровне объекта (из-за того, что все они открыты), а также извлекать ссылку на каждый поддерживаемый интерфейс явным образом через приведение:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Interface Hierarchy *****");
  // Вызвать на уровне объекта.
  BitmapImage myBitmap = new BitmapImage();
  myBitmap.Draw();
  myBitmap.DrawInBoundingBox(10, 10, 100, 150);
  myBitmap.DrawUpsideDown();
  // Получить IAdvancedDraw явным образом.
  IAdvancedDraw iAdvDraw = myBitmap as IAdvancedDraw;
  if(iAdvDraw != null)
    iAdvDraw.DrawUpsideDown();
  Console.ReadLine();
}
```

Исходный код. Проект `InterfaceHierarchy` доступен в подкаталоге `Chapter_8`.

Множественное наследование с помощью интерфейсных типов

В отличие от типов классов интерфейс может расширять множество базовых интерфейсов, что позволяет проектировать мощные и гибкие абстракции. Создадим новый проект консольного приложения по имени `MIInterfaceHierarchy`. Здесь имеется еще одна коллекция интерфейсов, которые моделируют разнообразные абстракции, связанные с визуализацией и фигурами. Обратите внимание, что интерфейс `IShape` расширяет и `IDrawable`, и `IPrintable`:

```
// Множественное наследование для интерфейсных типов допускается.
interface IDrawable
{
  void Draw();
}

interface IPrintable
{
  void Print();
  void Draw();  // <-- Возможен конфликт имен!
}

// Множественное наследование интерфейсов. Разрешено!
interface IShape : IDrawable, IPrintable
{
  int GetNumberOfSides();
}
```

На рис. 8.6 показана текущая иерархия интерфейсов.

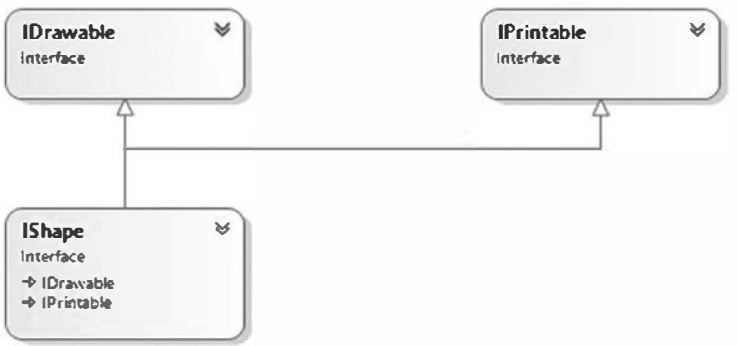


Рис. 8.6. В отличие от классов интерфейсы могут расширять сразу несколько базовых интерфейсов

В этот момент главный вопрос звучит так: если есть класс, поддерживающий `IShape`, то сколько методов он должен реализовать? Ответ: в зависимости от обстоятельств. Если вы хотите предоставить простую реализацию метода `Draw()`, то вам необходимо реализовать только три его члена, как иллюстрируется в следующем классе `Rectangle`:

```
class Rectangle : IShape
{
  public int GetNumberOfSides()
  { return 4; }

  public void Draw()
  { Console.WriteLine("Drawing..."); }

  public void Print()
  { Console.WriteLine("Printing..."); }
}
```

Если вы предпочитаете располагать специфическими реализациями для каждого метода `Draw()` (что в данном случае имеет смысл), то конфликт имен можно разрешить с использованием явной реализации интерфейсов, как это сделано в представленном ниже классе `Square`:

```
class Square : IShape
{
```

```
  // Использование явной реализации для устранения конфликта имен членов.
  void IPrintable.Draw()
  {
    // Вывести на принтер...
  }
  void IDrawable.Draw()
  {
    // Вывести на экран...
  }
  public void Print()
  {
    // Печатать...
  }
  public int GetNumberOfSides()
  { return 4; }
}
```

В идеале к этому моменту вы должны лучше понимать процесс определения и реализации специальных интерфейсов с применением синтаксиса C#. По правде говоря, на привыкание к программированию на основе интерфейсов может уйти определенное время, так что если вы находитесь в некотором недоумении, то это совершенно нормальная реакция.

Однако имейте в виду, что интерфейсы являются фундаментальным аспектом .NET Framework. Независимо от типа разрабатываемого приложения (веб-приложение, настольное приложение с графическим пользовательским интерфейсом, библиотека доступа к данным и т.п.), работа с интерфейсами будет составной частью этого процесса. Подводя итог изложенному выше, запомните, что интерфейсы могут быть исключительно полезны в следующих ситуациях:

- существует единственная иерархия, в которой только подмножество производных типов поддерживает общее поведение;
- необходимо моделировать общее поведение, которое встречается в нескольких иерархиях, не имеющих общего родительского класса кроме `System.Object`.

Итак, вы ознакомились со спецификой построения и реализации специальных интерфейсов. Остаток этой главы посвящен исследованию нескольких предопределенных интерфейсов, содержащихся в библиотеках базовых классов .NET. Как будет показано, вы можете реализовывать стандартные интерфейсы .NET в своих специальных типах, обеспечивая их бесшовную интеграцию с платформой.

Исходный код. Проект `MIInterfaceHierarchy` доступен в подкаталоге `Chapter_8`.

Интерфейсы IEnumerable и IEnumerator

Прежде чем приступать к исследованию процесса реализации существующих интерфейсов .NET, давайте сначала рассмотрим роль интерфейсов `IEnumerable` и `IEnumerator`. Вспомните, что язык C# поддерживает ключевое слово `foreach`, которое позволяет осуществлять проход по содержимому массива любого типа:

```
// Итерация по массиву элементов.
int[] myArrayOfInts = {10, 20, 30, 40};

foreach(int i in myArrayOfInts)
{
  Console.WriteLine(i);
}
```

Хотя может показаться, что данная конструкция подходит только для массивов, на самом деле foreach можно использовать с любым типом, который поддерживает метод GetEnumerator(). В целях иллюстрации создадим новый проект консольного приложения по имени CustomEnumerator и добавим в него файлы Car.cs и Radio.cs, которые были определены в примере SimpleException из главы 7 (через пункт меню Project⇨Add Existing Item).

На заметку! Во избежание импортирования в новый проект пространства имен CustomException имеет смысл переименовать пространство имен, содержащее типы Car и Radio, в CustomEnumerator.

Теперь вставим в проект новый класс Garage (гараж), который хранит набор объектов Car (автомобиль) внутри System.Array:

```
// Garage содержит набор объектов Car.
public class Garage
{
  private Car[] carArray = new Car[4];
  // При запуске заполнить несколькими объектами Car.
  public Garage()
  {
    carArray[0] = new Car("Rusty", 30);
    carArray[1] = new Car("Clunker", 55);
    carArray[2] = new Car("Zippy", 30);
    carArray[3] = new Car("Fred", 30);
  }
}
```

В идеальном случае было бы удобно проходить по внутренним элементам объекта Garage с применением конструкции foreach, как если бы это был массив значений данных:

```
// Код выглядит корректным...
public class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with IEnumerable / IEnumerator *****\n");
    Garage carLot = new Garage();
    // Проход по всем объектам Car в коллекции?
    foreach (Car c in carLot)
    {
      Console.WriteLine("{0} is going {1} MPH",
        c.PetName, c.CurrentSpeed);
    }
    Console.ReadLine();
  }
}
```

К сожалению, компилятор информирует вас о том, что в классе Garage не реализован метод по имени GetEnumerator(). Этот метод формально определен в интерфейсе IEnumerable, который находится в пространстве имен System.Collections.

На заметку! В главе 9 вы узнаете о роли обобщений и о пространстве имен System.Collections.Generic. Как будет показано, это пространство имен содержит обобщенные версии интерфейсов IEnumerable/IEnumerator, которые предоставляют более безопасный к типам способ итерации по элементам.

Классы или структуры, которые поддерживают такое поведение, позиционируются, как способные предоставлять доступ к содержащимся внутри них элементам вызывающему коду (в рассматриваемом примере самому ключевому слову `foreach`). Вот определение этого стандартного интерфейса .NET:

```
// Этот интерфейс информирует вызывающий код о том,
// что элементы объекта могут перечисляться.
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Как видите, метод `GetEnumerator()` возвращает ссылку на еще один интерфейс по имени `System.Collections.IEnumerator`. Данный интерфейс предлагает инфраструктуру, позволяющую вызывающему коду обходить внутренние объекты, которые содержатся в совместимом с `IEnumerable` контейнере:

```
// Этот интерфейс позволяет вызывающему коду получать элементы контейнера.
public interface IEnumerator
{
  bool MoveNext ();    // Переместить вперед внутреннюю позицию курсора.
  object Current { get;} // Получить текущий элемент (свойство только для чтения).
  void Reset ();       // Сбросить курсор в позицию перед первым элементом.
}
```

Если вы хотите обновить тип `Garage` для поддержки этих интерфейсов, то можете пойти длинным путем и реализовать каждый метод вручную. Хотя вы определенно вольны предоставить специализированные версии методов `GetEnumerator()`, `MoveNext()`, `Current` и `Reset()`, существует более легкий путь. Поскольку тип `System.Array` (а также многие другие классы коллекций) уже реализует интерфейсы `IEnumerable` и `IEnumerator`, вы можете просто делегировать запрос к `System.Array` следующим образом (обратите внимание, что понадобится импортировать пространство имен `System.Collections` в файл кода):

```
using System.Collections;
...
public class Garage : IEnumerable
{
  // System.Array уже реализует IEnumerator!
  private Car[] carArray = new Car[4];

  public Garage()
  {
    carArray[0] = new Car("FeeFee", 200);
    carArray[1] = new Car("Clunker", 90);
    carArray[2] = new Car("Zippy", 30);
    carArray[3] = new Car("Fred", 30);
  }

  public IEnumerator GetEnumerator()
  {
    // Возвратить IEnumerator объекта массива.
    return carArray.GetEnumerator();
  }
}
```

После такого изменения типа `Garage` можно безопасно использовать внутри конструкции `foreach`. Более того, учитывая, что метод `GetEnumerator()` был определен как открытый, пользователь объекта может также взаимодействовать с типом `IEnumerator`:

```
// Вручную работать с IEnumerator.
IEnumerator i = carLot.GetEnumerator();
i.MoveNext();
Car myCar = (Car)i.Current;
Console.WriteLine("{0} is going {1} MPH", myCar.PetName, myCar.
CurrentSpeed);
```

Тем не менее, если вы предпочитаете скрыть функциональность `IEnumerable` на уровне объектов, то просто примените явную реализацию интерфейса:

```
IEnumerator IEnumerable.GetEnumerator()
{
  // Возвратить IEnumerator объекта массива.
  return carArray.GetEnumerator();
}
```

В результате обычный пользователь объекта не обнаружит метод `GetEnumerator()` в классе `Garage`, в то время как конструкция `foreach` при необходимости будет получать интерфейс в фоновом режиме.

Исходный код. Проект `CustomEnumerator` доступен в подкаталоге `Chapter_8`.

Построение методов итератора с использованием ключевого слова `yield`

Существует альтернативный способ построения типов, которые работают с циклом `foreach`, с использованием *итераторов*. Попросту говоря, *итератор* — это член, который указывает, каким образом должны возвращаться внутренние элементы контейнера во время обработки в цикле `foreach`. В целях иллюстрации создадим новый проект консольного приложения по имени `CustomEnumeratorWithYield` и вставим в него типы `Car`, `Radio` и `Garage` из предыдущего примера (снова при желании переименовав пространство имен для текущего проекта). Затем модифицируем тип `Garage`:

```
public class Garage : IEnumerable
{
  private Car[] carArray = new Car[4];
  ...
  // Метод итератора.
  public IEnumerator GetEnumerator()
  {
    foreach (Car c in carArray)
    {
      yield return c;
    }
  }
}
```

Обратите внимание, что эта реализация метода `GetEnumerator()` осуществляет проход по элементам с применением внутренней логики `foreach` и возвращает каждый объект `Car` вызывающему коду, используя синтаксис `yield return`. Ключевое слово `yield` применяется для указания значения или значений, которые подлежат возвращению конструкцией `foreach` вызывающему коду. При достижении оператора `yield return` текущее местоположение в контейнере сохраняется и выполнение возобновляется с этого местоположения, когда итератор вызывается в следующий раз.

Методы итераторов не обязаны использовать ключевое слово `foreach` для возвращения своего содержимого. Метод итератора допускается также определять так, как показано ниже:

```
public IEnumerator GetEnumerator()
{
  yield return carArray[0];
  yield return carArray[1];
  yield return carArray[2];
  yield return carArray[3];
}
```

В этой реализации обратите внимание на то, что метод `GetEnumerator()` явно возвращает вызывающему коду новое значение при каждом своем проходе. Поступать так в рассматриваемом примере мало смысла, потому что если вы добавите дополнительные объекты к переменной-члену `carArray`, то метод `GetEnumerator()` станет рассогласованным. Тем не менее, такой синтаксис может быть полезен, когда вы хотите возвращать из метода локальные данные для обработки посредством `foreach`.

Построение именованного итератора

Также интересно отметить, что ключевое слово `yield` формально может применяться внутри любого метода независимо от его имени. Такие методы (которые официально называются *именованными итераторами*) уникальны тем, что могут принимать любое количество аргументов. При построении именованного итератора имейте в виду, что метод будет возвращать интерфейс `IEnumerable`, а не ожидаемый совместимый с `IEnumerator` тип. В целях иллюстрации добавим к типу `Garage` следующий метод:

```
public IEnumerable GetTheCars(bool ReturnRevesed)
{
  // Возвратить элементы в обратном порядке.
  if (ReturnRevesed)
  {
    for (int i = carArray.Length; i != 0; i--)
    {
      yield return carArray[i-1];
    }
  }
  else
  {
    // Возвратить элементы в том порядке, в каком они размещены в массиве.
    foreach (Car c in carArray)
    {
     yield return c;
    }
  }
}
```

Обратите внимание, что новый метод позволяет вызывающему коду получать элементы в прямом, а также в обратном порядке, если во входном параметре указано значение `true`. Вот как взаимодействовать с этим методом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with the Yield Keyword *****\n");
  Garage carLot = new Garage();
```

```
  // Получить элементы, используя GetEnumerator().
  foreach (Car c in carLot)
  {
    Console.WriteLine("{0} is going {1} MPH",
      c.PetName, c.CurrentSpeed);
  }
  Console.WriteLine();
  // Получить элементы (в обратном порядке!), используя именованный итератор.
  foreach (Car c in carLot.GetTheCars(true))
  {
    Console.WriteLine("{0} is going {1} MPH",
      c.PetName, c.CurrentSpeed);
  }
  Console.ReadLine();
}
```

Наверняка вы согласитесь с тем, что именованные итераторы являются удобными конструкциями, поскольку они позволяют определять в единственном специальном контейнере несколько способов для запрашивания возвращаемого набора.

Итак, в завершение темы построения перечислимых объектов запомните: для того, чтобы специальные типы могли работать с ключевым словом `foreach` языка C#, контейнер должен определять метод по имени `GetEnumerator()`, который формально определен интерфейсным типом `IEnumerable`. Реализация этого метода обычно осуществляется просто путем делегирования работы внутреннему члену, который хранит подобъекты; однако допускается также использовать синтаксис `yield return`, чтобы предоставить множество методов "именованных итераторов".

Исходный код. Проект `CustomEnumeratorWithYield` доступен в подкаталоге `Chapter_8`.

Интерфейс ICloneable

Вспомните из главы 6, что в классе `System.Object` определен метод по имени `MemberwiseClone()`. Этот метод применяется для получения *поверхностной (неглубокой) копии* текущего объекта. Пользователи объекта не могут вызывать указанный метод напрямую, т.к. он является защищенным. Тем не менее, отдельный объект может самостоятельно вызывать `MemberwiseClone()` во время процесса *клонирования*. Для примера создадим новый проект консольного приложения по имени `CloneablePoint`, в котором определен класс `Point`:

```
// Класс по имени Point.
public class Point
{
  public int X {get; set;}
  public int Y {get; set;}
  public Point(int xPos, int yPos) { X = xPos; Y = yPos;}
  public Point(){}
  // Переопределить Object.ToString().
  public override string ToString()
  { return string.Format(«X = {0}; Y = {1}», X, Y ); }
}
```

Учитывая имеющиеся у вас знания о ссылочных типах и типах значений (см. главу 4), должно быть понятно, что если вы присвоите одну переменную ссылочного типа

другой такой переменной, то получите две ссылки, которые указывают на тот же самый объект в памяти. Таким образом, следующая операция присваивания в результате дает две ссылки на один и тот же объект `Point` в куче; модификация с использованием любой из ссылок оказывает воздействие на тот же самый объект в куче:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Cloning *****\n");
  // Две ссылки на один и тот же объект!
  Point p1 = new Point(50, 50);
  Point p2 = p1;
  p2.X = 0;
  Console.WriteLine(p1);
  Console.WriteLine(p2);
  Console.ReadLine();
}
```

Чтобы предоставить специальному типу возможность возвращения вызывающему коду идентичную копию самого себя, можно реализовать стандартный интерфейс `ICloneable`. Как было показано в начале главы, интерфейс `ICloneable` определяет единственный метод по имени `Clone()`:

```
public interface ICloneable
{
  object Clone();
}
```

Очевидно, что реализация метода `Clone()` варьируется от класса к классу. Однако базовая функциональность в основном остается неизменной: копирование значений переменных-членов в новый объект того же самого типа и возвращение его пользователю. Чтобы продемонстрировать сказанное, модифицируем класс `Point`:

```
// Теперь Point поддерживает способность клонирования.
public class Point : ICloneable
{
  public int X { get; set; }
  public int Y { get; set; }

  public Point(int xPos, int yPos) { X = xPos; Y = yPos; }
  public Point() { }

  // Переопределить Object.ToString().
  public override string ToString()
  { return string.Format(«X = {0}; Y = {1}», X, Y); }

  // Возвратить копию текущего объекта.
  public object Clone()
  { return new Point(this.X, this.Y); }
}
```

Теперь можно создавать точные автономные копии типа `Point`, как показано далее:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Cloning *****\n");
  // Обратите внимание, что Clone() возвращает простой тип object.
  // Для получения производного типа требуется явное приведение.
  Point p3 = new Point(100, 100);
  Point p4 = (Point)p3.Clone();
```

```
  // Изменить p4.X (что не приводит к изменению p3.x).
  p4.X = 0;

  // Вывести все объекты.
  Console.WriteLine(p3);
  Console.WriteLine(p4);
  Console.ReadLine();
}
```

Хотя текущая реализация типа `Point` удовлетворяет всем требованиям, ее можно немного улучшить. Поскольку `Point` не содержит никаких внутренних переменных ссылочного типа, реализацию метода `Clone()` можно упростить:

```
public object Clone()
{
  // Копировать все поля Point по очереди.
  return this.MemberwiseClone();
}
```

Тем не менее, учтите, что если бы в типе `Point` содержались любые переменные-члены ссылочного типа, то метод `MemberwiseClone()` копировал бы ссылки на эти объекты (т.е. создавал бы *поверхностную копию*). Для поддержки настоящей *глубокой (детальной) копии* во время процесса клонирования понадобится создавать новые экземпляры каждой переменной-члена ссылочного типа. Давайте рассмотрим пример.

Более сложный пример клонирования

Теперь предположим, что класс `Point` содержит переменную-член ссылочного типа `PointDescription`. Данный класс представляет дружественное имя точки, а также ее идентификационный номер, выраженный как `System.Guid` (глобально уникальный идентификатор (globally unique identifier — GUID) — статистически уникальное 128-битное число). Вот как выглядит реализация:

```
// Этот класс описывает точку.
public class PointDescription
{
  public string PetName {get; set;}
  public Guid PointID {get; set;}

  public PointDescription()
  {
    PetName = "No-name";
    PointID = Guid.NewGuid();
  }
}
```

Начальные изменения самого класса `Point` включают модификацию метода `ToString()` для учета новых данных состояния, а также определение и создание ссылочного типа `PointDescription`. Чтобы позволить внешнему миру устанавливать дружественное имя для `Point`, необходимо также изменить аргументы, передаваемые перегруженному конструктору:

```
public class Point : ICloneable
{
  public int X { get; set; }
  public int Y { get; set; }
  public PointDescription desc = new PointDescription();
```

```
  public Point(int xPos, int yPos, string petName)
  {
    X = xPos; Y = yPos;
    desc.PetName = petName;
  }
  public Point(int xPos, int yPos)
  {
    X = xPos; Y = yPos;
  }
  public Point() { }

  // Переопределить Object.ToString().
  public override string ToString()
  {
    return string.Format(«X = {0}; Y = {1}; Name = {2};\nID = {3}\n»,
    X, Y, desc.PetName, desc.PointID);
  }

  // Возвратить копию текущего объекта.
  public object Clone()
  { return this.MemberwiseClone(); }
}
```

Обратите внимание, что метод `Clone()` **пока еще не обновлялся. Следовательно, когда пользователь объекта запросит клонирование с применением текущей реализации, будет создана поверхностная (почленная) копия. В целях иллюстрации модифицируем метод** `Main()`**, как показано ниже:**

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Cloning *****\n");
  Console.WriteLine("Cloned p3 and stored new Point in p4");
  Point p3 = new Point(100, 100, "Jane");
  Point p4 = (Point)p3.Clone();

  Console.WriteLine("Before modification:");
  Console.WriteLine("p3: {0}", p3);
  Console.WriteLine("p4: {0}", p4);
  p4.desc.PetName = "My new Point";
  p4.X = 9;

  Console.WriteLine("\nChanged p4.desc.petName and p4.X");
  Console.WriteLine("After modification:");
  Console.WriteLine("p3: {0}", p3);
  Console.WriteLine("p4: {0}", p4);
  Console.ReadLine();
}
```

В приведенном выводе видно, что хотя типы значений действительно были изменены, внутренние ссылочные типы поддерживают одни и те же значения, т.к. они "указывают" на те же самые объекты в памяти (скажем, теперь оба объекта имеют дружественное имя `My new Point`**):**

```
***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

p4: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509
```

```
Changed p4.desc.petName  and p4.X
After modification:
p3: X = 100; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

p4: X = 9; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509
```

Чтобы заставить метод `Clone()` создавать полную глубокую копию внутренних ссылочных типов, нужно сконфигурировать объект, возвращаемый методом `MemberwiseClone()`, для учета имени текущего объекта `Point` (тип `System.Guid` на самом деле является структурой, так что числовые данные будут действительно копироваться). Вот одна из возможных реализаций:

```
// Теперь необходимо скорректировать код для учета члена PointDescription.
public object Clone()
{
  // Сначала получить поверхностную копию.
  Point newPoint = (Point)this.MemberwiseClone();

  // Затем восполнить пробелы.
  PointDescription currentDesc = new PointDescription();
  currentDesc.PetName = this.desc.PetName;
  newPoint.desc = currentDesc;
  return newPoint;
}
```

Если снова запустить приложение и просмотреть его вывод (показанный далее), то будет видно, что возвращаемый методом `Clone()` объект `Point` действительно копирует свои внутренние переменные-члены ссылочного типа (обратите внимание, что дружественные имена у `p3` и `p4` теперь уникальны):

```
***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 100; Y = 100; Name = Jane;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a

Changed p4.desc.petName  and p4.X
After modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 9; Y = 100; Name = My new Point;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a
```

Давайте подведем итоги по процессу клонирования. При наличии класса или структуры, которая содержит только типы значений, необходимо реализовать метод `Clone()` с использованием метода `MemberwiseClone()`. Однако если есть специальный тип, поддерживающий ссылочные типы, то для построения глубокой копии может потребоваться создание нового объекта, который учитывает каждую переменную-член ссылочного типа.

Исходный код. Проект `CloneablePoint` доступен в подкаталоге `Chapter_8`.

Интерфейс IComparable

Интерфейс System.IComparable описывает поведение, которое позволяет сортировать объекты на основе указанного ключа. Вот его формальное определение:

```
// Этот интерфейс позволяет объекту указывать
// его отношение с другими подобными объектами.
public interface IComparable
{
  int CompareTo(object o);
}
```

На заметку! Обобщенная версия этого интерфейса (IComparable<T>) предлагает более безопасный в отношении типов способ обработки операций сравнения объектов. Обобщения исследуются в главе 9.

Давайте создадим новый проект консольного приложения по имени ComparableCar и обновим класс Car из главы 7 (обратите внимание, что мы просто добавили новое свойство для представления уникального идентификатора каждого автомобиля и модифицированный конструктор):

```
public class Car
{
  ...
  public int CarID {get; set;}
  public Car(string name, int currSp, int id)
  {
    CurrentSpeed = currSp;
    PetName = name;
    CarID = id;
  }
  ...
}
```

Теперь предположим, что имеется следующий массив объектов Car:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Sorting *****\n");

  // Создать массив объектов Car.
  Car[] myAutos = new Car[5];
  myAutos[0] = new Car("Rusty", 80, 1);
  myAutos[1] = new Car("Mary", 40, 234);
  myAutos[2] = new Car("Viper", 40, 34);
  myAutos[3] = new Car("Mel", 40, 4);
  myAutos[4] = new Car("Chucky", 40, 5);

  Console.ReadLine();
}
```

В классе System.Array определен статический метод по имени Sort(). Его вызов для массива внутренних типов (int, short, string и т.д.) приводит к сортировке элементов массива в числовом или алфавитном порядке, т.к. эти внутренние типы данных реализуют интерфейс IComparable. Но что произойдет, если передать методу Sort() массив объектов Car?

```
// Сортируются ли объекты Car? Пока еще нет!
Array.Sort(myAutos);
```

Запустив тестовый код, вы получите исключение времени выполнения, потому что класс Car не поддерживает необходимый интерфейс. При построении специальных типов вы можете реализовать интерфейс IComparable, чтобы позволить массивам, содержащим элементы этих типов, подвергаться сортировке. Когда вы реализуете детали CompareTo(), то должны самостоятельно принять решение о том, что должно браться за основу в операции упорядочивания. Для типа Car вполне логичным кандидатом может послужить внутреннее свойство CarID:

```
// Итерация по объектам Car может быть упорядочена на основе CarID.
public class Car : IComparable
{
  ...
  // Реализация интерфейса IComparable.
  int IComparable.CompareTo(object obj)
  {
    Car temp = obj as Car;
    if (temp != null)
    {
      if (this.CarID > temp.CarID)
        return 1;
      if (this.CarID < temp.CarID)
        return -1;
      else
        return 0;
    }
    else
      throw new ArgumentException("Parameter is not a Car!");
                                  // Параметр не является объектом типа Car!
  }
}
```

Как видите, логика метода CompareTo() заключается в сравнении входного объекта с текущим экземпляром на основе специфичного элемента данных. Возвращаемое значение метода CompareTo() применяется для выяснения того, является текущий объект меньше, больше или равным объекту, с которым он сравнивается (табл. 8.1).

Таблица 8.1. Возвращаемые значения метода CompareTo()

Возвращаемое значение	Описание
Любое число меньше нуля	Этот экземпляр находится перед указанным объектом в порядке сортировки
Ноль	Этот экземпляр равен указанному объекту
Любое число больше нуля	Этот экземпляр находится после указанного объекта в порядке сортировки

Предыдущую реализацию метода CompareTo() можно усовершенствовать с учетом того факта, что тип данных int в C# (который представляет собой просто сокращенное обозначение для типа System.Int32 в CLR) реализует интерфейс IComparable. Реализовать CompareTo() в Car можно было бы так:

```
int IComparable.CompareTo(object obj)
{
  Car temp = obj as Car;
  if (temp != null)
    return this.CarID.CompareTo(temp.CarID);
```

```
    else
      throw new ArgumentException("Parameter is not a Car!");
                                  // Параметр не является объектом типа Car!
  }
```

В том или другом случае, поскольку тип `Car` понимает, как сравнивать себя с подобными объектами, вы можете написать следующий тестовый код:

```
// Использование интерфейса IComparable.
static void Main(string[] args)
{
  // Создать массив объектов Car.
  ...
  // Отобразить текущее содержимое массива.
  Console.WriteLine("Here is the unordered set of cars:");
  foreach(Car c in myAutos)
    Console.WriteLine("{0} {1}", c.CarID, c.PetName);

  // Теперь отсортировать массив, используя IComparable!
  Array.Sort(myAutos);
  Console.WriteLine();

  // Отобразить отсортированное содержимое массива.
  Console.WriteLine("Here is the ordered set of cars:");
  foreach(Car c in myAutos)
    Console.WriteLine("{0} {1}", c.CarID, c.PetName);
  Console.ReadLine();
}
```

Ниже показан вывод, полученный в результате выполнения приведенного выше метода `Main()`:

```
***** Fun with Object Sorting *****

Here is the unordered set of cars:
1 Rusty
234 Mary
34 Viper
4 Mel
5 Chucky

Here is the ordered set of cars:
1 Rusty
4 Mel
5 Chucky
34 Viper
234 Mary
```

Указание множества порядков сортировки с помощью `IComparer`

В текущей версии класса `Car` в качестве основы для порядка сортировки используется идентификатор автомобиля (`CarID`). В другом проектном решении основой сортировки могло быть дружественное имя автомобиля (для вывода списка автомобилей в алфавитном порядке). А что если вы хотите построить класс `Car`, который можно было бы подвергать сортировке по идентификатору *и также* по дружественному имени? В таком случае вы должны ознакомиться с еще одним стандартным интерфейсом по имени `IComparer`, который определен в пространстве имен `System.Collections` следующим образом:

```
// Общий способ для сравнения двух объектов.
interface IComparer
{
  int Compare(object o1, object o2);
}
```

На заметку! Обобщенная версия этого интерфейса (`IComparable<T>`) предоставляет более безопасный в отношении типов способ обработки операций сравнения объектов. Обобщения подробно рассматриваются в главе 9.

В отличие от `IComparable` интерфейс `IComparer` обычно *не* реализуется в типе, который вы пытаетесь сортировать (т.е. `Car`). Взамен данный интерфейс реализуется в любом количестве вспомогательных классов, по одному для каждого порядка сортировки (на основе дружественного имени, идентификатора автомобиля и т.д.). В настоящий момент типу `Car` уже известно, как сравнивать автомобили друг с другом по внутреннему идентификатору. Следовательно, чтобы позволить пользователю объекта сортировать массив объектов `Car` по дружественному имени, потребуется создать дополнительный вспомогательный класс, реализующий интерфейс `IComparer`. Вот необходимый код (не забудьте импортировать в файл кода пространство имен `System.Collections`):

```
// Этот вспомогательный класс используется для сортировки
// массива объектов Car по дружественному имени.
public class PetNameComparer : IComparer
{
  // Проверить дружественное имя каждого объекта.
  int IComparer.Compare(object o1, object o2)
  {
    Car t1 = o1 as Car;
    Car t2 = o2 as Car;
    if(t1 != null && t2 != null)
      return String.Compare(t1.PetName, t2.PetName);
    else
      throw new ArgumentException("Parameter is not a Car!");
                                  // Параметр не является объектом типа Car!
  }
}
```

Теперь вспомогательный класс `PetNameComparer` можно задействовать в коде. Класс `System.Array` содержит несколько перегруженных версий метода `Sort()`, одна из которых принимает объект, реализующий интерфейс `IComparer`:

```
static void Main(string[] args)
{
  ...
  // Теперь сортировать по дружественному имени.
  Array.Sort(myAutos, new PetNameComparer());

  // Вывести отсортированный массив.
  Console.WriteLine("Ordering by pet name:");
  foreach(Car c in myAutos)
    Console.WriteLine("{0} {1}", c.CarID, c.PetName);
  ...
}
```

Специальные свойства и специальные типы сортировки

Важно отметить, что вы можете применять специальное статическое свойство, оказывая пользователю объекта помощь с сортировкой типов Car по специфичному элементу данных. Предположим, что в класс Car добавлено статическое свойство только для чтения по имени SortByPetName, которое возвращает экземпляр класса, реализующего интерфейс IComparer (в этом случае PetNameComparer; не забудьте импортировать пространство имен System.Collections):

```
// Теперь мы поддерживаем специальное свойство для возвращения
// корректного экземпляра, реализующего интерфейс IComparer.
public class Car : IComparable
{
  ...
  // Свойство, возвращающее PetNameComparer.
  public static IComparer SortByPetName
  { get { return (IComparer)new PetNameComparer(); } }
}
```

Теперь в коде массив можно сортировать по дружественному имени, используя жестко ассоциированное свойство, а не автономный класс PetNameComparer:

```
// Сортировка по дружественному имени становится немного яснее.
Array.Sort(myAutos, Car.SortByPetName);
```

Исходный код. Проект ComparableCar доступен в подкаталоге Chapter_8.

К этому моменту вы должны не только понимать способы определения и реализации собственных интерфейсов, но и оценить их полезность. Конечно, интерфейсы встречаются внутри каждого важного пространства имен .NET, и в оставшихся главах книги вы продолжите работать с разнообразными стандартными интерфейсами.

Резюме

Интерфейс может быть определен как именованная коллекция *абстрактных членов*. Поскольку интерфейс не предоставляет никаких деталей реализации, общепринято расценивать его как поведение, которое может поддерживаться заданным типом. Когда два или больше число типов реализуют один и тот же интерфейс, каждый из них может трактоваться одинаковым образом (полиморфизм на основе интерфейсов), даже если эти типы определены в разных иерархиях.

Для определения новых интерфейсов в языке C# предусмотрено ключевое слово interface. Как было показано в главе, тип может поддерживать столько интерфейсов, сколько необходимо, которые указываются в виде списка с разделителями-запятыми. Более того, разрешено создавать интерфейсы, которые являются производными от множества базовых интерфейсов.

В дополнение к построению специальных интерфейсов библиотеки .NET определяют набор стандартных (т.е. поставляемых вместе с платформой) интерфейсов. Вы видели, что можно создавать специальные типы, которые реализуют эти предопределенные интерфейсы с целью поддержки ряда желательных возможностей, таких как клонирование, сортировка и перечисление.

ЧАСТЬ IV

Дополнительные конструкции программирования на C#

В этой части

ГЛАВА 9

Коллекции и обобщения

Любому приложению, создаваемому с помощью платформы .NET, потребуется решать проблему поддержки и манипулирования набором значений данных в памяти. Эти значения данных могут поступать из множества местоположений, включая реляционную базу данных, локальный текстовый файл, XML-документ, вызов веб-службы или через предоставляемый пользователем источник ввода.

В первом выпуске платформы .NET программисты часто применяли классы из пространства имен `System.Collections` для хранения и взаимодействия с элементами данных, используемыми внутри приложения. В версии .NET 2.0 язык программирования C# был расширен поддержкой средства, которое называется *обобщениями*, и вместе с этим изменением в библиотеках базовых классов появилось совершенно новое пространство имен — `System.Collections.Generic`.

В настоящей главе представлен обзор разнообразных пространств имен и типов коллекций (обобщенных и необобщенных), находящихся в библиотеках базовых классов .NET. Вы увидите, что обобщенные контейнеры часто превосходят свои необобщенные аналоги, поскольку они обычно обеспечивают лучшую безопасность в отношении типов и дают выигрыш в плане производительности. После того, как вы научитесь создавать и манипулировать обобщенными элементами внутри платформы, в оставшемся материале главы будет продемонстрировано создание собственных обобщенных методов и типов. Вы узнаете о роли *ограничений* (и соответствующего ключевого слова `where` языка C#), которые позволяют строить классы, исключительно безопасные к типам.

Побудительные причины создания классов коллекций

Самым элементарным контейнером, который допускается применять для хранения данных приложения, несомненно, можно считать массив. В главе 4 вы узнали, что массив C# позволяет определить набор идентично типизированных элементов (включая массив элементов типа `System.Object`, по существу представляющий собой массив данных любых типов) с фиксированным верхним пределом. Кроме того, вспомните из главы 4, что все переменные массивов C# накапливают в себе много функциональных возможностей из класса `System.Array`. В качестве краткого напоминания взгляните на следующий метод `Main()`, который создает массив текстовых данных и манипулирует его содержимым разными способами:

```
static void Main(string[] args)
{
  // Создать массив строковых данных.
```

```
  string[] strArray = {"First", "Second", "Third" };

  // Отобразить количество элементов в массиве с помощью свойства Length.
  Console.WriteLine("This array has {0} items.", strArray.Length);
  Console.WriteLine();

  // Отобразить содержимое массива, используя перечислитель.
  foreach (string s in strArray)
  {
    Console.WriteLine("Array Entry: {0}", s);
  }
  Console.WriteLine();

  // Обратить массив и снова вывести его содержимое.
  Array.Reverse(strArray);
  foreach (string s in strArray)
  {
    Console.WriteLine("Array Entry: {0}", s);
  }

  Console.ReadLine();
}
```

Базовые массивы могут быть удобными для управления небольшими объемами данных фиксированного размера. Но есть немало случаев, когда требуются более гибкие структуры данных, такие как динамически расширяющийся и сокращающийся контейнер или контейнер, который может хранить только объекты, удовлетворяющие заданному критерию (например, объекты, производные от специфичного базового класса, или объекты, реализующие определенный интерфейс). Когда вы используете простой массив, всегда помните о том, что он имеет "фиксированный размер". Если вы создали массив из трех элементов, то вы и получите только три элемента; следовательно, представленный ниже код даст в результате исключение времени выполнения (конкретно — `IndexOutOfRangeException`):

```
static void Main(string[] args)
{
  // Создать массив строковых данных.
  string[] strArray = { "First", "Second", "Third" };

  // Попытка добавить новый элемент в конец массива? Ошибка во время выполнения!
  strArray[3] = "new item?";
  ...
}
```

На заметку! На самом деле изменять размер массива можно с применением обобщенного метода `Resize()<T>`. Однако такое действие приведет к копированию данных в новый объект массива и может оказаться неэффективным.

Чтобы помочь в преодолении ограничений простого массива, библиотеки базовых классов .NET поставляются с несколькими пространствами имен, которые содержат *классы коллекций*. В отличие от простого массива C# классы коллекций построены с возможностью динамического изменения своих размеров на лету по мере вставки либо удаления из них элементов. Более того, многие классы коллекций предлагают улучшенную безопасность в отношении типов и серьезно оптимизированы для обработки содержащихся внутри данных в манере, эффективной с точки зрения затрат памяти. В ходе чтения этой главы вы быстро заметите, что класс коллекции может принадлежать к одной из двух обширных категорий:

- необобщенные коллекции (в основном находящиеся в пространстве имен `System.Collections`);
- обобщенные коллекции (в основном находящиеся в пространстве имен `System.Collections.Generic`).

Необобщенные коллекции обычно спроектированы для оперирования над типами `System.Object` и таким образом являются слабо типизированными контейнерами (тем не менее, некоторые необобщенные коллекции работают только со специфическим типом данных наподобие объектов `string`). По контрасту обобщенные коллекции являются намного более безопасными к типам, учитывая, что при создании вы должны указывать "тип типа" данных, которые они будут содержать. Как вы увидите, признаком любого обобщенного элемента является наличие "параметра типа", обозначаемого с помощью угловых скобок (например, `List<T>`). Детали обобщений (в том числе связанные с ними преимущества) будут исследоваться позже в этой главе. А сейчас давайте ознакомимся с некоторыми ключевыми типами необобщенных коллекций из пространств имен `System.Collections` и `System.Collections.Specialized`.

Пространство имен `System.Collections`

С самого первого выпуска платформы .NET программисты часто использовали классы необобщенных коллекций из пространства имен `System.Collecitons`, которое содержит набор классов, предназначенных для управления и организации крупных объемов данных в памяти. В табл. 9.1 документированы распространенные классы коллекций, определенные в этом пространстве имен, а также основные интерфейсы, которые они реализуют.

Таблица 9.1. Полезные классы из пространства имен `System.Collections`

Класс `System.Collections`	Описание	Основные реализуемые интерфейсы
`ArrayList`	Представляет коллекцию с динамически изменяемым размером, выдающую объекты в последовательном порядке	`IList`, `ICollection`, `IEnumerable` и `ICloneable`
`BitArray`	Управляет компактным массивом битовых значений, которые представляются как булевские, где `true` обозначает установленный (1) бит, а `false` — неустановленный (0) бит	`ICollection`, `IEnumerable` и `ICloneable`
`Hashtable`	Представляет коллекцию пар "ключ-значение", организованных на основе хеш-кода ключа	`IDictionary`, `ICollection`, `IEnumerable` и `ICloneable`
`Queue`	Представляет стандартную очередь объектов, работающую по принципу FIFO ("первый вошел — первый вышел")	`ICollection`, `IEnumerable` и `ICloneable`
`SortedList`	Представляет коллекцию пар "ключ-значение", отсортированных по ключу и доступных по ключу и по индексу	`IDictionary`, `ICollection`, `IEnumerable` и `ICloneable`
`Stack`	Представляет стек LIFO ("последний вошел — первый вышел"), поддерживающий функциональность заталкивания и выталкивания, а также считывания	`ICollection`, `IEnumerable` и `ICloneable`

Интерфейсы, реализованные этими классами коллекций, позволяют проникнуть в суть их общей функциональности. В табл. 9.2 представлено описание общей природы этих основных интерфейсов, часть из которых кратко обсуждалась в главе 8.

Таблица 9.2. Основные интерфейсы, поддерживаемые классами из пространства имен `System.Collections`

Интерфейс `System.Collections`	Описание
`ICollection`	Определяет общие характеристики (например, размер, перечисление и безопасность к потокам) для всех необобщенных типов коллекций
`ICloneable`	Позволяет реализующему объекту возвращать вызывающему коду копию самого себя
`IDictionary`	Позволяет объекту необобщенной коллекции представлять свое содержимое в виде пар "ключ-значение"
`IEnumerable`	Возвращает объект, реализующий интерфейс `IEnumerator` (см. следующую строку в таблице)
`IEnumerator`	Делает возможной итерацию в стиле `foreach` по элементам коллекции
`IList`	Обеспечивает поведение добавления, удаления и индексирования элементов в последовательном списке объектов

Иллюстративный пример: работа с `ArrayList`

Возможно, вы уже имеете начальный опыт применения (или реализации) некоторых из указанных выше классических структур данных, таких как стеки, очереди или списки. Если это не так, то при рассмотрении обобщенных аналогов таких структур позже в главе будут предоставлены дополнительные сведения об отличиях между ними. А пока что взгляните на метод `Main()`, в котором используется объект `ArrayList`. Обратите внимание, что мы можем добавлять (и удалять) элементы на лету, а контейнер автоматически соответствующим образом изменяет свой размер:

```
// Для доступа к ArrayList потребуется импортировать
// пространство имен System.Collections.
static void Main(string[] args)
{
  ArrayList strArray = new ArrayList();
  strArray.AddRange(new string[] { "First", "Second", "Third" });

  // Отобразить количество элементов в ArrayList.
  Console.WriteLine("This collection has {0} items.", strArray.Count);
  Console.WriteLine();

  // Добавить новый элемент и отобразить текущее их количество.
  strArray.Add("Fourth!");
  Console.WriteLine("This collection has {0} items.", strArray.Count);

  // Отобразить содержимое.
  foreach (string s in strArray)
  {
    Console.WriteLine("Entry: {0}", s);
  }
  Console.WriteLine();
}
```

Как вы могли догадаться, помимо свойства Count и методов AddRange() и Add() класс ArrayList имеет много полезных членов, которые полностью описаны в документации .NET Framework. К слову, другие классы System.Collections (Stack, Queue и т.д.) также подробно документированы в справочной системе .NET.

Однако важно отметить, что в большинстве ваших проектов .NET классы коллекций из пространства имен System.Collections, скорее всего, применяться не будут! В наши дни намного чаще используются их обобщенные аналоги, находящиеся в пространстве имен System.Collections.Generic. С учетом этого остальные необобщенные классы из System.Collections здесь не обсуждаются (и примеры работы с ними не приводятся).

Обзор пространства имен System.Collections.Specialized

System.Collections — не единственное пространство имен .NET, которое содержит необобщенные классы коллекций. В пространстве имен System.Collections.Specialized определено несколько специализированных типов коллекций. В табл. 9.3 описаны наиболее полезные типы в этом конкретном пространстве имен, которые все являются необобщенными.

Таблица 9.3. Полезные классы из пространства имен System.Collections.Specialized

Класс System.Collections.Specialized	Описание
HybridDictionary	Этот класс реализует интерфейс IDictionary за счет применения ListDictionary, пока коллекция мала, и переключения на Hashtable, когда коллекция становится большой
ListDictionary	Этот класс удобен, когда необходимо управлять небольшим количеством элементов (10 или около того), которые могут изменяться со временем. Для управления своими данными класс использует односвязный список
StringCollection	Этот класс обеспечивает оптимальный способ для управления крупными коллекциями строковых данных
BitVector32	Этот класс предоставляет простую структуру, которая хранит булевские значения и небольшие целые числа в 32 битах памяти

Кроме указанных конкретных типов классов это пространство имен также содержит много дополнительных интерфейсов и абстрактных базовых классов, которые можно применять в качестве стартовых точек для создания специальных классов коллекций. Хотя в ряде ситуаций такие "специализированные" типы могут оказаться именно тем, что требуется в ваших проектах, здесь они рассматриваться не будут. Опять-таки, во многих случаях вы, вероятно, обнаружите, что пространство имен System.Collections.Generic предлагает классы с похожей функциональностью, но с добавочными преимуществами.

На заметку! В библиотеках базовых классов .NET доступны два дополнительных пространства имен, связанные с коллекциями (System.Collections.ObjectModel и System.Collections.Concurrent). Первое из них будет описано позже в главе, когда вы освоите тему обобщений. Пространство имен System.Collections.Concurrent предоставляет классы коллекций, хорошо подходящие для многопоточной среды (многопоточность обсуждается в главе 19).

Проблемы, присущие необобщенным коллекциям

Хотя на протяжении многих лет с использованием необобщенных классов коллекций (и интерфейсов) было построено немало успешных приложений .NET, опыт показал, что применение этих типов может привести к возникновению ряда проблем.

Первая проблема заключается в том, что использование классов коллекций `System.Collections` и `System.Collections.Specialized` приводит к созданию кода с низкой производительностью, особенно в случае манипулирования числовыми данными (например, типами значений). Как вы вскоре увидите, когда структуры хранятся в любом необобщенном классе коллекции, прототипированном для оперирования с `System.Object`, среда CLR должна выполнять некоторое количество операций перемещения в памяти, что может нанести ущерб скорости выполнения.

Вторая проблема связана с тем, что большинство необобщенных классов коллекций не являются безопасными в отношении типов, т.к. они были созданы для работы с `System.Object` и потому могут содержать в себе вообще все что угодно. Если разработчик .NET нуждался в создании безопасной к типам коллекции (скажем, контейнера, который может хранить объекты, реализующие только определенный интерфейс), то единственным реальным вариантом было создание нового класса коллекции вручную. Хотя задача не отличалась высокой трудоемкостью, она была несколько утомительной.

Прежде вы увидите, как применять обобщения в своих программах, полезно чуть глубже рассмотреть недостатки необобщенных классов коллекций; это поможет лучше понять проблемы, которые был призван решить механизм обобщений. Давайте создадим новый проект консольного приложения по имени `IssuesWithNongenericCollections`, после чего импортируем пространство имен `System.Collections` в файл кода C#:

```
using System.Collections;
```

Проблема производительности

Как уже было указано в главе 4, платформа .NET поддерживает две обширных категории представления данных: типы значений и ссылочные типы. Поскольку в .NET определены две основных категории типов, временами возникает необходимость представить переменную одной категории как переменную другой категории. Для этого в C# предлагается простой механизм, называемый *упаковкой* (boxing), который позволяет хранить данные типа значения внутри ссылочной переменной. Предположим, что в методе по имени `SimpleBoxUnboxOperation()` создана локальная переменная типа `int`. Если где-то в приложении понадобится представить этот тип значения как ссылочный тип, то значение придется *упаковать*:

```
static void SimpleBoxUnboxOperation()
{
  // Создать переменную ValueType (типа int).
  int myInt = 25;

  // Упаковать int в ссылку на object.
  object boxedInt = myInt;
}
```

Упаковку можно формально определить как процесс явного присваивания данных типа значения переменной `System.Object`. При упаковке значения среда CLR размещает в куче новый объект и копирует в него величину типа значения (в этом случае `25`). В качестве результата возвращается ссылка на вновь размещенный в куче объект.

Противоположная операция также разрешена и называется *распаковкой* (unboxing). Распаковка представляет собой процесс преобразования значения, хранящегося в объ-

ектной ссылке, обратно в соответствующий тип значения в стеке. Синтаксически операция распаковки выглядит как обычная операция приведения, но ее семантика несколько отличается. Среда CLR начинает с проверки того, что полученный тип данных эквивалентен упакованному типу, и если это так, то копирует значение в переменную, находящуюся в стеке. Например, следующие операции распаковки работают успешно при условии, что лежащим в основе типом `boxedInt` действительно является `int`:

```
static void SimpleBoxUnboxOperation()
{
  // Создать переменную ValueType (типа int).
  int myInt = 25;

  // Упаковать int в ссылку на object.
  object boxedInt = myInt;

  // Распаковать ссылку обратно в int.
  int unboxedInt = (int)boxedInt;
}
```

Когда компилятор C# встречает синтаксис упаковки/распаковки, он выпускает код CIL, который содержит коды операций `box/unbox`. Если вы просмотрите сборку с помощью утилиты `ildasm.exe`, то обнаружите в ней показанный ниже код CIL:

```
.method private hidebysig static void SimpleBoxUnboxOperation() cil managed
{
  // Code size 19 (0x13)
  .maxstack 1
  .locals init ([0] int32 myInt, [1] object boxedInt, [2] int32 unboxedInt)
  IL_0000: nop
  IL_0001: ldc.i4.s 25
  IL_0003: stloc.0
  IL_0004: ldloc.0
  IL_0005: box [mscorlib]System.Int32
  IL_000a: stloc.1
  IL_000b: ldloc.1
  IL_000c: unbox.any [mscorlib]System.Int32
  IL_0011: stloc.2
  IL_0012: ret
} // end of method Program::SimpleBoxUnboxOperation
```

Помните, что в отличие от обычного приведения распаковка *обязана* осуществляться только в подходящий тип данных. Попытка распаковать порцию данных в некорректный тип приводит к генерации исключения `InvalidCastException`. Для обеспечения высокой безопасности каждая операция распаковки должна быть помещена внутрь конструкции `try/catch`, но такое действие со всеми операциями распаковки в приложении может оказаться достаточно трудоемкой задачей. Ниже показан измененный код, который выдаст ошибку из-за того, что в нем предпринята попытка распаковки упакованного значения `int` в тип `long`:

```
static void SimpleBoxUnboxOperation()
{
  // Создать переменную ValueType (типа int).
  int myInt = 25;

  // Упаковать int в ссылку на object.
  object boxedInt = myInt;

  // Распаковать в неподходящий тип данных, чтобы
  // инициировать исключение времени выполнения.
```

```
  try
  {
    long unboxedInt = (long)boxedInt;
  }
  catch (InvalidCastException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

На первый взгляд упаковка/распаковка может показаться довольно непримечательным средством языка, с которым связан больше академический интерес, чем практическая ценность. В конце концов, необходимость хранения локального типа значения в локальной переменной `object` будет возникать нечасто. Тем не менее, оказывается, что процесс упаковки/распаковки очень полезен, поскольку позволяет предполагать, что все можно трактовать как `System.Object`, в то время как среда CLR самостоятельно позаботится о деталях, качающихся памяти.

Давайте обратимся к практическому использованию описанных приемов. Предположим, что создан необобщенный класс `System.Collections.ArrayList` для хранения множества числовых (расположенных в стеке) данных. Члены `ArrayList` прототипированы для работы с данными `System.Object`. Теперь рассмотрим методы `Add()`, `Insert()`, `Remove()`, а также индексатор класса:

```
public class ArrayList : object,
  IList, ICollection, IEnumerable, ICloneable
{
  ...
  public virtual int Add(object value);
  public virtual void Insert(int index, object value);
  public virtual void Remove(object obj);
  public virtual object this[int index] {get; set; }
}
```

Класс `ArrayList` был построен для оперирования с экземплярами `object`, которые представляют данные, находящиеся в куче, поэтому может показаться странным, что следующий код компилируется и выполняется без ошибок:

```
static void WorkWithArrayList()
{
  // Типы значений автоматически упаковываются при передаче
  // методу, который требует экземпляр object.
  ArrayList myInts = new ArrayList();
  myInts.Add(10);
  myInts.Add(20);
  myInts.Add(35);
}
```

Хотя здесь числовые данные напрямую передаются методам, которые требуют экземпляров `object`, исполняющая среда выполняет автоматическую упаковку этих данных, основанных на стеке. Когда позже понадобится извлечь элемент из `ArrayList` с применением индексатора типа, находящийся в куче объект должен быть распакован в целочисленное значение, расположенное в стеке, посредством операции приведения. Не забывайте, что индексатор `ArrayList` возвращает элементы типа `System.Object`, а не `System.Int32`:

```
static void WorkWithArrayList()
{
  // Типы значений автоматически упаковываются,
  // когда передаются члену, принимающему object.
  ArrayList myInts = new ArrayList();
  myInts.Add(10);
  myInts.Add(20);
  myInts.Add(35);

  // Распаковка происходит, когда объект преобразуется
  // обратно в данные, расположенные в стеке.
  int i = (int)myInts[0];

  // Теперь значение вновь упаковывается, т.к.
  // метод WriteLine() требует типа object!
  Console.WriteLine("Value of your int: {0}", i);
}
```

Обратите внимание, что расположенное в стеке значение типа `System.Int32` перед вызовом метода `ArrayList.Add()` упаковывается, чтобы оно могло быть передано в требуемом виде `System.Object`. Также отметьте, что объект `System.Object` распаковывается обратно в `System.Int32` после его извлечения из `ArrayList` через операцию приведения лишь для того, чтобы *снова* быть упакованными при передаче методу `Console.WriteLine()`, поскольку этот метод работает с типом `System.Object`.

Упаковка и распаковка удобны с точки зрения программиста, но такой упрощенный подход к передаче данных между стеком и кучей влечет за собой проблемы, связанные с производительностью (снижение скорости выполнения и увеличение размера кода), и приводит к утере безопасности в отношении типов. Чтобы понять проблемы с производительностью, обдумайте действия, которые должны произойти при упаковке и распаковке простого целочисленного значения.

1. Новый объект должен быть размещен в управляемой куче.
2. Значение данных, находящееся в стеке, должно быть передано в выделенное место в памяти.
3. При распаковке значение, которое хранится в объекте, находящемся в куче, должно быть передано обратно в стек.
4. Неиспользуемый в дальнейшем объект, расположенный в куче, будет (со временем) удален сборщиком мусора.

Несмотря на то что показанный конкретный метод `Main()` не создает главное узкое место в плане производительности, вы определенно заметите это влияние, если `ArrayList` будет содержать тысячи целочисленных значений, которыми программа манипулирует на регулярной основе. В идеальном мире мы могли бы обрабатывать данные, находящиеся внутри контейнера в стеке, безо всяких проблем с производительностью. Было бы замечательно иметь возможность извлекать данные из контейнера, не прибегая к конструкциям `try/catch` (это именно то, что позволяют делать обобщения).

Проблема безопасности к типам

Мы уже затрагивали проблему безопасности к типам, когда рассматривали операции распаковки. Вспомните, что данные должны быть распакованы в тот же самый тип, с которым они были объявлены перед упаковкой. Однако существует еще один аспект безопасности к типам, который необходимо иметь в виду в мире без обобщений: тот факт, что классы из пространства имен `System.Collections` обычно могут хранить любые данные, т.к. их члены прототипированы для оперирования с типом `System.Object`.

Например, следующий метод строит список ArrayList с произвольными фрагментами несвязанных данных:

```
static void ArrayListOfRandomObjects()
{
  // ArrayList может хранить вообще все что угодно.
  ArrayList allMyObjects = new ArrayList();
  allMyObjects.Add(true);
  allMyObjects.Add(new OperatingSystem(PlatformID.MacOSX, new Version(10, 0)));
  allMyObjects.Add(66);
  allMyObjects.Add(3.14);
}
```

В некоторых случаях вам будет требоваться исключительно гибкий контейнер, который способен хранить буквально все (как было здесь показано). Но большую часть времени вас интересует *безопасный в отношении типов* контейнер, который может работать только с определенным типом данных. Например, вы можете нуждаться в контейнере, хранящем только объекты типа подключения к базе данных, растрового изображения или класса, реализующего интерфейс IPointy.

До появления обобщений единственный способ решения проблемы, касающейся безопасности к типам, предусматривал создание вручную специального класса (строго типизированной) коллекции. Предположим, что вы хотите создать специальную коллекцию, которая может содержать только объекты типа Person:

```
public class Person
{
  public int Age {get; set;}
  public string FirstName {get; set;}
  public string LastName {get; set;}
  public Person(){}
  public Person(string firstName, string lastName, int age)
  {
    Age = age;
    FirstName = firstName;
    LastName = lastName;
  }
  public override string ToString()
  {
    return string.Format("Name: {0} {1}, Age: {2}",
      FirstName, LastName, Age);
  }
}
```

Чтобы построить коллекцию, которая способна хранить только объекты Person, можно определить переменную-член System.Collection.ArrayList внутри класса по имени PeopleCollection и сконфигурировать все члены для оперирования со строго типизированными объектами Person, а не с объектами типа System.Object. Ниже приведен простой пример (специальная коллекция производственного уровня могла бы поддерживать множество дополнительных членов и расширять абстрактный базовый класс из пространства имен System.Collections или System.Collections.Specialized):

```
public class PersonCollection : IEnumerable
{
  private ArrayList arPeople = new ArrayList();
  // Приведение для вызывающего кода.
  public Person GetPerson(int pos)
  { return (Person)arPeople[pos]; }
```

```
  // Вставка только объектов Person.
  public void AddPerson(Person p)
  { arPeople.Add(p); }

  public void ClearPeople()
  { arPeople.Clear(); }

  public int Count
  { get { return arPeople.Count; } }

  // Поддержка перечисления с помощью foreach.
  IEnumerator IEnumerable.GetEnumerator()
  { return arPeople.GetEnumerator(); }
}
```

Обратите внимание, что класс `PeopleCollection` реализует интерфейс `IEnumerable`, который делает возможной итерацию в стиле `foreach` по всем элементам, содержащимся в коллекции. Кроме того, методы `GetPerson()` и `AddPerson()` прототипированы для работы только с объектами `Person`, а не растровыми изображениями, строками, подключениями к базе данных или другими элементами. Благодаря определению таких классов теперь обеспечивается безопасность к типам, учитывая, что компилятор C# будет способен выявить любую попытку вставки элемента несовместимого типа:

```
static void UsePersonCollection()
{
  Console.WriteLine("***** Custom Person Collection *****\n");
  PersonCollection myPeople = new PersonCollection();
  myPeople.AddPerson(new Person("Homer", "Simpson", 40));
  myPeople.AddPerson(new Person("Marge", "Simpson", 38));
  myPeople.AddPerson(new Person("Lisa", "Simpson", 9));
  myPeople.AddPerson(new Person("Bart", "Simpson", 7));
  myPeople.AddPerson(new Person("Maggie", "Simpson", 2));

  // Это вызовет ошибку на этапе компиляции!
  // myPeople.AddPerson(new Car());

  foreach (Person p in myPeople)
  Console.WriteLine(p);
}
```

Хотя специальные коллекции гарантируют безопасность к типам, такой подход обязывает создавать (в основном идентичные) специальные коллекции для всех уникальных типов данных, которые планируется в них помещать. Таким образом, если нужна специальная коллекция, которая могла бы оперировать только с классами, производными от базового класса `Car`, то придется построить очень похожий класс коллекции:

```
public class CarCollection : IEnumerable
{
  private ArrayList arCars = new ArrayList();

  // Приведение для вызывающего кода.
  public Car GetCar(int pos)
  { return (Car) arCars[pos]; }

  // Вставка только объектов Car.
  public void AddCar(Car c)
  { arCars.Add(c); }

  public void ClearCars()
  { arCars.Clear(); }

  public int Count
  { get { return arCars.Count; } }
```

```
  // Поддержка перечисления с помощью foreach.
  IEnumerator IEnumerable.GetEnumerator()
  { return arCars.GetEnumerator(); }
}
```

Тем не менее, класс специальной коллекции ничего не делает для решения проблемы с накладными расходами по упаковке/распаковке. Даже если создать специальную коллекцию по имени `IntCollection`, которая предназначена для работы только с элементами `System.Int32`, все равно придется выделять какой-то вид объекта под хранение данных (например, `System.Array` и `ArrayList`):

```
public class IntCollection : IEnumerable
{
  private ArrayList arInts = new ArrayList();

  // Получение int (выполняется распаковка).
  public int GetInt(int pos)
  { return (int)arInts[pos]; }

  // Вставка int (выполняется упаковка).
  public void AddInt(int i)
  { arInts.Add(i); }

  public void ClearInts()
  { arInts.Clear(); }

  public int Count
  { get { return arInts.Count; } }

  IEnumerator IEnumerable.GetEnumerator()
  { return arInts.GetEnumerator(); }
}
```

Независимо от того, какой тип выбран для хранения целых чисел, затруднений с упаковкой избежать невозможно, если применяются необобщенные контейнеры.

Первый взгляд на обобщенные коллекции

В случае использования классов обобщенных коллекций все описанные выше проблемы исчезают, включая накладные расходы на упаковку/распаковку и отсутствие безопасности в отношении типов. К тому же необходимость в создании специального класса (обобщенной) коллекции становится довольно редкой. Вместо построения уникальных классов, которые могут хранить объекты людей, автомобилей и целые числа, можно задействовать класс обобщенной коллекции и указать тип хранимых элементов.

Взгляните на следующий метод, в котором используется класс `List<T>` (из пространства имен `System.Collection.Generic`) для хранения разнообразных видов данных в строго типизированной манере (пока что не обращайте внимания на детали синтаксиса обобщений):

```
static void UseGenericList()
{
  Console.WriteLine("***** Fun with Generics *****\n");

  // Этот объект List<> может хранить только объекты Person.
  List<Person> morePeople = new List<Person>();
  morePeople.Add(new Person ("Frank", "Black", 50));
  Console.WriteLine(morePeople[0]);

  // Этот объект List<> может хранить только целые числа.
  List<int> moreInts = new List<int>();
  moreInts.Add(10);
```

```
    moreInts.Add(2);
    int sum = moreInts[0] + moreInts[1];
    // Ошибка на этапе компиляции! Объект Person
    // не может быть добавлен в список элементов int!
    // moreInts.Add(new Person());
}
```

Первый контейнер `List<T>` может содержать только объекты `Person`. По этой причине выполнять приведение при извлечении элементов из контейнера не требуется, что делает такой подход более безопасным в отношении типов. Второй контейнер `List<T>` может хранить только целые числа, которые размещены в стеке; другими словами, здесь не происходит никакой скрытой упаковки/распаковки, как это имеет место в необобщенном `ArrayList`. Ниже приведен краткий перечень преимуществ обобщенных контейнеров по сравнению с их необобщенными аналогами.

- Обобщения обеспечивают лучшую производительность, т.к. лишены накладных расходов по упаковке/распаковке, когда хранят типы значений.
- Обобщения безопасны к типам, потому что могут содержать только объекты указанного типа.
- Обобщения значительно сокращают потребность в специальных типах коллекций, поскольку при создании обобщенного контейнера указывается "тип типа".

Исходный код. Проект `IssuesWithNonGenericCollections` доступен в подкаталоге `Chapter_9`.

Роль параметров обобщенных типов

Обобщенные классы, интерфейсы, структуры и делегаты вы можете обнаружить повсюду в библиотеках базовых классов .NET, и они могут быть частью любого пространства имен .NET. Кроме того, имейте в виду, что применение обобщений далеко не ограничивается простым определением класса коллекции. Конечно, в оставшихся главах книги вы встретите случаи использования многих других обобщений для самых разных целей.

На заметку! Обобщенным образом могут быть записаны только классы, структуры, интерфейсы и делегаты, но не перечисления.

Глядя на обобщенный элемент в документации .NET Framework или браузере объектов Visual Studio, вы заметите пару угловых скобок с буквой или другой лексемой внутри. На рис. 9.1 показано окно браузера объектов Visual Studio, в котором отображается набор обобщенных элементов из пространства имен `System.Collections.Generic`, включая выделенный класс `List<T>`.

Формально эти лексемы называются *параметрами типа*, но в более дружественных к пользователю терминах на них можно ссылаться просто как на *заполнители*. Конструкцию `<T>` можно читать как "типа `T`". Таким образом, `IEnumerable<T>` можно прочитать как "`IEnumerable` типа `T`" или, говоря по-другому, как "перечисление типа `T`".

На заметку! Имя параметра типа (заполнитель) к делу не относится и зависит от предпочтений разработчика, создавшего обобщенный элемент. Однако обычно имя *`T`* применяется для представления типов, *`TKey`* или *`K`* — для представления ключей и *`TValue`* или *`V`* — для представления значений.

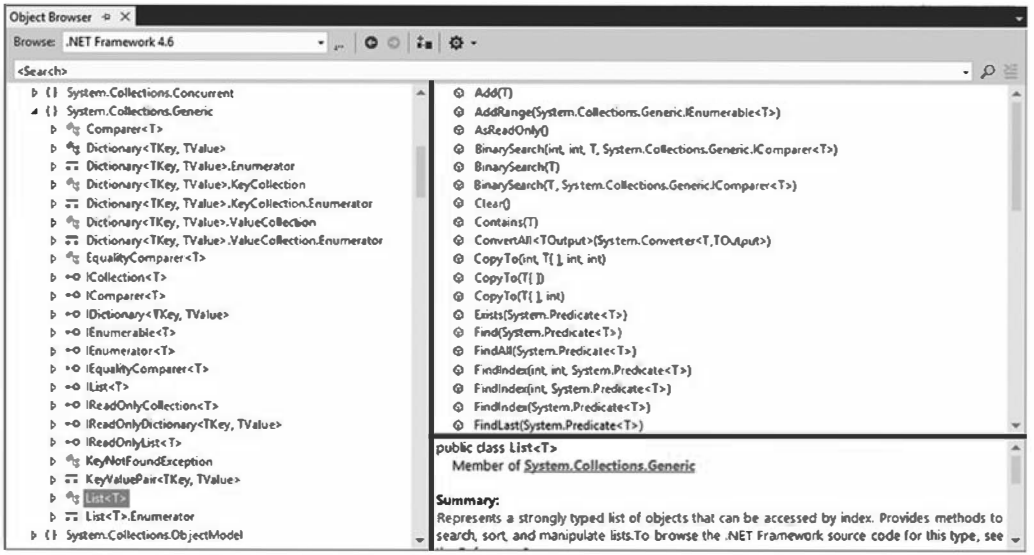


Рис. 9.1. Обобщенные элементы, поддерживающие параметры типа

Когда вы создаете обобщенный объект, реализуете обобщенный интерфейс или вызываете обобщенный член, на вас возлагается предоставление значения для параметра типа. Многочисленные примеры вы увидите как в этой главе, так и в остальных материалах книги. Тем не менее, для начала давайте рассмотрим основы взаимодействия с обобщенными типами и членами.

Указание параметров типа для обобщенных классов и структур

При создании экземпляра обобщенного класса или структуры вы указываете параметр типа, когда объявляете переменную и когда вызываете конструктор. Как было показано в предыдущем фрагменте кода, в методе `UseGenericList()` определены два объекта `List<T>`:

```
// Этот объект List<> может хранить только объекты Person.
List<Person> morePeople = new List<Person>();
```

Данный фрагмент можно трактовать как "`List<>` объектов `T`, где `T` — тип `Person`" или более просто как "список объектов действующих лиц". После указания параметра типа обобщенного элемента изменить его нельзя (помните, что сущностью обобщений является безопасность к типам). Когда параметр типа задается для обобщенного класса или структуры, все вхождения заполнителя (заполнителей) заменяются предоставленным значением. Если вы просмотрите полное объявление обобщенного класса `List<T>` в браузере объектов Visual Studio, то заметите, что заполнитель `T` используется в определении повсеместно. Ниже приведен частичный листинг (обратите внимание на элементы, выделенные полужирным):

```
// Частичное определение класса List<T>.
namespace System.Collections.Generic
{
  public class List<T> :
    IList<T>, ICollection<T>, IEnumerable<T>, IReadOnlyList<T>
    IList, ICollection, IEnumerable
  {
    ...
```

```
    public void Add(T item);
    public ReadOnlyCollection<T> AsReadOnly();
    public int BinarySearch(T item);
    public bool Contains(T item);
    public void CopyTo(T[] array);
    public int FindIndex(System.Predicate<T> match);
    public T FindLast(System.Predicate<T> match);
    public bool Remove(T item);
    public int RemoveAll(System.Predicate<T> match);
    public T[] ToArray();
    public bool TrueForAll(System.Predicate<T> match);
    public T this[int index] { get; set; }
  }
}
```

В случае создания `List<T>` с указанием объектов `Person` результат будет таким же, как если бы тип `List<T>` был определен следующим образом:

```
namespace System.Collections.Generic
{
  public class List<Person> :
    IList<Person>, ICollection<Person>, IEnumerable<Person>,
    IReadOnlyList<Person> IList, ICollection, IEnumerable
  {
    ...
    public void Add(Person item);
    public ReadOnlyCollection<Person> AsReadOnly();
    public int BinarySearch(Person item);
    public bool Contains(Person item);
    public void CopyTo(Person[] array);
    public int FindIndex(System.Predicate<Person> match);
    public Person FindLast(System.Predicate<Person> match);
    public bool Remove(Person item);
    public int RemoveAll(System.Predicate<Person> match);
    public Person[] ToArray();
    public bool TrueForAll(System.Predicate<Person> match);
    public Person this[int index] { get; set; }
  }
}
```

Разумеется, когда вы создаете в коде переменную обобщенного типа `List<T>`, компилятор вовсе не создает новую реализацию класса `List<T>`. Вместо этого он принимает во внимание только члены обобщенного типа, к которым вы действительно обращаетесь.

Указание параметров типа для обобщенных членов

Для необобщенного класса или структуры вполне нормально поддерживать несколько обобщенных членов (к примеру, методов и свойств). В таких случаях вы должны также указывать значение заполнителя во время вызова метода. Например, класс `System.Array` поддерживает набор обобщенных методов. В частности, необобщенный статический метод `Sort()` имеет обобщенный аналог по имени `Sort<T>()`. Рассмотрим представленный далее фрагмент кода, где `T` — это тип `int`:

```
int[] myInts = { 10, 4, 2, 33, 93 };
// Указание заполнителя для обобщенного метода Sort<>().
Array.Sort<int>(myInts);
foreach (int i in myInts)
{
  Console.WriteLine(i);
}
```

Указание параметров типов для обобщенных интерфейсов

Обобщенные интерфейсы обычно реализуются при построении классов или структур, которые нуждаются в поддержке разнообразных аспектов поведения платформы (скажем, клонирования, сортировки и перечисления). В главе 8 вы узнали о нескольких необобщенных интерфейсах, таких как `IComparable`, `IEnumerable`, `IEnumerator` и `IComparer`. Вспомните, что необобщенный интерфейс `IComparable` определен примерно так:

```
public interface IComparable
{
  int CompareTo(object obj);
}
```

Там же, в главе 8, этот интерфейс был реализован классом `Car`, чтобы сделать возможной сортировку стандартного массива. Однако код требовал нескольких проверок времени выполнения и операций приведения, потому что параметром был общий тип `System.Object`:

```
public class Car : IComparable
{
  ...
  // Реализация IComparable.
  int IComparable.CompareTo(object obj)
  {
    Car temp = obj as Car;
    if (temp != null)
    {
      if (this.CarID > temp.CarID)
        return 1;
      if (this.CarID < temp.CarID)
        return -1;
      else
        return 0;
    }
    else
      throw new ArgumentException("Parameter is not a Car!");
  }
}
```

Теперь представим, что применяется обобщенный аналог данного интерфейса:

```
public interface IComparable<T>
{
  int CompareTo(T obj);
}
```

В таком случае код реализации будет значительно яснее:

```
public class Car : IComparable<Car>
{
  ...
  // Реализация IComparable<T>.
  int IComparable<Car>.CompareTo(Car obj)
  {
    if (this.CarID > obj.CarID)
      return 1;
    if (this.CarID < obj.CarID)
      return -1;
    else
      return 0;
  }
}
```

Здесь уже не нужно проверять, относится ли входной параметр к типу Car, потому что он может быть *только* Car! В случае передачи несовместимого типа данных возникает ошибка на этапе компиляции. Теперь, когда вы лучше понимаете, как взаимодействовать с обобщенными элементами, а также роль параметров типа (т.е. заполнителей), вы готовы к исследованию классов и интерфейсов из пространства имен System.Collections.Generic.

Пространство имен System.Collections.Generic

Когда вы строите приложение .NET и необходим способ управления данными в памяти, классы из пространства имен System.Collections.Generic, скорее всего, удовлетворят всем требованиям. В начале настоящей главы кратко упоминались некоторые основные необобщенные интерфейсы, реализуемые необобщенными классами коллекций. Не должен вызывать удивление тот факт, что в пространстве имен System.Collections.Generic определены обобщенные замены для многих из них.

В действительности вы сможете найти некоторое количество обобщенных интерфейсов, которые расширяют свои необобщенные аналоги. Это может показаться странным; тем не менее, за счет этого реализующие их классы будут также поддерживать унаследованную функциональность, которая имеется в их необобщенных родственных версиях. Например, интерфейс IEnumerable<T> расширяет IEnumerable. В табл. 9.4 документированы основные обобщенные интерфейсы, с которыми вы столкнетесь во время работы с обобщенными классами коллекций.

Таблица 9.4. Основные интерфейсы, поддерживаемые классами из пространства имен System.Collections.Generic

Интерфейс System.Collections.Generic	Описание
ICollection<T>	Определяет общие характеристики (например, размер, перечисление и безопасность к потокам) для всех типов обобщенных коллекций
IComparer<T>	Определяет способ сравнения объектов
IDictionary<TKey, TValue>	Позволяет объекту обобщенной коллекции представлять свое содержимое посредством пар "ключ-значение"
IEnumerable<T>	Возвращает интерфейс IEnumerator<T> для заданного объекта
IEnumerator<T>	Позволяет выполнять итерацию в стиле foreach по элементам коллекции
IList<T>	Обеспечивает поведение добавления, удаления и индексации элементов в последовательном списке объектов
ISet<T>	Предоставляет базовый интерфейс для абстракции множеств

В пространстве имен System.Collections.Generic также определены классы, реализующие многие из указанных основных интерфейсов. В табл. 9.5 описаны часто используемые классы из этого пространства имен, реализуемые ими интерфейсы, а также их базовая функциональность.

В пространстве имен System.Collections.Generic также определены многие вспомогательные классы и структуры, которые работают в сочетании со специфическим контейнером. Например, тип LinkedListNode<T> представляет узел внутри обобщенного контейнера LinkedList<T>, исключение KeyNotFoundException генерируется при попытке получить элемент из коллекции, применяя несуществующий ключ, и т.д.

Таблица 9.5. Классы из пространства имен `System.Collections.Generic`

Обобщенный класс	Поддерживаемые основные интерфейсы	Описание
`Dictionary<TKey, TValue>`	`ICollection<T>`, `IDictionary<TKey, TValue>`, `IEnumerable<T>`	Представляет обобщенную коллекцию ключей и значений
`LinkedList<T>`	`ICollection<T>`, `IEnumerable<T>`	Представляет двухсвязный список
`List<T>`	`ICollection<T>`, `IEnumerable<T>`, `IList<T>`	Последовательный список элементов с динамически изменяемым размером
`Queue<T>`	`ICollection` (это не опечатка; именно так называется необобщенный интерфейс коллекции), `IEnumerable<T>`	Обобщенная реализация списка, работающего по алгоритму "первый вошел — первый вышел" (FIFO)
`SortedDictionary<TKey, TValue>`	`ICollection<T>`, `IDictionary<TKey, TValue>`, `IEnumerable<T>`	Обобщенная реализация сортированного множества пар "ключ-значение"
`SortedSet<T>`	`ICollection<T>`, `IEnumerable<T>`, `ISet<T>`	Представляет коллекцию объектов, поддерживаемых в сортированном порядке без дубликатов
`Stack<T>`	`ICollection` (это не опечатка; именно так называется необобщенный интерфейс коллекции), `IEnumerable<T>`	Обобщенная реализация списка, работающего по алгоритму "последний вошел — первый вышел" (LIFO)

Полезно отметить, что `mscorlib.dll` и `System.dll` — не единственные сборки, которые добавляют новые типы в пространство имен `System.Collections.Generic`. Например, `System.Core.dll` добавляет класс `HashSet<T>`. Подробные сведения о пространстве имен `System.Collections.Generic` доступны в документации .NET Framework.

В любом случае следующая ваша задача состоит в том, чтобы научиться использовать некоторые из этих классов обобщенных коллекций. Тем не менее, сначала полезно ознакомиться со средством языка C# (введенным в версии .NET 3.5), которое упрощает заполнение данными обобщенных (и необобщенных) коллекций.

Синтаксис инициализации коллекций

В главе 4 вы узнали о *синтаксисе инициализации массивов*, который позволяет устанавливать элементы новой переменной массива во время ее создания. С ним тесно связан *синтаксис инициализации коллекций*. Это средство языка C# позволяет наполнять многие контейнеры (такие как `ArrayList` или `List<T>`) элементами с применением синтаксиса, похожего на тот, который используется для наполнения базовых массивов.

На заметку! Синтаксис инициализации коллекций может быть применен только к классам, которые поддерживают метод `Add()`, формально определяемый интерфейсами `ICollection<T>` и `ICollection`.

Взгляните на следующие примеры:

```
// Инициализация стандартного массива.
int[] myArrayOfInts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Инициализация обобщенного List<> с элементами int.
List<int> myGenericList = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Инициализация ArrayList числовыми данными.
ArrayList myList = new ArrayList { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Если контейнером является коллекция классов или структур, то синтаксис инициализации коллекций можно смешивать с синтаксисом инициализации объектов, получая довольно функциональный код. Вспомните класс `Point` из главы 5, в котором были определены два свойства, `X` и `Y`. Для построения обобщенного списка `List<T>` объектов `P` можно написать такой код:

```
List<Point> myListOfPoints = new List<Point>
{
  new Point { X = 2, Y = 2 },
  new Point { X = 3, Y = 3 },
  new Point(PointColor.BloodRed){ X = 4, Y = 4 }
};

foreach (var pt in myListOfPoints)
{
  Console.WriteLine(pt);
}
```

Преимущество этого синтаксиса связано с сокращением объема клавиатурного ввода. Хотя вложенные фигурные скобки могут затруднять чтение, если не позаботиться о надлежащем форматировании, вы только вообразите себе объем кода, который пришлось бы написать для наполнения следующего списка `List<T>` объектов `Rectangle` без использования синтаксиса инициализации коллекций (класс `Rectangle` был создан в главе 4 и содержит два свойства, инкапсулирующие объекты `Point`):

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
  new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
                 BottomRight = new Point { X = 200, Y = 200}},
  new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
                 BottomRight = new Point { X = 100, Y = 100}},
  new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
                 BottomRight = new Point { X = 90, Y = 75}}
};

foreach (var r in myListOfRects)
{
  Console.WriteLine(r);
}
```

Работа с классом `List<T>`

Создадим новый проект консольного приложения по имени `FunWithGeneric Collections`. Обратите внимание, что в начальном файле кода C# пространство имен `System.Collections.Generic` уже импортировано.

Первым мы будем исследовать обобщенный класс `List<T>`, который уже применялся ранее в главе. Класс `List<T>` используется чаще других классов из пространства имен `System.Collections.Generic`, т.к. он позволяет динамически изменять размер кон-

тейнера. Чтобы проиллюстрировать его особенности, добавим в класс `Program` метод `UseGenericList()`, в котором задействован класс `List<T>` для манипулирования набором объектов `Person`; вспомните, что в классе `Person` определены три свойства (`Age`, `FirstName` и `LastName`), а также специальная реализация метода `ToString()`:

```
static void UseGenericList()
{
  // Создать список объектов Person и заполнить его с помощью
  // синтаксиса инициализации объектов и коллекции.
  List<Person> people = new List<Person>()
  {
    new Person {FirstName= "Homer", LastName="Simpson", Age=47},
    new Person {FirstName= "Marge", LastName="Simpson", Age=45},
    new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
    new Person {FirstName= "Bart", LastName="Simpson", Age=8}
  };

  // Вывести количество элементов в списке.
  Console.WriteLine("Items in list: {0}", people.Count);

  // Выполнить перечисление по списку.
  foreach (Person p in people)
    Console.WriteLine(p);

  // Вставить новый объект Person.
  Console.WriteLine("\n->Inserting new person.");
  people.Insert(2, new Person { FirstName = "Maggie",
                LastName = "Simpson", Age = 2 });
  Console.WriteLine("Items in list: {0}", people.Count);

  // Скопировать данные в новый массив.
  Person[] arrayOfPeople = people.ToArray();
  for (int i = 0; i < arrayOfPeople.Length; i++)
  {
    Console.WriteLine("First Names: {0}", arrayOfPeople[i].FirstName);
  }
}
```

Здесь для наполнения списка `List<T>` объектами применяется синтаксис инициализации в качестве сокращенной записи *многократного* вызова метода `Add()`. После вывода количества элементов в коллекции (а также перечисления по всем элементам) вызывается метод `Insert()`. Как видите, метод `Insert()` позволяет вставить новый элемент в `List<T>` по указанному индексу.

Наконец, обратите внимание на вызов метода `ToArray()`, который возвращает массив объектов `Person`, основанный на содержимом исходного списка `List<T>`. Затем осуществляется проход по всем элементам этого массива с использованием синтаксиса индексатора массива. Вызвав метод `UseGenericList()` из `Main()`, вы получите следующий вывод:

```
***** Fun with Generic Collections *****

Items in list: 4
Name: Homer Simpson, Age: 47
Name: Marge Simpson, Age: 45
Name: Lisa Simpson, Age: 9
Name: Bart Simpson, Age: 8

->Inserting new person.
Items in list: 5
```

```
First Names: Homer
First Names: Marge
First Names: Maggie
First Names: Lisa
First Names: Bart
```

В классе `List<T>` определено множество дополнительных членов, представляющих интерес, поэтому за полным их описанием обращайтесь в документацию .NET Framework. Давайте рассмотрим еще несколько обобщенных коллекций, в частности `Stack<T>`, `Queue<T>` и `SortedSet<T>`. Это должно обеспечить лучшее понимание основных вариантов хранения данных в приложении.

Работа с классом `Stack<T>`

Класс `Stack<T>` представляет коллекцию элементов, которая обслуживает элементы в стиле "последний вошел — первый вышел" (LIFO). Как и можно было ожидать, в `Stack<T>` определены члены `Push()` и `Pop()`, предназначенные для вставки и удаления элементов из стека. Приведенный ниже метод создает стек объектов `Person`:

```
static void UseGenericStack()
{
  Stack<Person> stackOfPeople = new Stack<Person>();
  stackOfPeople.Push(new Person
    { FirstName = "Homer", LastName = "Simpson", Age = 47 });
  stackOfPeople.Push(new Person
    { FirstName = "Marge", LastName = "Simpson", Age = 45 });
  stackOfPeople.Push(new Person
    { FirstName = "Lisa", LastName = "Simpson", Age = 9 });

  // Просмотреть верхний элемент, вытолкнуть его и просмотреть снова.
  Console.WriteLine("First person is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());

  Console.WriteLine("\nFirst person is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());

  Console.WriteLine("\nFirst person item is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());

  try
  {
    Console.WriteLine("\nnFirst person is: {0}", stackOfPeople.Peek());
    Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine("\nError! {0}", ex.Message);  // Ошибка! Стек пуст.
  }
}
```

Здесь мы строим стек, который содержит информацию о трех лицах, добавленных в порядке их имен: `Homer`, `Marge` и `Lisa`. Заглядывая (посредством метода `Peek()`) в стек, вы будете всегда видеть объект, находящийся на его вершине; следовательно, первый вызов `Peek()` возвращает третий объект `Person`. После серии вызовов `Pop()` и `Peek()` стек, в конце концов, опустошается, после чего дополнительные вызовы `Peek()` и `Pop()` приводят к генерации системного исключения. Вот как выглядит вывод этого примера:

```
***** Fun with Generic Collections *****

First person is: Name: Lisa Simpson, Age: 9
Popped off Name: Lisa Simpson, Age: 9

First person is: Name: Marge Simpson, Age: 45
Popped off Name: Marge Simpson, Age: 45

First person item is: Name: Homer Simpson, Age: 47
Popped off Name: Homer Simpson, Age: 47

Error! Stack empty.
```

Работа с классом Queue<T>

Очереди — это контейнеры, которые обеспечивают доступ к элементам в манере "первый вошел — первый вышел" (FIFO). К сожалению, людям приходится сталкиваться с очередями практически ежедневно: в банке, в супермаркете, в кафе. Когда нужно смоделировать сценарий, в котором элементы обрабатываются в режиме FIFO, класс `Queue<T>` подходит наилучшим образом. Дополнительно к функциональности, предоставляемой поддерживаемыми интерфейсами, в `Queue` определены основные члены, перечисленные в табл. 9.6.

Таблица 9.6. Члены типа Queue<T>

Член `Queue<T>`	Описание
`Dequeue()`	Удаляет и возвращает объект из начала `Queue<T>`
`Enqueue()`	Добавляет объект в конец `Queue<T>`
`Peek()`	Возвращает объект из начала `Queue<T>`, не удаляя его

Теперь давайте посмотрим на эти методы в работе. Можно снова задействовать класс `Person` и построить объект `Queue<T>`, эмулирующий очередь людей, которые ожидают заказа кофе. Первым делом предположим, что имеется следующий статический метод:

```
static void GetCoffee(Person p)
{
  Console.WriteLine("{0} got coffee!", p.FirstName);
}
```

Кроме того, есть также дополнительный вспомогательный метод, внутри которого вызывается `GetCoffee()`:

```
static void UseGenericQueue()
{
  // Создать очередь из трех человек.
  Queue<Person> peopleQ = new Queue<Person>();
  peopleQ.Enqueue(new Person {FirstName= "Homer",
    LastName="Simpson", Age=47});
  peopleQ.Enqueue(new Person {FirstName= "Marge",
    LastName="Simpson", Age=45});
  peopleQ.Enqueue(new Person {FirstName= "Lisa",
    LastName="Simpson", Age=9});

  // Заглянуть, кто первый в очереди.
  Console.WriteLine("{0} is first in line!", peopleQ.Peek().FirstName);

  // Удалить всех из очереди.
  GetCoffee(peopleQ.Dequeue());
```

```
    GetCoffee(peopleQ.Dequeue());
    GetCoffee(peopleQ.Dequeue());

    // Попробовать извлечь кого-то из очереди снова.
    try
    {
      GetCoffee(peopleQ.Dequeue());
    }
    catch(InvalidOperationException e)
    {
      Console.WriteLine("Error! {0}", e.Message);  // Ошибка! Очередь пуста.
    }
  }
```

В коде мы вставляем три элемента в `Queue<T>` с применением метода `Enqueue()`. Вызов `Peek()` позволяет просматривать (но не удалять) первый элемент, находящийся в текущий момент в `Queue`. Наконец, вызов `Dequeue()` удаляет элемент из очереди и передает его на обработку вспомогательной функции `GetCoffee()`. Обратите внимание, что если попробовать удалить элемент из пустой очереди, то сгенерируется исключение времени выполнения. Ниже показан вывод, полученный в результате вызова этого метода:

```
***** Fun with Generic Collections *****

Homer is first in line!
Homer got coffee!
Marge got coffee!
Lisa got coffee!
Error! Queue empty.
```

Работа с классом `SortedSet<T>`

Класс `SortedSet<T>` полезен тем, что при вставке или удалении элементов он автоматически обеспечивает сортировку элементов в наборе. Однако классу `SortedSet<T>` необходимо сообщить, *каким образом* должны сортироваться объекты, путем передачи его конструктору в качестве аргумента объекта, который реализует обобщенный интерфейс `IComparer<T>`.

Начнем с создания нового класса по имени `SortPeopleByAge`, реализующего интерфейс `IComparer<T>`, где `T` — тип `Person`. Вспомните, что в этом интерфейсе определен единственный метод по имени `Compare()`, в котором можно запрограммировать логику сравнения элементов. Вот простая реализация:

```
class SortPeopleByAge : IComparer<Person>
{
  public int Compare(Person firstPerson, Person secondPerson)
  {
    if (firstPerson.Age > secondPerson.Age)
      return 1;
    if (firstPerson.Age < secondPerson.Age)
      return -1;
    else
      return 0;
  }
}
```

Теперь добавим в класс `Program` следующий новый метод, который понадобится вызвать внутри `Main()`:

```
static void UseSortedSet()
{
  // Создать несколько объектов Person с разными значениями возраста.
  SortedSet<Person> setOfPeople =
       new SortedSet<Person>(new SortPeopleByAge())
  {
    new Person {FirstName= "Homer", LastName="Simpson", Age=47},
    new Person {FirstName= "Marge", LastName="Simpson", Age=45},
    new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
    new Person {FirstName= "Bart", LastName="Simpson", Age=8}
  };
  // Обратите внимание, что элементы отсортированы по возрасту.
  foreach (Person p in setOfPeople)
  {
    Console.WriteLine(p);
  }
    Console.WriteLine();
  // Добавить еще несколько объектов Person с разными значениями возраста.
  setOfPeople.Add(new Person { FirstName = "Saku",
                               LastName = "Jones", Age = 1 });
  setOfPeople.Add(new Person { FirstName = "Mikko",
                               LastName = "Jones", Age = 32 });
  // Элементы по-прежнему отсортированы по возрасту.
  foreach (Person p in setOfPeople)
  {
    Console.WriteLine(p);
  }
}
```

Запустив приложение, легко заметить, что список объектов будет всегда упорядочен на основе значения свойства `Age` независимо от порядка вставки и удаления объектов:

```
***** Fun with Generic Collections *****

Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47

Name: Saku Jones, Age: 1
Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Mikko Jones, Age: 32
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47
```

Работа с классом `Dictionary<TKey, TValue>`

Еще одной удобной обобщенной коллекцией является класс `Dictionary<TKey,TValue>`, позволяющий хранить любое количество объектов, на которые можно ссылаться через уникальный ключ. Таким образом, вместо получения элемента из `List<T>` с использованием числового идентификатора (например, "извлечь второй объект") можно применять уникальный строковый ключ (скажем, "предоставить объект с ключом `Homer`").

Как и другие классы коллекций, наполнять `Dictionary<TKey,TValue>` можно путем вызова обобщенного метода `Add()` вручную. Тем не менее, заполнить `Dictionary<TKey,TValue>` можно также с использованием синтаксиса инициализа-

ции коллекций. Имейте в виду, что при наполнении данного объекта коллекции ключи должны быть уникальными. Если вы по ошибке укажете один и тот же ключ несколько раз, то получите исключение времени выполнения.

Взгляните на следующий метод, который заполняет `Dictionary<K,V>` разнообразными объектами. Обратите внимание, что при создании объекта `Dictionary<TKey,TValue>` в качестве аргументов конструктора передаются тип ключа (`TKey`) и тип внутренних объектов (`TValue`). Здесь для ключа указывается тип данных `string` (хотя это не обязательно; ключ может относиться к любому типу), а для значения — тип `Person`:

```
private static void UseDictionary()
{
    // Наполнить с помощью метода Add().
    Dictionary<string, Person> peopleA = new Dictionary<string, Person>();
    peopleA.Add("Homer", new Person { FirstName = "Homer",
                                      LastName = "Simpson", Age = 47 });
    peopleA.Add("Marge", new Person { FirstName = "Marge",
                                      LastName = "Simpson", Age = 45 });
    peopleA.Add("Lisa", new Person { FirstName = "Lisa",
                                     LastName = "Simpson", Age = 9 });

    // Получить элемент с ключом Homer.
    Person homer = peopleA["Homer"];
    Console.WriteLine(homer);

    // Наполнить с помощью синтаксиса инициализации.
    Dictionary<string, Person> peopleB = new Dictionary<string, Person>()
    {
      { "Homer", new Person { FirstName = "Homer",
                              LastName = "Simpson", Age = 47 } },
      { "Marge", new Person { FirstName = "Marge",
                              LastName = "Simpson", Age = 45 } },
      { "Lisa", new Person { FirstName = "Lisa", LastName = "Simpson", Age = 9 } }
    };

    // Получить элемент с ключом Lisa.
    Person lisa = peopleB["Lisa"];
    Console.WriteLine(lisa);
}
```

Наполнять `Dictionary<TKey,TValue>` также возможно с применением связанного синтаксиса инициализации, появившегося в текущей версии .NET, который является специфичным для контейнера этого типа (вполне ожидаемо называемый *инициализацией словаря*). Подобно синтаксису, который использовался при наполнении объекта `personB` в предыдущем примере, для объекта коллекции определяется область инициализации; однако можно также применять индексатор, чтобы указать ключ, и присвоить ему новый объект:

```
// Наполнить с помощью синтаксиса инициализации словаря.
Dictionary<string, Person> peopleC = new Dictionary<string, Person>()
{
  ["Homer"] = new Person { FirstName = "Homer", LastName = "Simpson", Age = 47 },
  ["Marge"] = new Person { FirstName = "Marge", LastName = "Simpson", Age = 45 },
  ["Lisa"] = new Person { FirstName = "Lisa", LastName = "Simpson", Age = 9 }
};
```

Исходный код. Проект `FunWithGenericCollections` доступен в подкаталоге `Chapter_9`.

Пространство имен System.Collections.ObjectModel

Теперь, когда вы понимаете, как работать с основными обобщенными классами, мы можем кратко рассмотреть дополнительное пространство имен, связанное с коллекциями — System.Collections.ObjectModel. Это относительно небольшое пространство имен, содержащее лишь горстку классов. В табл. 9.7 документированы два класса, о которых вы должны быть обязательно осведомлены.

Таблица 9.7. Полезные классы из пространства имен System.Collections.ObjectModel

Класс System.Collections.ObjectModel	Описание
ObservableCollection<T>	Представляет динамическую коллекцию данных, которая обеспечивает уведомление при добавлении и удалении элементов, а также при обновлении всего списка
ReadOnlyObservableCollection<T>	Представляет версию ObservableCollection<T>, допускающую только чтение

Класс ObservableCollection<T> удобен тем, что он обладает возможностью информировать внешние объекты, когда его содержимое каким-то образом изменяется (как и можно было догадаться, работа с ReadOnlyObservableCollection<T> похожа, но предусматривает только чтение).

Работа с классом ObservableCollection<T>

Создадим новый проект консольного приложения по имени FunWithObservable Collection и импортируем в первоначальный файл кода C# пространство имен System. Collections.ObjectModel. Во многих отношениях работа с ObservableCollection<T> идентична работе с List<T>, учитывая, что оба класса реализуют те же самые основные интерфейсы. Уникальным класс ObservableCollection<T> делает тот факт, что он поддерживает событие по имени CollectionChanged. Это событие будет инициироваться каждый раз, когда вставляется новый элемент, удаляется (или перемещается) существующий элемент либо модифицируется вся коллекция целиком.

Подобно любому другому событию событие CollectionChanged определено в терминах делегата, которым в данном случае является NotifyCollectionChangedEventHandler. Этот делегат может вызывать любой метод, который принимает object в первом параметре и NotifyCollectionChangedEventArgs — во втором. Рассмотрим следующий метод Main(), который наполняет наблюдаемую коллекцию, содержащую объекты Person, и привязывается к событию CollectionChanged:

```
class Program
{
  static void Main(string[] args)
  {
    // Сделать коллекцию наблюдаемой и добавить в нее несколько объектов Person.
    ObservableCollection<Person> people = new ObservableCollection<Person>()
    {
      new Person{ FirstName = "Peter", LastName = "Murphy", Age = 52 },
```

```
      new Person{ FirstName = "Kevin", LastName = "Key", Age = 48 },
    };
    // Привязаться к событию CollectionChanged.
    people.CollectionChanged += people_CollectionChanged;
  }
  static void people_CollectionChanged(object sender,
    System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
  {
    throw new NotImplementedException();
  }
}
```

Входной параметр `NotifyCollectionChangedEventArgs` **определяет два важных свойства,** `OldItems` **и** `NewItems`**, которые выдают список элементов, имеющихся в коллекции перед генерацией события, и новых элементов, вовлеченных в изменение. Тем не менее, эти списки будут исследоваться только в подходящих обстоятельствах. Вспомните, что событие** `CollectionChanged` **инициируется, когда элементы добавляются, удаляются, перемещаются или сбрасываются. Чтобы выяснить, какое из упомянутых действий запустило событие, можно использовать свойство** `Action` **объекта** `NotifyCollectionChangedEventArgs`**. Свойство** `Action` **допускается проверять на равенство любому из членов перечисления** `NotifyCollectionChangedAction`:

```
public enum NotifyCollectionChangedAction
{
  Add = 0,
  Remove = 1,
  Replace = 2,
  Move = 3,
  Reset = 4,
}
```

Ниже показана реализация обработчика событий `CollectionChanged`**, который будет обходить старый и новый наборы, когда элемент вставляется или удаляется из имеющейся коллекции:**

```
static void people_CollectionChanged(object sender,
  System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
  // Выяснить действие, которое привело к генерации события.
  Console.WriteLine("Action for this event: {0}", e.Action);
  // Было что-то удалено.
  if (e.Action ==
     System.Collections.Specialized.NotifyCollectionChangedAction.Remove)
  {
    Console.WriteLine("Here are the OLD items:");
    foreach (Person p in e.OldItems)
    {
      Console.WriteLine(p.ToString());
    }
    Console.WriteLine();
  }
  // Было что-то добавлено.
  if (e.Action ==
      System.Collections.Specialized.NotifyCollectionChangedAction.Add)
  {
```

```
    // Теперь вывести новые элементы, которые были вставлены.
    Console.WriteLine("Here are the NEW items:");
    foreach (Person p in e.NewItems)
    {
      Console.WriteLine(p.ToString());
    }
  }
}
```

Предполагая, что вы модифицировали метод `Main()` для добавления и удаления элемента, вывод будет выглядеть следующим образом:

```
Action for this event: Add
Here are the NEW items:
Name: Fred Smith, Age: 32

Action for this event: Remove
Here are the OLD items:
Name: Peter Murphy, Age: 52
```

На этом исследование различных пространств имен, связанных с коллекциями, в библиотеках базовых классов .NET завершено. В конце главы будет также объясняться, как и для чего строить собственные обобщенные методы и обобщенные типы.

Исходный код. Проект `FunWithObservableCollection` доступен в подкаталоге `Chapter_9`.

Создание специальных обобщенных методов

Хотя большинство разработчиков обычно применяют обобщенные типы, существующие в библиотеках базовых классов, есть также возможность построения собственных обобщенных методов и специальных обобщенных типов. Давайте посмотрим, как включать обобщения в собственные проекты. Первым делом мы построим обобщенный метод обмена. Начнем с создания нового проекта консольного приложения по имени `CustomGenericMethods`.

Построение специальных обобщенных методов представляет собой более развитую версию традиционной перегрузки методов. В главе 2 вы узнали, что перегрузка — это действие по определению нескольких версий одного метода, которые отличаются друг от друга количеством или типами параметров.

Хотя перегрузка является полезным средством объектно-ориентированного языка, проблема заключается в том, что при этом довольно легко получить в итоге огромное количество методов, которые по существу делают одно и то же. Например, пусть необходимо создать методы, которые позволяют менять местами два фрагмента данных посредством простой процедуры. Для начала можно написать метод, оперирующий с целочисленными значениями:

```
// Обмен двух целочисленных значений.
static void Swap(ref int a, ref int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Пока все идет хорошо. Но теперь предположим, что нужно менять местами также и два объекта `Person`; это потребует новой версии метода `Swap()`:

```
// Обмен двух объектов Person.
static void Swap(ref Person a, ref Person b)
{
  Person temp;
  temp = a;
  a = b;
  b = temp;
}
```

Без сомнения вам должно быть ясно, чем это закончится. Если также понадобится менять местами два значения с плавающей точкой, два объекта растровых изображений, два объекта автомобилей, два объекта кнопок или еще что-нибудь, то придется писать дополнительные методы, что в итоге превратится в настоящий кошмар при сопровождении. Можно было бы построить один (необобщенный) метод, оперирующий с параметрами типа `object`, но тогда возвратятся все проблемы, которые были описаны ранее в главе, т.е. упаковка, распаковка, отсутствие безопасности к типам, явное приведение и т.д.

Наличие группы перегруженных методов, отличающихся только входными аргументами — это явный признак того, что обобщения могут облегчить ситуацию. Рассмотрим следующий обобщенный метод `Swap<T>`, который способен менять местами два значения типа `T`:

```
// Этот метод будет менять местами два элемента
// типа, указанного в параметре <T>.
static void Swap<T>(ref T a, ref T b)
{
  Console.WriteLine("You sent the Swap() method a {0}", typeof(T));
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

Обратите внимание, что обобщенный метод определен за счет указания параметра типа после имени метода, но перед списком параметров. Здесь устанавливается, что метод `Swap()` может оперировать с любыми двумя параметрами типа `<T>`. Для придания некоторой пикантности имя замещаемого типа выводится на консоль с использованием операции `typeof()` языка C#. Теперь взгляните на показанный ниже метод `Main()`, который меняет местами целочисленные и строковые значения:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom Generic Methods *****\n");

  // Обмен двух целочисленных значений.
  int a = 10, b = 90;
  Console.WriteLine("Before swap: {0}, {1}", a, b);
  Swap<int>(ref a, ref b);
  Console.WriteLine("After swap: {0}, {1}", a, b);
  Console.WriteLine();

  // Обмен двух строковых значений.
  string s1 = "Hello", s2 = "There";
  Console.WriteLine("Before swap: {0} {1}!", s1, s2);
  Swap<string>(ref s1, ref s2);
  Console.WriteLine("After swap: {0} {1}!", s1, s2);

  Console.ReadLine();
}
```

Вот вывод:

```
***** Fun with Custom Generic Methods *****

Before swap: 10, 90
You sent the Swap() method a System.Int32
After swap: 90, 10

Before swap: Hello There!
You sent the Swap() method a System.String
After swap: There Hello!
```

Главное преимущество такого подхода в том, что придется сопровождать только одну версию `Swap<T>()`, однако она способна работать с любыми двумя элементами заданного типа в безопасной к типам манере. Еще лучше то, что находящиеся в стеке элементы остаются в стеке, а расположенные в куче — соответственно, в куче.

Выведение параметров типа

При вызове обобщенных методов вроде `Swap<T>` параметр типа можно опускать, если (и только если) обобщенный метод принимает аргументы, поскольку компилятор в состоянии вывести параметр типа на основе параметров членов. Например, добавив в метод `Main()` следующий код, можно менять местами значения `System.Boolean`:

```
// Компилятор выведет тип System.Boolean.
bool b1 = true, b2 = false;
Console.WriteLine("Before swap: {0}, {1}", b1, b2);
Swap(ref b1, ref b2);
Console.WriteLine("After swap: {0}, {1}", b1, b2);
```

Несмотря на то что компилятор может определить параметр типа на основе типа данных, который применялся в объявлениях `b1` и `b2`, вы должны выработать привычку всегда указывать параметр типа явно:

```
Swap<string>(ref b1, ref b2);
```

Это позволяет другим программистам понять, что метод на самом деле является обобщенным. Кроме того, выведение типов параметров работает только в случае, если обобщенный метод принимает, по крайней мере, один параметр. Например, пусть в классе `Program` определен такой обобщенный метод:

```
static void DisplayBaseClass<T>()
{
  // BaseType - это метод, используемый в рефлексии;
  // он будет описан в главе 15.
  Console.WriteLine("Base class of {0} is: {1}.",
    typeof(T), typeof(T).BaseType);
}
```

При его вызове потребуется указать параметр типа:

```
static void Main(string[] args)
{
  ...
  // Если метод не принимает параметров, то должен быть указан параметр типа.
  DisplayBaseClass<int>();
  DisplayBaseClass<string>();
  // Ошибка на этапе компиляции! Нет параметров?
  // Должен быть предоставлен заполнитель!
  // DisplayBaseClass();
  Console.ReadLine();
}
```

В настоящее время обобщенные методы `Swap<T>` и `DisplayBaseClass<T>` определены в классе `Program` приложения. Разумеется, как и в случае любого метода, вы можете определить эти члены в отдельном типе класса (`MyGenericMethods`), если предпочитаете поступать подобным образом:

```
public static class MyGenericMethods
{
  public static void Swap<T>(ref T a, ref T b)
  {
    Console.WriteLine("You sent the Swap() method a {0}",
      typeof(T));
    T temp;
    temp = a;
    a = b;
    b = temp;
  }
  public static void DisplayBaseClass<T>()
  {
    Console.WriteLine(«Base class of {0} is: {1}.»,
      typeof(T), typeof(T).BaseType);
  }
}
```

Статические методы `Swap<T>` и `DisplayBaseClass<T>` находятся в области действия нового статического класса, поэтому при вызове любого члена необходимо указывать имя типа:

```
MyGenericMethods.Swap<int>(ref a, ref b);
```

Конечно, методы не обязательно должны быть статическими. Если бы `Swap<T>` и `DisplayBaseClass<T>` были методами уровня экземпляра (и определялись в нестатическом классе), то понадобилось бы просто создать экземпляр `MyGenericMethods` и вызвать их с использованием объектной переменной:

```
MyGenericMethods c = new MyGenericMethods();
c.Swap<int>(ref a, ref b);
```

Исходный код. Проект `CustomGenericMethods` доступен в подкаталоге `Chapter_9`.

Создание специальных обобщенных структур и классов

Теперь, когда вы знаете, как определять и вызывать обобщенные методы, наступило время уделить внимание конструированию обобщенной структуры (процесс построения обобщенного класса идентичен) в новом проекте консольного приложения по имени `GenericPoint`. Предположим, что вы построили обобщенную структуру `Point`, которая поддерживает единственный параметр типа, определяющий внутреннее представление координат (x, y). Затем вызывающий код может создавать типы `Point<T>` следующим образом:

```
// Точка с координатами типа int.
Point<int> p = new Point<int>(10, 10);

// Точка с координатами типа double.
Point<double> p2 = new Point<double>(5.4, 3.3);
```

Вот полное определение структуры `Point<T>`:

```
// Обобщенная структура Point.
public struct Point<T>
{
  // Обобщенные данные состояния.
  private T xPos;
  private T yPos;
  // Обобщенный конструктор.
  public Point(T xVal, T yVal)
  {
    xPos = xVal;
    yPos = yVal;
  }
  // Обобщенные свойства.
  public T X
  {
    get { return xPos; }
    set { xPos = value; }
  }
  public T Y
  {
    get { return yPos; }
    set { yPos = value; }
  }
  public override string ToString()
  {
    return string.Format("[{0}, {1}]", xPos, yPos);
  }
  // Сбросить поля в стандартные значения
  // для заданного параметра типа.
  public void ResetPoint()
  {
    xPos = default(T);
    yPos = default(T);
  }
}
```

Ключевое слово `default` в обобщенном коде

Как видите, структура `Point<T>` задействует параметр типа в определениях полей данных, в аргументах конструктора и в определениях свойств. Обратите внимание, что в дополнение к переопределению метода `ToString()` структура `Point<T>` определяет метод по имени `ResetPoint()`, в котором применяется не встречавшийся ранее новый синтаксис:

```
// Ключевое слово default в языке C# перегружено.
// При использовании с обобщениями оно представляет
// стандартное значение для параметра типа.
public void ResetPoint()
{
  X = default(T);
  Y = default(T);
}
```

С появлением обобщений ключевое слово `default` получило двойную идентичность. Вдобавок к использованию внутри конструкции `switch` оно также может применяться

для установки параметра типа в стандартное значение. Это очень удобно, т.к. действительные типы, подставляемые вместо заполнителей, обобщенному типу заранее не известны, поэтому он не может безопасно предполагать, какими будут стандартные значения. Параметры типа подчиняются следующим правилам:

- числовые типы имеют стандартное значение `0`;
- ссылочные типы имеют стандартное значение `null`;
- поля структур устанавливаются в `0` (для типов значений) или в `null` (для ссылочных типов).

Для `Point<T>` устанавливать значения `X` и `Y` в `0` можно напрямую, поскольку вполне безопасно предполагать, что вызывающий код будет предоставлять только числовые данные. Однако использование синтаксиса `default(T)` в целом повышает гибкость обобщенного типа. Теперь методы типа `Point<T>` можно применять так:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Generic Structures *****\n");

  // Точка с координатами типа int.
  Point<int> p = new Point<int>(10, 10);
  Console.WriteLine("p.ToString()={0}", p.ToString());
  p.ResetPoint();
  Console.WriteLine("p.ToString()={0}", p.ToString());
  Console.WriteLine();

  // Точка с координатами типа double.
  Point<double> p2 = new Point<double>(5.4, 3.3);
  Console.WriteLine("p2.ToString()={0}", p2.ToString());
  p2.ResetPoint();
  Console.WriteLine("p2.ToString()={0}", p2.ToString());
  Console.ReadLine();
}
```

Ниже приведен вывод:

```
***** Fun with Generic Structures *****

p.ToString()=[10, 10]
p.ToString()=[0, 0]

p2.ToString()=[5.4, 3.3]
p2.ToString()=[0, 0]
```

Исходный код. Проект `GenericPoint` доступен в подкаталоге `Chapter_9`.

Ограничение параметров типа

Как показано в настоящей главе, любой обобщенный элемент имеет, по крайней мере, один параметр типа, который необходимо указывать во время взаимодействия с данным обобщенным типом или его членом. Одно это обстоятельство уже позволяет строить код, безопасный к типам; тем не менее, платформа .NET позволяет использовать ключевое слово `where` для определения особых требований к отдельному параметру типа.

С помощью ключевого слова `this` можно добавлять набор ограничений к конкретному параметру типа, которые компилятор C# проверит на этапе компиляции. В частности, параметр типа можно ограничить, как описано в табл. 9.8.

Таблица 9.8. Возможные ограничения для параметров типа в обобщениях

Ограничение	Описание
`where T : struct`	Параметр типа <T> должен иметь класс `System.ValueType` в своей цепочке наследования (т.е. <T> должен быть структурой)
`where T : class`	Параметр типа <T> *не* должен иметь класс `System.ValueType` в своей цепочке наследования (т.е. <T> должен быть ссылочным типом)
`where T : new()`	Параметр типа <T> должен иметь стандартный конструктор. Это полезно, если обобщенный тип должен создавать экземпляры параметра типа, т.к. предугадать формат специальных конструкторов невозможно. Обратите внимание, что в типе с множеством ограничений данное ограничение должно указываться последним
`where T :` *ИмяБазовогоКласса*	Параметр типа <T> должен быть производным от класса, указанного как *ИмяБазовогоКласса*
`where T :` *ИмяИнтерфейса*	Параметр типа <T> должен реализовывать интерфейс, указанный как *ИмяИнтерфейса*. Можно задавать список из нескольких интерфейсов, разделяя их запятыми

Применять ключевое слово `where` в проектах C#, возможно, вам никогда и не придется, если только не требуется строить какие-то исключительно безопасные к типам специальные коллекции. Невзирая на это, в следующих нескольких примерах (частичного) кода демонстрируется работа с ключевым словом `where`.

Примеры использования ключевого слова `where`

Начнем с предположения о том, что создан специальный обобщенный класс, и необходимо гарантировать наличие в параметре типа стандартного конструктора. Это может быть полезно, когда специальный обобщенный класс должен создавать экземпляры типа `T`, потому что стандартный конструктор является единственным конструктором, потенциально общим для всех типов. Кроме того, подобное ограничение `T` позволяет получить проверку на этапе компиляции; если `T` — ссылочный тип, то программист будет помнить о повторном определении стандартного конструктора в объявлении класса (как вам уже известно, стандартный конструктор удаляется из класса в случае определения собственного конструктора).

```
// Класс MyGenericClass является производным от object, в то время как
// содержащиеся в нем элементы должны иметь стандартный конструктор.
public class MyGenericClass<T> where T : new()
{
  ...
}
```

Обратите внимание, что конструкция `where` указывает параметр типа, подлежащий ограничению, за которым следует операция двоеточия. После операции двоеточия перечисляются все возможные ограничения (в данном случае — стандартный конструктор). Вот еще один пример:

```
// Класс MyGenericClass является производным от object, в то время как
// содержащиеся в нем элементы должны относиться к классу, реализующему
// интерфейс IDrawable, и поддерживать стандартный конструктор.
public class MyGenericClass<T> where T : class, IDrawable, new()
{
  ...
}
```

Здесь к типу T предъявляются три требования. Во-первых, он должен быть ссылочным типом (не структурой), как помечено лексемой class. Во-вторых, T должен реализовывать интерфейс IDrawable. В-третьих, тип T также должен иметь стандартный конструктор. Множество ограничений перечисляются в виде списка с разделителями-запятыми, но имейте в виду, что ограничение new() должно быть указано последним! Таким образом, представленный далее код не скомпилируется:

```
// Ошибка! Ограничение new() должно быть последним в списке!
public class MyGenericClass<T> where T : new(), class, IDrawable
{
  ...
}
```

При создании класса обобщенной коллекции с несколькими параметрами типа можно указывать уникальный набор ограничений для каждого параметра, применяя отдельные конструкции where:

```
// Тип <K> должен расширять SomeBaseClass и иметь стандартный конструктор,
// в то время как тип <T> должен быть структурой и реализовывать
// обобщенный интерфейс IComparable.
public class MyGenericClass<K, T> where K : SomeBaseClass, new()
  where T : struct, IComparable<T>
{
  ...
}
```

Необходимость построения полного специального обобщенного класса коллекции возникает редко; однако ключевое слово where допускается использовать также в обобщенных методах. Например, если нужно гарантировать, что метод Swap<T>() может работать только со структурами, измените его код следующим образом:

```
// Этот метод меняет местами любые структуры, но не классы.
static void Swap<T>(ref T a, ref T b) where T : struct
{
  ...
}
```

Обратите внимание, что если ограничить метод Swap() в подобной манере, то менять местами объекты string (как было показано в коде примера) больше не удастся, т.к. string является ссылочным типом.

Отсутствие ограничений операций

В завершение главы следует упомянуть об еще одном факте, связанном с обобщенными методами и ограничениями. При создании обобщенных методов может оказаться неожиданным получение ошибок на этапе компиляции в случае применения к параметрам типа любых операций C# (+, -, *, == и т.д.). Например, только вообразите, насколько полезным оказался бы класс, способный выполнять сложение, вычитание, умножение и деление с обобщенными типами:

```
// Ошибка на этапе компиляции! Невозможно применять операции к параметрам типа!
public class BasicMath<T>
{
  public T Add(T arg1, T arg2)
  { return arg1 + arg2; }
  public T Subtract(T arg1, T arg2)
  { return arg1 - arg2; }
  public T Multiply(T arg1, T arg2)
  { return arg1 * arg2; }
  public T Divide(T arg1, T arg2)
  { return arg1 / arg2; }
}
```

К сожалению, приведенный выше класс BasicMath<T> не скомпилируется. Хотя это может показаться крупным недостатком, следует вспомнить, что обобщения имеют общий характер. Конечно, числовые данные прекрасно работают с двоичными операциями C#. Тем не менее, если аргумент <T> является специальным классом или структурой, то компилятор мог бы предположить, что он поддерживает операции +, -, * и /. В идеале язык C# позволял бы ограничивать обобщенный тип поддерживаемыми операциями, как показано ниже:

```
// Только в целях иллюстрации!
public class BasicMath<T> where T : operator +, operator -,
  operator *, operator /
{
  public T Add(T arg1, T arg2)
  { return arg1 + arg2; }
  public T Subtract(T arg1, T arg2)
  { return arg1 - arg2; }
  public T Multiply(T arg1, T arg2)
  { return arg1 * arg2; }
  public T Divide(T arg1, T arg2)
  { return arg1 / arg2; }
}
```

Увы, ограничения операций в текущей версии C# не поддерживаются. Однако достичь желаемого результата можно (хотя и с дополнительными усилиями) путем определения интерфейса, который поддерживает эти операции (интерфейсы C# могут определять операции!), и указания ограничения интерфейса для обобщенного класса. Итак, первоначальный обзор построения специальных обобщенных типов завершен. В главе 10 мы вновь обратимся к теме обобщений, когда будем исследовать тип делегата .NET.

Резюме

Настоящая глава начиналась с рассмотрения необобщенных типов коллекций в пространствах имен System.Collections и System.Collections.Specialized, включая разнообразные проблемы, которые связаны со многими необобщенными контейнерами, в том числе отсутствие безопасности к типам и накладные расходы времени выполнения в форме операций упаковки и распаковки. Как упоминалось, именно по этим причинам в современных приложениях .NET будут использоваться классы обобщенных коллекций из пространств имен System.Collections.Generic и System.Collections.ObjectModel.

Вы видели, что обобщенный элемент позволяет указывать заполнители (параметры типа), которые задаются во время создания объекта (или вызова в случае обобщенных методов). Хотя чаще всего вы будете просто применять обобщенные типы, предоставляемые библиотеками базовых классов .NET, также имеется возможность создавать собственные обобщенные типы (и обобщенные методы). При этом допускается указывать любое количество ограничений (с использованием ключевого слова where) для повышения уровня безопасности к типам и гарантирования того, что операции выполняются над типами *известного размера*, демонстрируя наличие определенных базовых возможностей.

В качестве финального замечания: не забывайте, что обобщения можно обнаружить во многих местах внутри библиотек базовых классов .NET. Здесь мы сосредоточились конкретно на обобщенных коллекциях. Тем не менее, по мере проработки материала оставшихся глав (и освоения платформы) вы наверняка найдете обобщенные классы, структуры и делегаты в том или ином пространстве имен. Кроме того, будьте готовы столкнуться с обобщенными членами в необобщенном классе!

ГЛАВА 10

Делегаты, события и лямбда-выражения

Вплоть до этого момента в большинстве разработанных приложений к методу `Main()` добавлялись разнообразные порции кода, тем или иным способом отправляющие запросы к заданному объекту. Однако многие приложения требуют, чтобы объект имел возможность обращаться *обратно* к сущности, которая его создала, используя механизм обратного вызова. Хотя механизмы обратного вызова могут применяться в любом приложении, они особенно важны в графических пользовательских интерфейсах, где элементы управления (такие как кнопки) нуждаются в вызове внешних методов при надлежащих обстоятельствах (когда произведен щелчок на кнопке, курсор мыши наведен на поверхность кнопки и т.д.).

В рамках платформы .NET предпочтительным средством определения и реагирования на обратные вызовы в приложении является тип *делегата*. По существу тип делегата .NET — это безопасный к типам объект, "указывающий" на метод или список методов, которые могут быть вызваны позднее. Тем не менее, в отличие от традиционного указателя на функцию C++ делегаты .NET представляют собой классы, которые обладают встроенной поддержкой группового и асинхронного вызова методов.

В этой главе вы узнаете, каким образом создавать и манипулировать типами делегатов, а также использовать ключевое слово `event` языка C#, которое облегчает работу с типами делегатов. По ходу дела вы также изучите несколько языковых средств C#, ориентированных на делегаты и события, в том числе анонимные методы и групповые преобразования методов.

Глава завершается исследованием *лямбда-выражений*. С помощью лямбда-операции C# (=>) можно указывать блок операторов кода (и параметры, подлежащие передаче этим операторам) везде, где требуется строго типизированный делегат. Как будет показано, лямбда-выражение является не более чем замаскированным анонимным методом и представляет собой упрощенный подход к работе с делегатами. Вдобавок та же самая операция (в версии .NET 4.6) может применяться для реализации метода или свойства, содержащего единственный оператор, посредством лаконичного синтаксиса.

Понятие типа делегата .NET

Прежде чем формально определить делегаты .NET, давайте оглянемся немного назад. Исторически сложилось так, что в API-интерфейсе Windows часто использовались указатели на функции в стиле C для создания сущностей под названием *функции обратного вызова* или просто *обратные вызовы*. С помощью обратных вызовов программисты могли конфигурировать одну функцию так, чтобы она обращалась к другой функции

в приложении (т.е. делала обратный вызов). Применяя такой подход, разработчики Windows-приложений имели возможность обрабатывать щелчки на кнопках, перемещение курсора мыши, выбор пунктов меню и общие двусторонние коммуникации между двумя сущностями в памяти.

В .NET Framework обратные вызовы выполняются в безопасной к типам объектно-ориентированной манере с использованием *делегатов*. В сущности, делегат — это безопасный в отношении типов объект, указывающий на другой метод или возможно на список методов приложения, которые могут быть вызваны в более позднее время. В частности, делегат поддерживает три важных порции информации:

- адрес метода, на котором он вызывается;
- аргументы (если есть) этого метода;
- возвращаемое значение (если есть) этого метода.

На заметку! Делегаты .NET могут указывать либо на статические методы, либо на методы экземпляра.

После того как делегат создан и снабжен необходимой информацией, он может динамически вызывать метод или методы, на которые указывает, во время выполнения. Каждый делегат в .NET Framework (включая ваши специальные делегаты) автоматически оснащается способностью вызывать свои методы *синхронно* или *асинхронно*. Данный факт значительно упрощает задачи программирования, поскольку метод можно вызывать во вторичном потоке выполнения без ручного создания и управления объектом `Thread`.

На заметку! Вы ознакомитесь с асинхронным поведением типов делегатов во время исследования многопоточности и асинхронных вызовов в главе 19. Здесь же мы будем касаться только синхронных аспектов типа делегата.

Определение типа делегата в C#

Для определения типа делегата в языке C# применяется ключевое слово `delegate`. Имя типа делегата может быть любым желаемым. Однако сигнатура определяемого делегата должна совпадать с сигнатурой метода или методов, на которые он будет указывать. Например, приведенный ниже тип делегата (по имени `BinaryOp`) может указывать на любой метод, который возвращает целое число и принимает два целых числа в качестве входных параметров (вы самостоятельно построите этот делегат позже в главе, а пока что он показан кратко):

```
// Этот делегат может указывать на любой метод, который принимает
// два целочисленных значения и возвращает целое значение.
public delegate int BinaryOp(int x, int y);
```

Когда компилятор C# обрабатывает тип делегата, он автоматически генерирует запечатанный (`sealed`) класс, производный от `System.MulticastDelegate`. Этот класс (в сочетании с его базовым классом `System.Delegate`) предоставляет необходимую инфраструктуру для делегата, позволяющую хранить список методов, которые подлежат вызову в будущем. Например, если вы изучите делегат `BinaryOp` с помощью утилиты `ildasm.exe`, то обнаружите класс, показанный на рис. 10.1 (если хотите, можете построить этот полный пример).

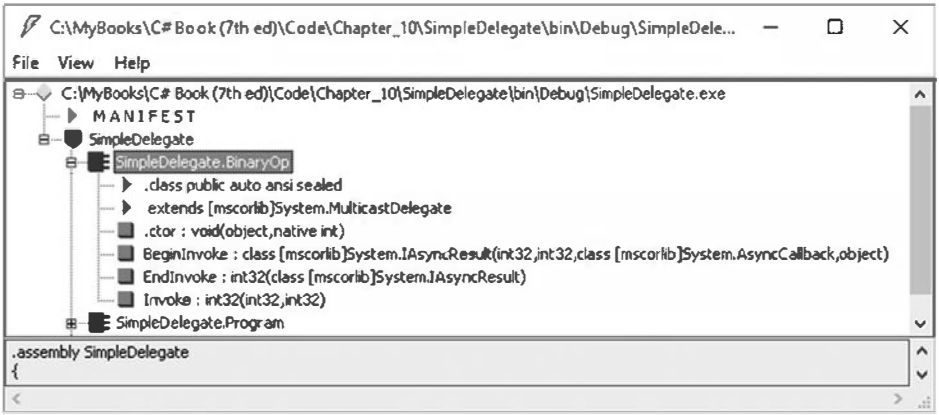


Рис. 10.1. Ключевое слово `delegate` языка C# представляет запечатанный класс, производный от `System.MulticastDelegate`

Как видите, сгенерированный компилятором класс `BinaryOp` определяет три открытых метода. Вероятно, главным из них является `Invoke()`, т.к. он используется для вызова каждого метода, поддерживаемого объектом делегата, в *синхронной* манере; это означает, что вызывающий код должен ожидать завершения вызова, прежде чем продолжить свою работу. Довольно странно, но синхронный метод `Invoke()` может не нуждаться в явном вызове внутри вашего кода C#. Вскоре будет показано, что `Invoke()` вызывается "за кулисами", когда вы применяете соответствующий синтаксис C#.

Методы `BeginInvoke()` и `EndInvoke()` обеспечивают возможность вызова текущего метода *асинхронным* образом в отдельном потоке выполнения. Если у вас есть опыт многопоточной разработки, то вы знаете, что одна из наиболее распространенных причин, вынуждающих разработчиков создавать вторичные потоки выполнения, связана с необходимостью вызова методов, которые требуют определенного времени на свое завершение. Хотя в библиотеках базовых классов .NET поставляется несколько пространств имен, предназначенных для многопоточного и параллельного программирования, делегаты предлагают такую функциональность в готовом виде.

Так благодаря чему же компилятор знает, как определять методы `Invoke()`, `BeginInvoke()` и `EndInvoke()`? Для понимания этого процесса ниже приведен код сгенерированного компилятором класса `BinaryOp` (полужирным курсивом выделены элементы, указанные в определении типа делегата):

```
sealed class BinaryOp : System.MulticastDelegate
{
  public int Invoke(int x, int y);
  public IAsyncResult BeginInvoke(int x, int y,
    AsyncCallback cb, object state);
  public int EndInvoke(IAsyncResult result);
}
```

Первым делом обратите внимание, что параметры и возвращаемый тип для метода `Invoke()` в точности соответствуют определению делегата `BinaryOp`. Начальные параметры для членов `BeginInvoke()` (в данном случае два целых числа) также основаны на делегате `BinaryOp`; тем не менее, `BeginInvoke()` всегда будет предоставлять два финальных параметра (типа `AsyncCallback` и `object`), которые используются для облегчения асинхронного вызова методов. Наконец, возвращаемый тип `EndInvoke()` идентичен исходному объявлению делегата и будет всегда принимать единственный параметр — объект, реализующий интерфейс `IAsyncResult`.

Давайте рассмотрим еще один пример. Предположим, что определен тип делегата, который может указывать на любой метод, возвращающий значение `string` и принимающий три входных параметра типа `System.Boolean`:

```
public delegate string MyDelegate(bool a, bool b, bool c);
```

На этот раз сгенерированный компилятором класс можно представить так:

```
sealed class MyDelegate : System.MulticastDelegate
{
  public string Invoke(bool a, bool b, bool c);
  public IAsyncResult BeginInvoke(bool a, bool b, bool c,
    AsyncCallback cb, object state);
  public string EndInvoke(IAsyncResult result);
}
```

Делегаты могут также "указывать" на методы, которые содержат любое количество параметров `out` и `ref` (а также параметры типа массивов, помеченные с помощью ключевого слова `params`). Например, пусть имеется следующий тип делегата:

```
public delegate string MyOtherDelegate(out bool a, ref bool b, int c);
```

Сигнатуры методов `Invoke()` и `BeginInvoke()` выглядят вполне ожидаемо; однако взгляните на метод `EndInvoke()`, который теперь включает набор аргументов `out/ref`, определенных типом делегата:

```
public sealed class MyOtherDelegate : System.MulticastDelegate
{
  public string Invoke(out bool a, ref bool b, int c);
  public IAsyncResult BeginInvoke(out bool a, ref bool b, int c,
    AsyncCallback cb, object state);
  public string EndInvoke(out bool a, ref bool b, IAsyncResult result);
}
```

Чтобы подвести итог: определение типа делегата C# дает в результате запечатанный класс с тремя сгенерированными компилятором методами, в которых типы параметров и возвращаемые типы основаны на объявлении делегата. Базовый шаблон может быть описан с помощью следующего псевдокода:

```
// Это всего лишь псевдокод!
public sealed class ИмяДелегата : System.MulticastDelegate
{
  public возвращаемоеЗначениеДелегата Invoke(всеВходныеRefиOutПараметрыДелегата);
  public IAsyncResult BeginInvoke(всеВходныеRefиOutПараметрыДелегата,
    AsyncCallback cb, object state);
  public возвращаемоеЗначениеДелегата EndInvoke(всеRefиOutПараметрыДелегата,
    IAsyncResult result);
}
```

Базовые классы `System.MulticastDelegate` и `System.Delegate`

Итак, когда вы строите тип с применением ключевого слова `delegate`, то неявно объявляете тип класса, производного от `System.MulticastDelegate`. Этот класс предоставляет своим наследникам доступ к списку, который содержит адреса методов, поддерживаемых типом делегата, а также несколько дополнительных методов (и перегруженных операций) для взаимодействия со списком вызовов. Ниже приведены избранные методы класса `System.MulticastDelegate`:

```
public abstract class MulticastDelegate : Delegate
{
  // Возвращает список методов, на которые "указывает" делегат.
  public sealed override Delegate[] GetInvocationList();
  // Перегруженные операции.
  public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2);
  public static bool operator !=(MulticastDelegate d1, MulticastDelegate d2);
  // Используются внутренне для управления списком методов,
  // поддерживаемых делегатом.
  private IntPtr _invocationCount;
  private object _invocationList;
}
```

Класс `System.MulticastDelegate` получает дополнительную функциональность от своего родительского класса `System.Delegate`. Вот фрагмент его определения:

```
public abstract class Delegate : ICloneable, ISerializable
{
  // Методы для взаимодействия со списком функций.
  public static Delegate Combine(params Delegate[] delegates);
  public static Delegate Combine(Delegate a, Delegate b);
  public static Delegate Remove(Delegate source, Delegate value);
  public static Delegate RemoveAll(Delegate source, Delegate value);
  // Перегруженные операции.
  public static bool operator ==(Delegate d1, Delegate d2);
  public static bool operator !=(Delegate d1, Delegate d2);
  // Свойства, открывающие доступ к цели делегата.
  public MethodInfo Method { get; }
  public object Target { get; }
}
```

Имейте в виду, что вы никогда не сможете напрямую наследовать от этих базовых классов в своем коде (попытка сделать это приводит к ошибке на этапе компиляции). Тем не менее, когда вы используете ключевое слово `delegate`, то тем самым неявно создаете класс, который "является" `MulticastDelegate`. В табл. 10.1 описаны основные члены, общие для всех типов делегатов.

Таблица 10.1. Избранные члены классов `System.MultcastDelegate/System.Delegate`

Член	Назначение
`Method`	Это свойство возвращает объект `System.Reflection.Method`, который представляет детали статического метода, поддерживаемого делегатом
`Target`	Если метод, подлежащий вызову, определен на уровне объектов (т.е. он нестатический), то это свойство возвращает объект, который представляет метод, поддерживаемый делегатом. Если возвращенное `Target` значение равно `null`, то подлежащий вызову метод является статическим
`Combine()`	Этот статический метод добавляет метод в список, поддерживаемый делегатом. В языке C# данный метод вызывается с применением перегруженной операции += в качестве сокращенной записи
`GetInvokationList()`	Этот метод возвращает массив объектов `System.Delegate`, каждый из которых представляет определенный метод, доступный для вызова
`Remove()` `RemoveAll()`	Эти статические методы удаляют метод (или все методы) из списка вызовов делегата. В языке C# метод `Remove()` может быть вызван косвенно с использованием перегруженной операции -=

Пример простейшего делегата

На первый взгляд делегаты могут показаться несколько запутанными. Рассмотрим для начала простой проект консольного приложения (по имени SimpleDelegate), в котором применяется определенный ранее тип делегата BinaryOp. Ниже показан полный код с последующим анализом:

```
namespace SimpleDelegate
{
  // Этот делегат может указывать на любой метод, который принимает
  // два целочисленных значения и возвращает целое значение.
  public delegate int BinaryOp(int x, int y);

  // Этот класс содержит методы, на которые
  // будет указывать BinaryOp.
  public class SimpleMath
  {
    public static int Add(int x, int y)
    { return x + y; }
    public static int Subtract(int x, int y)
    { return x - y; }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Simple Delegate Example *****\n");

      // Создать объект делегата BinaryOp,
      // который "указывает" на SimpleMath.Add().
      BinaryOp b = new BinaryOp(SimpleMath.Add);

      // Вызвать метод Add() косвенно с использованием объекта делегата.
      Console.WriteLine("10 + 10 is {0}", b(10, 10));
      Console.ReadLine();
    }
  }
}
```

И снова обратите внимание на формат объявления типа делегата BinaryOp; он определяет, что объекты делегата BinaryOp могут указывать на любой метод, принимающий два целочисленных значения и возвращающий целое значение (действительное имя метода, на который он указывает, к делу не относится). Здесь мы создали класс по имени SimpleMath, определяющий два статических метода, которые соответствуют шаблону, определяемому делегатом BinaryOp.

Когда вы хотите присвоить целевой метод заданному объекту делегата, просто передайте имя нужного метода конструктору делегата:

```
// Создать объект делегата BinaryOp,
// который "указывает" на SimpleMath.Add().
BinaryOp b = new BinaryOp(SimpleMath.Add);
```

На этой стадии метод, на который указывает делегат, можно вызывать с использованием синтаксиса, выглядящего подобным прямому вызову функции:

```
// В действительно здесь вызывается метод Invoke()!
Console.WriteLine("10 + 10 is {0}", b(10, 10));
```

"За кулисами" исполняющая среда на самом деле вызывает сгенерированный компилятором метод `Invoke()` на вашем производном от `MulticastDelegate` классе. В этом можно удостовериться, открыв сборку в утилите `ildasm.exe` и просмотрев код CIL внутри метода `Main()`:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  ...
  callvirt instance int32 SimpleDelegate.BinaryOp::Invoke(int32, int32)
}
```

Язык C# вовсе не требует явного вызова метода `Invoke()` внутри вашего кода. Поскольку `BinaryOp` может указывать на методы, которые принимают два аргумента, следующий оператор кода также допустим:

```
Console.WriteLine("10 + 10 is {0}", b.Invoke(10, 10));
```

Вспомните, что делегаты .NET *безопасны в отношении типов*. Следовательно, если вы попытаетесь передать делегату метод, который не соответствует его шаблону, то получите ошибку на этапе компиляции. В целях иллюстрации предположим, что в классе `SimpleMath` теперь определен дополнительный метод по имени `SquareNumber()`, принимающий единственный целочисленный аргумент:

```
public class SimpleMath
{
  ...
  public static int SquareNumber(int a)
  { return a * a; }
}
```

Учитывая, что делегат `BinaryOp` может указывать *только* на методы, которые принимают два целочисленных значения и возвращают целое значение, представленный ниже код некорректен и приведет к ошибке на этапе компиляции:

```
// Ошибка на этапе компиляции! Метод не соответствует шаблону делегата!
BinaryOp b2 = new BinaryOp(SimpleMath.SquareNumber);
```

Исследование объекта делегата

Давайте усложним текущий пример, создав в классе `Program` статический метод (по имени `DisplayDelegateInfo()`). Он будет выводить на консоль имена методов, поддерживаемых объектом делегата, а также имя класса, определяющего метод. Для этого организуется итерация по массиву `System.Delegate`, возвращенному методом `GetInvocationList()`, с обращением к свойствам `Target` и `Method` каждого объекта:

```
static void DisplayDelegateInfo(Delegate delObj)
{
  // Вывести имена всех членов в списке вызовов делегата.
  foreach (Delegate d in delObj.GetInvocationList())
  {
    Console.WriteLine("Method Name: {0}", d.Method);  // Имя метода
    Console.WriteLine("Type Name: {0}", d.Target);    // Имя типа
  }
}
```

Предположив, что в метод `Main()` добавлен вызов нового вспомогательного метода:

```
BinaryOp b = new BinaryOp(SimpleMath.Add);
DisplayDelegateInfo(b);
```

вывод приложения будет таким:

```
***** Simple Delegate Example *****

Method Name: Int32 Add(Int32, Int32)
Type Name:
10 + 10 is 20
```

Обратите внимание, что имя целевого класса (SimpleMath) в настоящий момент *не* отображается при обращении к свойству Target. Причина в том, что делегат BinaryOp указывает на *статический метод*, поэтому объект для ссылки попросту отсутствует! Однако если сделать методы Add() и Substract() нестатическими (удалив ключевое слово static из их объявлений), то можно будет создавать экземпляр класса SimpleMath и указывать методы для вызова с применением ссылки на объект:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Delegate Example *****\n");

  // Делегаты .NET могут также указывать на методы экземпляра.
  SimpleMath m = new SimpleMath();
  BinaryOp b = new BinaryOp(m.Add);

  // Вывести сведения об объекте.
  DisplayDelegateInfo(b);

  Console.WriteLine("10 + 10 is {0}", b(10, 10));
  Console.ReadLine();
}
```

В данном случае вывод будет выглядеть следующим образом:

```
***** Simple Delegate Example *****

Method Name: Int32 Add(Int32, Int32)
Type Name: SimpleDelegate.SimpleMath
10 + 10 is 20
```

Исходный код. Проект SimpleDelegate доступен в подкаталоге Chapter_10.

Отправка уведомлений о состоянии объекта с использованием делегатов

Очевидно, что предыдущий пример SimpleDelegate был чисто иллюстративным по своей природе, т.к. нет особых причин создавать делегат просто для того, чтобы сложить два числа. Рассмотрим более реалистичный пример, в котором делегаты применяются для определения класса Car, обладающего способностью информировать внешние сущности о текущем состоянии двигателя. Для этого понадобится выполнить перечисленные ниже действия.

1. Определить новый тип делегата, который будет использоваться для отправки уведомлений вызывающему коду.
2. Объявить переменную-член этого типа делегата в классе Car.
3. Создать в классе Car вспомогательную функцию, которая позволяет вызывающему коду указывать метод для обратного вызова.
4. Реализовать метод Accelerate() для обращения к списку вызовов делегата в подходящих обстоятельствах.

Для начала создадим новый проект консольного приложения по имени CarDelegate. Определим в нем новый класс Car, начальный код которого показан ниже:

```
public class Car
{
  // Данные состояния.
  public int CurrentSpeed { get; set; }
  public int MaxSpeed { get; set; } = 100;
  public string PetName { get; set; }

  // Исправен ли автомобиль?
  private bool carIsDead;

  // Конструкторы класса.
  public Car() {}
  public Car(string name, int maxSp, int currSp)
  {
    CurrentSpeed = currSp;
    MaxSpeed = maxSp;
    PetName = name;
  }
}
```

А теперь модифицируем его, выполнив первые три действия из числа указанных выше:

```
public class Car
{
  ...
  // 1. Определить тип делегата.
  public delegate void CarEngineHandler(string msgForCaller);

  // 2. Определить переменную-член этого типа делегата.
  private CarEngineHandler listOfHandlers;

  // 3. Добавить регистрационную функцию для вызывающего кода.
  public void RegisterWithCarEngine(CarEngineHandler methodToCall)
  {
    listOfHandlers = methodToCall;
  }
}
```

Обратите внимание в данном примере, что типы делегатов определяются прямо внутри области действия класса Car; безусловно, это необязательно, но помогает закрепить идею о том, что делегат естественным образом работает с этим отдельным классом. Тип делегата CarEngineHandler может указывать на любой метод, принимающий значение string в качестве параметра и имеющий void в качестве типа возврата.

Кроме того, была объявлена закрытая переменная-член делегата (listOfHandlers) и вспомогательная функция (RegisterWithCarEngine()), которая позволяет вызывающему коду добавлять метод в список вызовов делегата.

На заметку! Строго говоря, переменную-член типа делегата можно было бы определить как public, избежав тем самым необходимости в создании дополнительных методов регистрации. Тем не менее, за счет определения этой переменной-члена типа делегата как private усиливается инкапсуляция и обеспечивается решение, более безопасное в отношении типов. Мы еще вернемся к анализу рисков объявления переменных-членов с типами делегатов как public позже в этой главе, когда будем рассматривать ключевое слово event языка C#.

Теперь необходимо создать метод `Accelerate()`. Вспомните, что цель здесь в том, чтобы позволить объекту `Car` отправлять связанные с двигателем сообщения любому подписавшемуся прослушивателю. Вот необходимое обновление:

```
// 4. Реализовать метод Accelerate() для обращения к списку
//    вызовов делегата в подходящих обстоятельствах.
public void Accelerate(int delta)
{
  // Если этот автомобиль сломан, отправить сообщение об этом.
  if (carIsDead)
  {
    if (listOfHandlers != null)
      listOfHandlers("Sorry, this car is dead...");
  }
  else
  {
    CurrentSpeed += delta;

    // Автомобиль почти сломан?
    if (10 == (MaxSpeed - CurrentSpeed)
        && listOfHandlers != null)
    {
      listOfHandlers("Careful buddy! Gonna blow!");
    }
    if (CurrentSpeed >= MaxSpeed)
      carIsDead = true;
    else
      Console.WriteLine("CurrentSpeed = {0}", CurrentSpeed);
  }
}
```

Обратите внимание, что прежде чем вызывать методы, поддерживаемые переменной-членом `listOfHandlers`, ее значение проверяется на равенство `null`. Причина в том, что создание этих объектов посредством вызова вспомогательного метода `RegisterWithCarEngine()` является задачей вызывающего кода. Если вызывающий код не вызывал этот метод, а мы попытаемся обратиться к списку вызовов делегата, то получим исключение `NullReferenceException` во время выполнения. Теперь, имея всю инфраструктуру делегатов, рассмотрим модификацию класса `Program`:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Delegates as event enablers *****\n");

    // Сначала создать объект Car.
    Car c1 = new Car("SlugBug", 100, 10);

    // Теперь сообщить ему, какой метод вызывать,
    // когда он пожелает отправить сообщение.
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

    // Увеличить скорость (это инициирует события).
    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);
    Console.ReadLine();
  }
```

```
  // Это цель для входящих сообщений.
  public static void OnCarEngineEvent(string msg)
  {
    Console.WriteLine("\n***** Message From Car Object *****");
    Console.WriteLine("=> {0}", msg);
    Console.WriteLine("************************************\n");
  }
}
```

Метод `Main()` начинается с создания нового объекта `Car`. Поскольку мы заинтересованы в событиях, связанных с двигателем, следующий шаг заключается в вызове специальной регистрационной функции `RegisterWithCarEngine()`. Вспомните, что этот метод ожидает получения экземпляра вложенного делегата `CarEngineHandler`, и как с любым делегатом, метод, на который он должен указывать, передается в параметре конструктора. Трюк здесь в том, что интересующий метод находится в классе `Program`! Опять-таки, обратите внимание, что метод `OnCarEngineEvent()` полностью соответствует связанному делегату, т.к. принимает `string` и возвращает `void`. Ниже показан вывод этого примера:

```
***** Delegates as event enablers *****
***** Speeding up *****
CurrentSpeed = 30
CurrentSpeed = 50
CurrentSpeed = 70
***** Message From Car Object *****
=> Careful buddy! Gonna blow!
************************************
CurrentSpeed = 90
***** Message From Car Object *****
=> Sorry, this car is dead...
************************************
```

Включение группового вызова

Вспомните, что делегаты .NET обладают встроенной возможностью *группового вызова*. Другими словами, объект делегата может поддерживать целый список методов для вызова, а не просто единственный такой метод. Для добавления нескольких методов к объекту делегата вместо прямого присваивания применяется перегруженная операция `+=`. Чтобы включить групповой вызов в классе `Car`, можно модифицировать метод `RegisterWithCarEngine()`:

```
public class Car
{
  // Добавление поддержки группового вызова.
  // Обратите внимание на использование операции +=,
  // а не обычной операции присваивания (=).
  public void RegisterWithCarEngine(CarEngineHandler methodToCall)
  {
    listOfHandlers += methodToCall;
  }
  ...
}
```

Когда операция `+=` используется с объектом делегата, компилятор преобразует это в вызов статического метода `Delegate.Combine()`. В действительности можно было бы вызывать `Delegate.Combine()` напрямую, однако операция `+=` предлагает более

простую альтернативу. Нет никакой необходимости в модификации текущего метода RegisterWithCarEngine(), но ниже представлен пример применения Delegate.Combine() вместо операции +=:

```
public void RegisterWithCarEngine( CarEngineHandler methodToCall )
{
  if (listOfHandlers == null)
    listOfHandlers = methodToCall;
  else
    Delegate.Combine(listOfHandlers, methodToCall);
}
```

В любом случае вызывающий код теперь может регистрировать множественные цели для одного и того же обратного вызова. Здесь второй обработчик выводит входное сообщение в верхнем регистре просто в целях отображения:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Delegates as event enablers *****\n");

    // Создать объект Car.
    Car c1 = new Car("SlugBug", 100, 10);

    // Зарегистрировать несколько обработчиков событий.
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent2));

    // Увеличить скорость (это инициирует события).
    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);
    Console.ReadLine();
  }

  // Теперь есть ДВА метода, которые будут
  // вызываться Car при отправке уведомлений.
  public static void OnCarEngineEvent(string msg)
  {
    Console.WriteLine("\n***** Message From Car Object *****");
    Console.WriteLine("=> {0}", msg);
    Console.WriteLine("************************************\n");
  }

  public static void OnCarEngineEvent2(string msg)
  {
    Console.WriteLine("=> {0}", msg.ToUpper());
  }
}
```

Удаление целей из списка вызовов делегата

В классе Delegate также определен статический метод Remove(), который позволяет вызывающему коду динамически удалять отдельные методы из списка вызовов объекта делегата. Это позволяет вызывающему коду легко "отменять подписку" на заданное уведомление во время выполнения. Хотя метод Delegate.Remove() можно вызывать в коде напрямую, разработчики C# могут использовать операцию -= в качестве удобного сокращения. Давайте добавим в класс Car новый метод, который позволяет вызывающему коду исключать метод из списка вызовов:

```
public class Car
{
  ...
  public void UnRegisterWithCarEngine(CarEngineHandler methodToCall)
  {
    listOfHandlers -= methodToCall;
  }
}
```

При таком обновлении класса `Car` прекратить получение уведомлений от второго обработчика можно за счет изменения метода `Main()` следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Delegates as event enablers *****\n");

  // Создать объект Car.
  Car c1 = new Car("SlugBug", 100, 10);
  c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

  // На этот раз сохранить объект делегата,
  // чтобы позже можно было отменить регистрацию.
  Car.CarEngineHandler handler2 = new Car.CarEngineHandler(OnCarEngineEvent2);
  c1.RegisterWithCarEngine(handler2);

  // Увеличить скорость (это инициирует события).
  Console.WriteLine("***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);

  // Отменить регистрацию второго обработчика.
  c1.UnRegisterWithCarEngine(handler2);

  // Сообщения в верхнем регистре больше не выводятся.
  Console.WriteLine("***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);

  Console.ReadLine();
}
```

Метод `Main()` отличается тем, что на этот раз создается объект `Car.CarEngineHandler`, который сохраняется в локальной переменной, чтобы можно было впоследствии отменить подписку на получение уведомлений. Таким образом, при втором увеличении скорости в объекте `Car` версия входящего сообщения в верхнем регистре больше выводиться не будет, поскольку эта цель исключена из списка вызовов делегата.

Исходный код. Проект `CarDelegate` доступен в подкаталоге `Chapter_10`.

Синтаксис групповых преобразований методов

В предыдущем примере `CarDelegate` явно создавались экземпляры класса делегата `Car.CarEngineHandler` для регистрации и отмены регистрации на получение уведомлений:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Delegates as event enablers *****\n");
  Car c1 = new Car("SlugBug", 100, 10);
  c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));
```

```
  Car.CarEngineHandler handler2 =
    new Car.CarEngineHandler(OnCarEngineEvent2);
  c1.RegisterWithCarEngine(handler2);
  ...
}
```

Конечно, если нужно вызывать любые унаследованные члены класса `MulticastDelegate` или `Delegate`, то для этого проще всего вручную создать переменную делегата. Однако в большинстве случаев иметь дело с внутренним устройством объекта делегата не требуется. Объект делегата обычно необходимо применять только для передачи имени метода в параметре конструктора.

Для простоты в языке C# предлагается сокращение, называемое *групповым преобразованием методов*. Это средство позволяет указывать вместо объекта делегата прямое имя метода, когда вызываются методы, которые принимают делегаты в качестве аргументов.

На заметку! Позже в этой главе вы увидите, что синтаксис группового преобразования методов можно также использовать для упрощения регистрации событий C#.

В целях иллюстрации создадим новый проект консольного приложения по имени `CarDelegateMethodGroupConversion` и добавим к нему файл, содержащий класс `Car`, который был определен в проекте `CarDelegate` (а также скорректируем название пространства имен в файле `Car.cs`). В показанном ниже классе `Program` для регистрации и отмены регистрации подписки на уведомления применяется групповое преобразование методов:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Method Group Conversion *****\n");
    Car c1 = new Car();

    // Зарегистрировать простое имя метода.
    c1.RegisterWithCarEngine(CallMeHere);

    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);

    // Отменить регистрацию простого имени метода.
    c1.UnRegisterWithCarEngine(CallMeHere);

    // Уведомления больше не поступают!
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);

    Console.ReadLine();
  }

  static void CallMeHere(string msg)
  {
    Console.WriteLine("=> Message from Car: {0}", msg);
  }
}
```

Обратите внимание, что мы не создаем напрямую ассоциированный объект делегата, а просто указываем метод, который соответствует ожидаемой сигнатуре делегата (в этом случае метод, возвращающий `void` и принимающий единственный аргумент

string). Имейте в виду, что компилятор C# по-прежнему обеспечивает безопасность к типам. Таким образом, если метод CallMeHere() не принимает string и не возвращает void, то возникнет ошибка на этапе компиляции.

Исходный код. Проект CarDelegateMethodGroupConversion доступен в подкаталоге Chapter_10.

Понятие обобщенных делегатов

В предыдущей главе упоминалось о том, что язык C# позволяет определять обобщенные типы делегатов. Например, предположим, что необходимо определить тип делегата, который может вызывать любой метод, возвращающий void и принимающий единственный параметр. Если передаваемый аргумент может изменяться, то это легко смоделировать с использованием параметра типа. Взгляните на следующий код внутри нового проекта консольного приложения по имени GenericDelegate:

```
namespace GenericDelegate
{
  // Этот обобщенный делегат может вызывать любой метод, который
  // возвращает void и принимает единственный параметр типа T.
  public delegate void MyGenericDelegate<T>(T arg);

  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Generic Delegates *****\n");

      // Зарегистрировать цели.
      MyGenericDelegate<string> strTarget =
        new MyGenericDelegate<string>(StringTarget);
      strTarget("Some string data");

      MyGenericDelegate<int> intTarget =
        new MyGenericDelegate<int>(IntTarget);
      intTarget(9);
      Console.ReadLine();
    }

    static void StringTarget(string arg)
    {
      Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
    }

    static void IntTarget(int arg)
    {
      Console.WriteLine("++arg is: {0}", ++arg);
    }
  }
}
```

Как видите, в типе делегата MyGenericDelegate<T> определен единственный параметр, представляющий аргумент для передачи цели делегата. При создании экземпляра этого типа должно быть указано значение параметра типа наряду с именем метода, который делегат может вызывать. Таким образом, если указать тип string, то целевому методу будет отправляться строковое значение:

```
// Создать экземпляр MyGenericDelegate<T> с
// указанием string в качестве параметра типа.
MyGenericDelegate<string> strTarget =
  new MyGenericDelegate<string>(StringTarget);
strTarget("Some string data");
```

С учетом формата объекта `strTarget` метод `StringTarget` теперь должен принимать в качестве параметра единственную строку:

```
static void StringTarget(string arg)
{
  Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}
```

Исходный код. Проект `GenericDelegate` доступен в подкаталоге `Chapter_10`.

Обобщенные делегаты `Action<>` и `Func<>`

В настоящей главе вы уже видели, что когда нужно применять делегаты для обратных вызовов в приложениях, обычно должны быть выполнены следующие шаги:

- определение специального делегата, соответствующего формату метода, на который он указывает;
- создание экземпляра специального делегата с передачей имени метода в качестве аргумента конструктора;
- косвенное обращение к методу через вызов `Invoke()` на объекте делегата.

В случае принятия такого подхода в итоге, как правило, получается несколько специальных делегатов, которые могут никогда не использоваться за рамками текущей задачи (например, `MyGenericDelegate<T>`, `CarEngineHandler` и т.д.). Хотя вполне может быть и так, что для проекта требуется специальный уникально именованный делегат, в других ситуациях точное *имя* типа делегата роли не играет. Во многих случаях просто необходим "какой-нибудь делегат", который принимает набор аргументов и возможно возвращает значение, отличное от `void`. В таких ситуациях можно применять встроенные в платформу делегаты `Action<>` и `Func<>`. Чтобы продемонстрировать их полезность, создадим новый проект консольного приложения по имени `ActionAndFuncDelegates`.

Обобщенный делегат `Action<>` определен в пространствах имен `System` внутри сборок `mscorlib.dll` и `System.Core.dll`. Его можно использовать для "указания" на метод, который принимает вплоть до *16 аргументов* (чего должно быть достаточно!) и возвращает `void`. Вспомните, что поскольку `Action<>` является обобщенным делегатом, понадобится также указывать типы всех параметров.

Модифицируем класс `Program`, определив в нем новый статический метод, который принимает три (или около того) уникальных параметра:

```
// Это цель для делегата Action<>.
static void DisplayMessage(string msg, ConsoleColor txtColor, int printCount)
{
  // Установить цвет текста консоли.
  ConsoleColor previous = Console.ForegroundColor;
  Console.ForegroundColor = txtColor;

  for (int i = 0; i < printCount; i++)
  {
    Console.WriteLine(msg);
  }
```

```
  // Восстановить цвет.
  Console.ForegroundColor = previous;
}
```

Теперь вместо построения специального делегата вручную для передачи потока программы методу `DisplayMessage()` мы можем применять готовый делегат `Action<>`, как показано ниже:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Action and Func *****");
  // Использовать делегат Action<> для указания на DisplayMessage.
  Action<string, ConsoleColor, int> actionTarget =
    new Action<string, ConsoleColor, int>(DisplayMessage);
  actionTarget("Action Message!", ConsoleColor.Yellow, 5);
  Console.ReadLine();
}
```

Как видите, при использовании делегата `Action<>` не нужно беспокоиться об определении специального типа делегата. Тем не менее, как уже упоминалось, тип делегата `Action<>` позволяет указывать только на методы, возвращающие `void`. Если вы хотите указывать на метод, имеющий возвращаемое значение (и нет желания заниматься написанием собственного типа делегата), можно применять тип делегата `Func<>`.

Обобщенный делегат `Func<>` способен указывать на методы, которые (подобно `Action<>`) принимают вплоть до 16 параметров и имеют специальное возвращаемое значение. В целях иллюстрации добавим в класс `Program` новый метод:

```
// Цель для делегата Func<>.
static int Add(int x, int y)
{
  return x + y;
}
```

Ранее в этой главе был построен специальный делегат `BinaryOp` для "указания" на методы сложения и вычитания. Теперь задачу можно упростить за счет использования версии `Func<>`, которая принимает всего три параметра типа. Учтите, что *последний* параметр в `Func<>` *всегда* представляет возвращаемое значение метода. Чтобы закрепить данный момент, предположим, что в классе `Program` также определен следующий метод:

```
static string SumToString(int x, int y)
{
  return (x + y).ToString();
}
```

Вызовем эти методы внутри `Main()`:

```
Func<int, int, int> funcTarget = new Func<int, int, int>(Add);
int result = funcTarget.Invoke(40, 40);
Console.WriteLine("40 + 40 = {0}", result);

Func<int, int, string> funcTarget2 = new Func<int, int,
string>(SumToString);
string sum = funcTarget2(90, 300);
Console.WriteLine(sum);
```

Синтаксис групповых преобразований методов позволяет упростить предшествующий код:

```
Func<int, int, int> funcTarget = Add;
int result = funcTarget.Invoke(40, 40);
Console.WriteLine("40 + 40 = {0}", result);

Func<int, int, string> funcTarget2 = SumToString;
string sum = funcTarget2(90, 300);
Console.WriteLine(sum);
```

С учетом того, что делегаты `Action<>` и `Func<>` могут устранить шаг по ручному определению специального делегата, вас может интересовать, должны ли вы применять их все время. Подобно большинству аспектов программирования, ответом будет: в зависимости от ситуации. Во многих случаях `Action<>` и `Func<>` будут предпочтительным вариантом. Однако если необходим делегат со специальным именем, которое, как вам кажется, помогает лучше отразить предметную область, то построение специального делегата сводится к единственному оператору кода. В оставшихся материалах книги вы увидите оба подхода.

На заметку! Делегаты `Action<>` и `Func<>` интенсивно используются во многих важных API-интерфейсах .NET, включая инфраструктуру параллельного программирования и LINQ (помимо прочих).

Итак, на этом первоначальный экскурс в типы делегатов .NET завершен. Мы обратимся к ряду дополнительных деталей работы с делегатами в конце настоящей главы и еще раз в главе 19, когда будем рассматривать многопоточность и асинхронные вызовы. А теперь давайте перейдем к обсуждению связанной темы — ключевого слова `event` языка C#.

Исходный код. Проект `ActionAndFuncDelegates` доступен в подкаталоге `Chapter_10`.

Понятие событий C#

Делегаты — довольно интересные конструкции в том плане, что позволяют объектам, находящимся в памяти, участвовать в двустороннем взаимодействии. Тем не менее, работа с делегатами непосредственно может приводить к написанию стереотипного кода (определение делегата, определение необходимых переменных-членов, создание специальных методов регистрации и отмены регистрации для предохранения инкапсуляции и т.д.).

Более того, во время применения делегатов непосредственным образом как механизма обратного вызова в приложениях, если вы не определите переменную-член типа делегата в классе как закрытую, то вызывающий код будет иметь прямой доступ к объектам делегатов. В этом случае вызывающий код может присвоить переменной-члену новый объект делегата (фактически удалив текущий список функций, подлежащих вызову) и, что даже хуже, вызывающий код сможет напрямую обращаться к списку вызовов делегата. Чтобы проиллюстрировать проблему, рассмотрим следующую переделанную (и упрощенную) версию класса `Car` из предыдущего примера `CarDelegate`:

```
public class Car
{
  public delegate void CarEngineHandler(string msgForCaller);

  // Теперь это член public!
  public CarEngineHandler listOfHandlers;

  // Просто вызвать уведомление Exploded.
```

```
  public void Accelerate(int delta)
  {
    if (listOfHandlers != null)
      listOfHandlers("Sorry, this car is dead...");
  }
}
```

Обратите внимание, что теперь больше нет закрытых переменных-членов с типами делегатов, инкапсулированных с помощью специальных методов регистрации. Поскольку эти члены на самом деле открытые, вызывающий код может получить доступ прямо к переменной-члену `listOfHandlers`, присвоить ей новые объекты `CarEngineHandler` и вызвать делегат по своему желанию:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Agh! No Encapsulation! *****\n");
    // Создать объект Car.
    Car myCar = new Car();
    // Есть прямой доступ к делегату!
    myCar.listOfHandlers = new Car.CarEngineHandler(CallWhenExploded);
    myCar.Accelerate(10);

    // Теперь можно присвоить полностью новый объект...
    // сбивает с толку в лучшем случае.
    myCar.listOfHandlers = new Car.CarEngineHandler(CallHereToo);
    myCar.Accelerate(10);

    // Вызывающий код может также напрямую вызывать делегат!
    myCar.listOfHandlers.Invoke("hee, hee, hee...");
    Console.ReadLine();
  }

  static void CallWhenExploded(string msg)
  { Console.WriteLine(msg); }

  static void CallHereToo(string msg)
  { Console.WriteLine(msg); }
}
```

Открытие доступа к членам типа делегатов нарушает инкапсуляцию, что не только затруднит сопровождение кода (и отладку), но также сделает приложение уязвимым в плане безопасности! Ниже показан вывод текущего примера:

```
***** Agh! No Encapsulation! *****

Sorry, this car is dead...
Sorry, this car is dead...
hee, hee, hee...
```

Очевидно, что вы не захотите предоставлять другим приложениям возможность изменять то, на что указывает делегат, или вызывать его члены без вашего разрешения. С учетом этого общепринятая практика предусматривает объявление переменных-членов, имеющих типы делегатов, как закрытых.

Исходный код. Проект `PublicDelegateProblem` доступен в подкаталоге `Chapter_10`.

Ключевое слово event

В качестве сокращения, избавляющего от необходимости создавать специальные методы для добавления и удаления методов из списка вызовов делегата, в языке C# предусмотрено ключевое слово event. В результате обработки компилятором ключевого слова event вы автоматически получаете методы регистрации и отмены регистрации, а также все необходимые переменные-члены для типов делегатов. Такие переменные-члены с типами делегатов *всегда* объявляются как закрытые, поэтому они не доступны напрямую из объекта, инициирующего событие. В итоге ключевое слово event может использоваться для упрощения отправки специальным классом уведомлений внешним объектам.

Определение события представляет собой двухэтапный процесс. Во-первых, понадобится определить тип делегата (или задействовать существующий тип), который будет хранить список методов, подлежащих вызову при возникновении события. Во-вторых, необходимо объявить событие (с применением ключевого слова event) в терминах связанного типа делегата.

Чтобы продемонстрировать использование ключевого слова event, создадим новый проект консольного приложения по имени CarEvents. В этой версии класса Car будут определены два события под названиями AboutToBlow и Exploded, которые ассоциированы с единственным типом делегата по имени CarEngineHandler. Ниже показаны начальные изменения, внесенные в класс Car:

```
public class Car
{
  // Этот делегат работает в сочетании с событиями Car.
  public delegate void CarEngineHandler(string msg);

  // Car может отправлять следующие события:
  public event CarEngineHandler Exploded;
  public event CarEngineHandler AboutToBlow;
 ...
}
```

Отправка события вызывающему коду — это просто указание события по имени наряду со всеми обязательными параметрами, как определено ассоциированным делегатом. Чтобы удостовериться в том, что вызывающий код действительно зарегистрировал событие, перед вызовом набора методов делегата событие следует проверить на равенство null. Ниже приведена новая версия метода Accelerate() класса Car:

```
public void Accelerate(int delta)
{
  // Если автомобиль сломан, инициировать событие Exploded.
  if (carIsDead)
  {
    if (Exploded != null)
      Exploded("Sorry, this car is dead...");
  }
  else
  {
    CurrentSpeed += delta;

    // Почти сломан?
    if (10 == MaxSpeed - CurrentSpeed
      && AboutToBlow != null)
    {
      AboutToBlow("Careful buddy! Gonna blow!");
    }
```

```
      // Все еще в порядке!
      if (CurrentSpeed >= MaxSpeed)
        carIsDead = true;
      else
        Console.WriteLine("CurrentSpeed = {0}", CurrentSpeed);
    }
  }
```

Итак, класс `Car` был сконфигурирован для отправки двух специальных событий без необходимости в определении специальных функций регистрации или в объявлении переменных-членов, имеющих типы делегатов. Применение этого нового объекта вы увидите очень скоро, но сначала давайте рассмотрим архитектуру событий немного подробнее.

"За кулисами" событий

Когда компилятор C# обрабатывает ключевое слово `event`, он генерирует два скрытых метода, один из которых имеет префикс `add_`, а другой — `remove_`. За префиксом следует имя события C#. Например, событие `Exploded` дает в результате два скрытых метода с именами `add_Exploded()` и `remove_Exploded()`. Если заглянуть в код CIL метода `add_AboutToBlow()`, то можно обнаружить вызов метода `Delegate.Combine()`. Взгляните на частичный код CIL:

```
.method public hidebysig specialname instance void
add_AboutToBlow(class CarEvents.Car/CarEngineHandler 'value') cil managed
{
  ...
  call class [mscorlib]System.Delegate
  [mscorlib]System.Delegate::Combine(
    class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
  ...
}
```

Как и можно было ожидать, метод `remove_AboutToBlow()` будет вызывать `Delegate.Remove()`:

```
.method public hidebysig specialname instance void
  remove_AboutToBlow(class CarEvents.Car/CarEngineHandler 'value')
  cil managed
{
  ...
  call class [mscorlib]System.Delegate
    [mscorlib]System.Delegate::Remove(
      class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
  ...
}
```

Наконец, в коде CIL, представляющем само событие, используются директивы `.addon` и `.removeon` для отображения на имена корректных методов `add_XXX()` и `remove_XXX()`, подлежащих вызову:

```
.event CarEvents.Car/EngineHandler AboutToBlow
{
  .addon instance void CarEvents.Car::add_AboutToBlow
    (class CarEvents.Car/CarEngineHandler)
  .removeon instance void CarEvents.Car::remove_AboutToBlow
    (class CarEvents.Car/CarEngineHandler)
}
```

Теперь, когда вы понимаете, каким образом строить класс, способный отправлять события C# (и знаете, что события — всего лишь способ сэкономить время на наборе кода), следующий крупный вопрос связан с организацией прослушивания входящих событий на стороне вызывающего кода.

Прослушивание входящих событий

События C# также упрощают действие по регистрации обработчиков событий на стороне вызывающего кода. Вместо того чтобы указывать специальные вспомогательные методы, вызывающий код просто применяет операции += и -= напрямую (что приводит к внутренним вызовам методов `add_XXX()` или `remove_XXX()`). При регистрации события руководствуйтесь показанным ниже шаблоном:

```
// ИмяОбъекта.ИмяСобытия += new СвязанныйДелегат(функцияДляВызова);
//
Car.CarEngineHandler d = new Car.CarEngineHandler(CarExplodedEventHandler);
myCar.Exploded += d;
```

Для отключения от источника событий служит операция -= в соответствии со следующим шаблоном:

```
// ИмяОбъекта.ИмяСобытия -= СвязанныйДелегат(функцияДляВызова);
//
myCar.Exploded -= d;
```

Имея эти весьма предсказуемые шаблоны, переделаем метод `Main()`, используя на этот раз синтаксис регистрации методов C#:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Events *****\n");
    Car c1 = new Car("SlugBug", 100, 10);

    // Зарегистрировать обработчики событий.
    c1.AboutToBlow += new Car.CarEngineHandler(CarIsAlmostDoomed);
    c1.AboutToBlow += new Car.CarEngineHandler(CarAboutToBlow);

    Car.CarEngineHandler d = new Car.CarEngineHandler(CarExploded);
    c1.Exploded += d;

    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);

    // Удалить метод CarExploded из списка вызовов.
    c1.Exploded -= d;

    Console.WriteLine("\n***** Speeding up *****");
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);
    Console.ReadLine();
  }
  public static void CarAboutToBlow(string msg)
  { Console.WriteLine(msg); }
  public static void CarIsAlmostDoomed(string msg)
  { Console.WriteLine("=> Critical Message from Car: {0}", msg); }
  public static void CarExploded(string msg)
  { Console.WriteLine(msg); }
}
```

Чтобы еще больше упростить регистрацию событий, можно применять групповое преобразование методов. Вот очередная модификация метода `Main()`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Events *****\n");
  Car c1 = new Car("SlugBug", 100, 10);
  // Зарегистрировать обработчики событий.
  c1.AboutToBlow += CarIsAlmostDoomed;
  c1.AboutToBlow += CarAboutToBlow;
  c1.Exploded += CarExploded;
  Console.WriteLine("***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);
  c1.Exploded -= CarExploded;
  Console.WriteLine("\n***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);
  Console.ReadLine();
}
```

Упрощение регистрации событий с использованием Visual Studio

Среда Visual Studio предоставляет помощь в процессе регистрации обработчиков событий. В случае применения синтаксиса += во время регистрации событий открывается окно IntelliSense, приглашающее нажать клавишу <Tab> для автоматического завершения связанного экземпляра делегата (рис. 10.2), что достигается с использованием *синтаксиса групповых преобразований методов*.

После нажатия клавиши <Tab> будет сгенерирован новый метод, как показано на рис. 10.3. Обратите внимание, что код заглушки имеет корректный формат цели делегата (кроме того, этот метод объявлен как `static`, т.к. событие было зарегистрировано внутри статического метода):

```
static void NewCar_AboutToBlow(string msg)
{
  // Удалите следующую строку и добавьте свой код!
  throw new NotImplementedException();
}
```

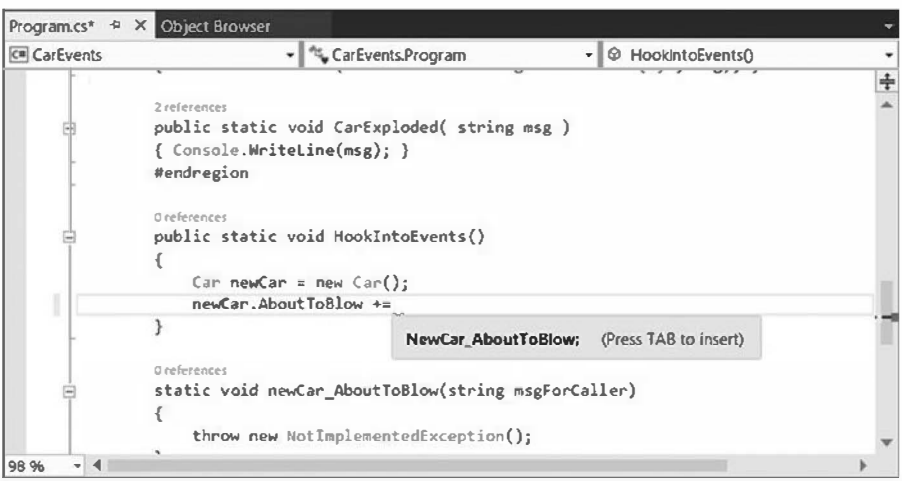


Рис. 10.2. Выбор делегата с помощью средства IntelliSense

Средство IntelliSense доступно для всех событий .NET из библиотек базовых классов. Эта возможность IDE-среды значительно экономит время, избавляя от необходимости выяснять с помощью справочной системы .NET подходящий тип делегата для применения с заданным событием и формат целевого метода делегата.

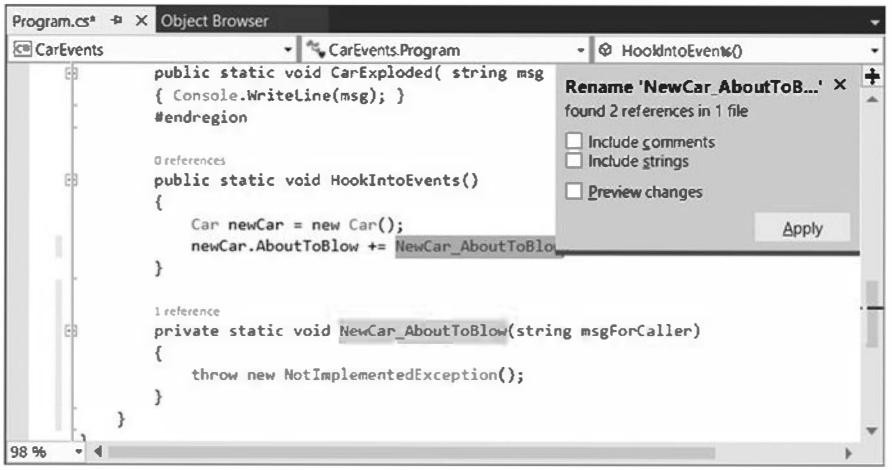


Рис. 10.3. Форматирование цели делегата средством IntelliSense

Приведение в порядок кода обращения к событиям с использованием `null`-условной операции C# 6.0

В рассматриваемом примере вы наверняка заметили, что перед отправкой события любому прослушивателю производится проверка на равенство `null`. Это важно, т.к. если событие никто не прослушивает, но вы в любом случае его инициируете, то при выполнении будет получено исключение, связанное со ссылкой `null`. В то же время вы согласитесь с тем, что код с многочисленными проверками на предмет `null` выглядит несколько неуклюжим.

К счастью, в текущей версии языка C# можно задействовать `null`-условную операцию (`?`), которая по существу выполняет проверку такого рода автоматически. Имейте в виду, что при использовании нового упрощенного синтаксиса потребуется вручную вызывать метод `Invoke()` лежащего в основе делегата. Например, вместо кода:

```
// Если автомобиль сломан, инициировать событие Exploded.
if (carIsDead)
{
  if (Exploded != null)
    Exploded("Sorry, this car is dead...");
}
```

теперь можно применять следующий код:

```
// Если автомобиль сломан, инициировать событие Exploded.
if (carIsDead)
{
  Exploded?.Invoke("Sorry, this car is dead...");
}
```

Кроме того, аналогичным образом можно также обновить код, инициирующий событие `AboutToBlow` (обратите внимание, что здесь убрана проверка на равенство `null`, которая присутствовала в первоначальном операторе `if`):

```
// Почти сломан?
if (10 == MaxSpeed - CurrentSpeed)
{
  AboutToBlow?.Invoke("Careful buddy!  Gonna blow!");
}
```

Из-за такого упрощенного синтаксиса, скорее всего, вы отдадите предпочтение null-условной операции при инициировании событий. Однако реализация проверки на предмет null вручную, когда это необходимо, по-прежнему остается совершенно приемлемой.

Исходный код. Проект CarEvents доступен в подкаталоге Chapter_10.

Создание специальных аргументов событий

По правде говоря, в текущую итерацию класса Car можно было бы внести последнее усовершенствование, которое отражает рекомендованный Microsoft шаблон событий. Если вы начнете исследовать события, отправляемые определенным типом из библиотек базовых классов, то обнаружите, что первый параметр лежащего в основе делегата имеет тип System.Object, в то время как второй — тип, производный от System.EventArgs.

Параметр System.Object представляет ссылку на объект, который отправляет событие (такой как Car), а второй параметр — информацию, относящуюся к обрабатываемому событию. Базовый класс System.EventArgs представляет событие, которое не сопровождается какой-либо специальной информацией:

```
public class EventArgs
{
  public static readonly EventArgs Empty;
  public EventArgs();
}
```

Для простых событий экземпляр EventArgs можно передать напрямую. Тем не менее, когда нужно передавать специальные данные, вы должны построить подходящий класс, производный от EventArgs. В этом примере предположим, что есть класс по имени CarEventArgs, который поддерживает строковое представление сообщения, отправленного получателю:

```
public class CarEventArgs : EventArgs
{
  public readonly string msg;
  public CarEventArgs(string message)
  {
    msg = message;
  }
}
```

Теперь можно модифицировать тип делегата CarEngineHandler, как показано ниже (события не будут изменяться):

```
public class Car
{
  public delegate void CarEngineHandler(object sender, CarEventArgs e);
  ...
}
```

Здесь при инициировании событий внутри метода `Accelerate()` необходимо использовать ссылку на текущий объект `Car` (посредством ключевого слова `this`) и экземпляр типа `CarEventArgs`. Например, рассмотрим следующее обновление:

```
public void Accelerate(int delta)
{
  // Если этот автомобиль сломан, инициировать событие Exploded.
  if (carIsDead)
  {
    Exploded?.Invoke(this, new CarEventArgs("Sorry, this car is dead..."));
  }
  ...
}
```

На вызывающей стороне понадобится только обновить обработчики событий для приема входных параметров и получения сообщения через поле, доступное только для чтения. Вот пример:

```
public static void CarAboutToBlow(object sender, CarEventArgs e)
{
  Console.WriteLine("{0} says: {1}", sender, e.msg);
}
```

Если получатель желает взаимодействовать с объектом, отправившим событие, то можно выполнить явное приведение `System.Object`. Такая ссылка позволит вызывать любой открытый метод объекта, который отправил уведомление:

```
public static void CarAboutToBlow(object sender, CarEventArgs e)
{
  // Просто для подстраховки перед приведением
  // произвести проверку во время выполнения.
  if (sender is Car)
  {
    Car c = (Car)sender;
    Console.WriteLine("Critical Message from {0}: {1}", c.PetName, e.msg);
  }
}
```

Исходный код. Проект `CarEventArgs` доступен в подкаталоге `Chapter_10`.

Обобщенный делегат `EventHandler<T>`

Учитывая, что очень многие специальные делегаты принимают экземпляр `object` в первом параметре и экземпляр класса, производного от `EventArgs`, во втором, предыдущий пример можно дополнительно упростить, за счет применения обобщенного типа `EventHandler<T>`, где `T` — специальный тип, производный от `EventArgs`. Рассмотрим следующую модификацию типа `Car` (обратите внимание, что определять специальный тип делегата больше не нужно):

```
public class Car
{
  public event EventHandler<CarEventArgs> Exploded;
  public event EventHandler<CarEventArgs> AboutToBlow;
  ...
}
```

После этого в методе `Main()` можно использовать тип `EventHandler<CarEventArgs>` везде, где ранее указывался `CarEngineHandler` (или снова применять групповое преобразование методов):

```
static void Main(string[] args)
{
  Console.WriteLine("***** Prim and Proper Events *****\n");

  // Создать объект Car обычным образом.
  Car c1 = new Car("SlugBug", 100, 10);

  // Зарегистрировать обработчики событий.
  c1.AboutToBlow += CarIsAlmostDoomed;
  c1.AboutToBlow += CarAboutToBlow;

  EventHandler<CarEventArgs> d = new EventHandler<CarEventArgs>(CarExploded);
  c1.Exploded += d;
  ...
}
```

К настоящему моменту вы видели основные аспекты работы с делегатами и событиями в C#. Хотя этого вполне достаточно для решения практически любых задач, связанных с обратными вызовами, в завершение главы мы рассмотрим несколько финальных упрощений, в частности анонимные методы и лямбда-выражения.

Исходный код. Проект `GenericCarEventArgs` доступен в подкаталоге `Chapter_10`.

Понятие анонимных методов C#

Как было показано ранее, когда вызывающий код желает прослушивать входящие события, он должен определить специальный метод в классе (или структуре), который соответствует сигнатуре ассоциированного делегата. Ниже приведен пример:

```
class Program
{
  static void Main(string[] args)
  {
    SomeType t = new SomeType();

    // Предположим, что SomeDeletage может указывать на методы,
    // которые не принимают аргументов и возвращают void.
    t.SomeEvent += new SomeDelegate(MyEventHandler);
  }

  // Обычно вызывается только объектом SomeDelegate.
  public static void MyEventHandler()
  {
    // Что-то делать при возникновении события.
  }
}
```

Однако если подумать, то такие методы, как `MyEventHandler()`, редко предназначены для вызова из любой другой части программы кроме делегата. С точки зрения продуктивности вручную определять отдельный метод для вызова объектом делегата несколько хлопотно (хотя и вполне допустимо).

Для решения этой проблемы событие можно ассоциировать прямо с блоком операторов кода во время регистрации события. Формально такой код называется *анонимным методом*. Чтобы проиллюстрировать синтаксис, давайте создадим метод `Main()`, кото-

рый обрабатывает события, отправленные из класса Car, с использованием анонимных методов вместо специальных именованных обработчиков событий:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Anonymous Methods *****\n");
    Car c1 = new Car("SlugBug", 100, 10);

    // Зарегистрировать обработчики событий как анонимные методы.
    c1.AboutToBlow += delegate
    {
      Console.WriteLine("Eek! Going too fast!");
    };

    c1.AboutToBlow += delegate(object sender, CarEventArgs e)
    {
      Console.WriteLine("Message from Car: {0}", e.msg);
    };

    c1.Exploded += delegate(object sender, CarEventArgs e)
    {
      Console.WriteLine("Fatal Message from Car: {0}", e.msg);
    };

    // В конце концов, этот код будет инициировать события.
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);

    Console.ReadLine();
  }
}
```

На заметку! После последней фигурной скобки в анонимном методе должна указываться точка с запятой. Если забыть сделать это, возникнет ошибка на этапе компиляции.

И снова легко заметить, что специальные статические обработчики событий вроде CarAboutToBlow() или CarExploded() в классе Program больше не определяются. Взамен с помощью синтаксиса += определяются встроенные неименованные (т.е. анонимные) методы, к которым вызывающий код будет обращаться во время обработки события. Базовый синтаксис анонимного метода представлен следующим псевдокодом:

```
class Program
{
  static void Main(string[] args)
  {
    НекоторыйТип t = new НекоторыйТип();
    t.НекотороеСобытие += delegate (дополнительноУказанныеАргументыДелегата)
    { /* операторы */ };
  }
}
```

Обратите внимание, что при обработке первого события AboutToBlow внутри предыдущего метода Main() аргументы, передаваемые из делегата, не указывались:

```
c1.AboutToBlow += delegate
{
  Console.WriteLine("Eek! Going too fast!");
};
```

Строго говоря, вы не обязаны принимать входные аргументы, отправленные специфическим событием. Но если вы хотите задействовать эти входные аргументы, то понадобится указать параметры, прототипированные типом делегата (как показано во второй обработке событий `AboutToBlow` и `Exploded`). Например:

```
c1.AboutToBlow += delegate(object sender, CarEventArgs e)
{
  Console.WriteLine("Critical Message from Car: {0}", e.msg);
};
```

Доступ к локальным переменным

Анонимные методы интересны тем, что способны обращаться к локальным переменным метода, где они определены. Формально такие переменные называются *внешними переменными* анонимного метода. Ниже перечислены важные моменты, касающиеся взаимодействия между областью действия анонимного метода и областью действия метода, в котором он определен.

- Анонимный метод не имеет доступа к параметрам `ref` и `out` определяющего метода.
- Анонимный метод не может иметь локальную переменную, имя которой совпадает с именем локальной переменной внешнего метода.
- Анонимный метод может обращаться к переменным экземпляра (или статическим переменным) из области действия внешнего класса.
- Анонимный метод может объявлять локальную переменную с тем же именем, что и у переменной-члена внешнего класса (локальные переменные имеют отдельную область действия и скрывают переменные-члены из внешнего класса).

Предположим, что в методе `Main()` определена локальная переменная по имени `aboutToBlowCounter` типа `int`. Внутри анонимных методов, которые обрабатывают событие `AboutToBlow`, мы увеличим значение этого счетчика на 1 и выведем результат на консоль перед завершением `Main()`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Anonymous Methods *****\n");
  int aboutToBlowCounter = 0;
  // Создать объект Car обычным образом.
  Car c1 = new Car("SlugBug", 100, 10);
  // Зарегистрировать обработчики событий как анонимные методы.
  c1.AboutToBlow += delegate
  {
    aboutToBlowCounter++;
    Console.WriteLine("Eek! Going too fast!");
  };
  c1.AboutToBlow += delegate(object sender, CarEventArgs e)
  {
    aboutToBlowCounter++;
    Console.WriteLine("Critical Message from Car: {0}", e.msg);
  };
  // В конце концов, это будет инициировать события.
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);
  Console.WriteLine("AboutToBlow event was fired {0} times.", aboutToBlowCounter);
  Console.ReadLine();
}
```

После запуска модифицированного метода `Main()` вы обнаружите, что финальный вывод `Console.WriteLine()` сообщает о том, что событие `AboutToBlow` было инициировано два раза.

Исходный код. Проект `AnonymousMethods` доступен в подкаталоге `Chapter_10`.

Понятие лямбда-выражений

Чтобы завершить знакомство с архитектурой событий .NET, необходимо исследовать *лямбда-выражения*. Как объяснялось ранее в этой главе, язык C# поддерживает возможность обработки событий "встроенным образом", позволяя назначать блок операторов кода прямо событию с применением анонимных методов вместо построения отдельного метода, который должен вызываться делегатом. Лямбда-выражения — это всего лишь лаконичный способ записи анонимных методов, который в конечном итоге упрощает работу с типами делегатов .NET.

В целях подготовки фундамента для изучения лямбда-выражений создадим новый проект консольного приложения по имени `SimpleLambdaExpressions`. Для начала взгляните на метод `FindAll()` обобщенного класса `List<T>`. Данный метод может вызываться, когда нужно извлечь подмножество элементов из коллекции, и обладает следующим прототипом:

```
// Метод класса System.Collections.Generic.List<T>.
public List<T> FindAll(Predicate<T> match)
```

Как видите, метод `FindAll()` возвращает новый объект `List<T>`, который представляет подмножество данных. Также обратите внимание, что единственным параметром `FindAll()` является обобщенный делегат типа `System.Predicate<T>`. Этот тип делегата может указывать на любой метод, возвращающий `bool` и принимающий единственный параметр:

```
// Этот делегат используется методом FindAll()
// для извлечения подмножества.
public delegate bool Predicate<T>(T obj);
```

Когда вызывается `FindAll()`, каждый элемент в `List<T>` передается методу, указанному объектом `Predicate<T>`. Реализация упомянутого метода будет выполнять некоторые вычисления для проверки соответствия элемента данных указанному критерию, возвращая в результате `true` или `false`. Если метод возвращает `true`, то текущий элемент будет добавлен в новый объект `List<T>`, представляющий интересующее подмножество.

Прежде чем мы посмотрим, как лямбда-выражения могут упростить работу с методом `FindAll()`, давайте решим эту задачу длинным способом, используя объекты делегатов непосредственно. Добавим в класс `Program` метод (`TraditionalDelegateSyntax()`), который взаимодействует с типом `System.Predicate<T>` для обнаружения четных чисел в списке `List<T>` целочисленных значений:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Lambdas *****\n");
    TraditionalDelegateSyntax();
    Console.ReadLine();
  }
```

```
static void TraditionalDelegateSyntax()
{
  // Создать список целочисленных значений.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // Вызвать FindAll() с применением традиционного синтаксиса делегатов.
  Predicate<int> callback = IsEvenNumber;
  List<int> evenNumbers = list.FindAll(callback);

  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}

// Цель для делегата Predicate<>.
static bool IsEvenNumber(int i)
{
  // Это четное число?
  return (i % 2) == 0;
}
}
```

Здесь мы имеем метод (`IsEvenNumber()`), который отвечает за проверку входного целочисленного параметра на предмет четности или нечетности с применением операции получения остатка от деления C# (`%`). Запуск приложения приводит к выводу на консоль чисел 20, 4, 8 и 44.

Хотя такой традиционный подход к работе с делегатами ведет себя ожидаемым образом, метод `IsEvenNumber()` вызывается только при ограниченных обстоятельствах — в частности, когда вызывается `FindAll()`, который возлагает на нас обязанность по полному определению метода. Если бы взамен использовался анонимный метод, то код стал бы значительно чище. Рассмотрим следующий новый метод в классе `Program`:

```
static void AnonymousMethodSyntax()
{
  // Создать список целочисленных значений.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // Теперь использовать анонимный метод.
  List<int> evenNumbers = list.FindAll(delegate(int i)
    { return (i % 2) == 0; } );
  // Вывести четные числа.
  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

В этом случае вместо прямого создания объекта делегата `Predicate<T>` и последующего написания отдельного метода у вас есть возможность встроить метод как анонимный. Несмотря на шаг в правильном направлении, вам по-прежнему придется применять ключевое слово `delegate` (или строго типизированный класс `Predicate<T>`) и обеспечивать точное соответствие списка параметров:

```
List<int> evenNumbers = list.FindAll(
  delegate(int i)
  {
    return (i % 2) == 0;
  }
);
```

Для еще большего упрощения вызова метода `FindAll()` могут использоваться *лямбда-выражения*. Во время применения синтаксиса лямбда-выражения вообще не приходится иметь дело с лежащим в основе объектом делегата. Взгляните на показанный далее новый метод в классе `Program`:

```
static void LambdaExpressionSyntax()
{
  // Создать список целочисленных значений.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // Теперь использовать лямбда-выражение C#.
  List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);

  // Вывести четные числа.
  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

Обратите внимание на довольно странный оператор кода, передаваемый методу `FindAll()`, который на самом деле и представляет собой лямбда-выражение. В этой версии примера нет вообще никаких следов делегата `Predicate<T>` (или ключевого слова `delegate`, если на то пошло). Вместо них указано только лямбда-выражение:

```
i => (i % 2) == 0
```

Перед разбором синтаксиса запомните, что лямбда-выражения могут использоваться везде, где должен применяться анонимный метод или строго типизированный делегат (обычно с гораздо меньшим клавиатурным набором). "За кулисами" компилятор C# транслирует лямбда-выражение в стандартный анонимный метод, использующий тип делегата `Predicate<T>` (в чем можно удостовериться с помощью утилиты `ildasm.exe` или `reflector.exe`). Скажем, следующий оператор кода:

```
// Это лямбда-выражение...
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

компилируется в приблизительно такой код C#:

```
// ...становится следующим анонимным методом.
List<int> evenNumbers = list.FindAll(delegate (int i)
{
  return (i % 2) == 0;
});
```

Анализ лямбда-выражения

Лямбда-выражение начинается со списка параметров, за которым следует лексема => (лексема C# для лямбда-операции позаимствована из *лямбда-вычислений*), а за ней — набор операторов (или одиночный оператор), который будет обрабатывать пере-

даваемые аргументы. На самом высоком уровне лямбда-выражение можно представить следующим образом:

```
АргументыДляОбработки => ОбрабатывающиеОператоры
```

То, что находится внутри метода `LambdaExpressionSyntax()`, понимается так:

```
// i - список параметров.
// (i % 2) == 0 - набор операторов для обработки i.
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

Параметры лямбда-выражения могут быть явно или неявно типизированными. В настоящий момент тип данных, представляющий параметр `i` (целочисленное значение), определяется неявно. Компилятор в состоянии понять, что `i` является целочисленным значением, на основе области действия всего лямбда-выражения и лежащего в основе делегата. Тем не менее, определять тип каждого параметра в лямбда-выражении можно также и явно, помещая тип данных и имя переменной в пару круглых скобок, как показано ниже:

```
// Теперь установим тип параметров явно.
List<int> evenNumbers = list.FindAll((int i) => (i % 2) == 0);
```

Как вы уже видели, если лямбда-выражение имеет одиночный неявно типизированный параметр, то круглые скобки в списке параметров могут быть опущены. Если вы желаете соблюдать согласованность относительно применения параметров лямбда-выражений, то можете *всегда* заключать в скобки список параметров:

```
List<int> evenNumbers = list.FindAll((i) => (i % 2) == 0);
```

Наконец, обратите внимание, что в текущий момент выражение не заключено в круглые скобки (естественно, вычисление остатка от деления помещено в скобки, чтобы гарантировать его выполнение перед проверкой на равенство). В лямбда-выражениях разрешено заключать операторы в круглые скобки:

```
// Поместить в скобки и выражение.
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

Теперь, когда вы ознакомились с разными способами построения лямбда-выражения, как его можно читать в понятных человеку терминах? Оставив чистую математику в стороне, можно привести следующее объяснение:

```
// Список параметров (в этом случае единственное целочисленное
// значение по имени i) будет обработан выражением (i % 2) == 0.
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

Обработка аргументов внутри множества операторов

Первое рассмотренное лямбда-выражение включало единственный оператор, который в итоге вычислялся в булевское значение. Однако, как вы знаете, многие цели делегатов должны выполнять несколько операторов кода. По этой причине C# позволяет строить лямбда-выражения, состоящие из множества блоков операторов. Когда выражение должно обрабатывать параметры, используя несколько строк кода, для операторов понадобится обозначить область действия с помощью фигурных скобок. Взгляните на приведенную ниже модификацию метода `LambdaExpressionSyntax()`:

```
static void LambdaExpressionSyntax()
{
  // Создать список целочисленных значений.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
```

```
  // Обработать каждый аргумент внутри группы операторов кода.
  List<int> evenNumbers = list.FindAll((i) =>
  {
    Console.WriteLine("value of i is currently: {0}", i);
    bool isEven = ((i % 2) == 0);
    return isEven;
  });
  // Вывести четные числа.
  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

В данном случае список параметров (опять состоящий из единственного целочисленного значения i) обрабатывается набором операторов кода. Помимо вызова метода `Console.WriteLine()` оператор вычисления остатка от деления разбит на два оператора для повышения читабельности. Предположим, что каждый из рассмотренных выше методов вызывается в `Main()`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Lambdas *****\n");
  TraditionalDelegateSyntax();
  AnonymousMethodSyntax();
  Console.WriteLine();
  LambdaExpressionSyntax();
  Console.ReadLine();
}
```

Запуск приложения дает следующий вывод:

```
***** Fun with Lambdas *****
Here are your even numbers:
20     4      8      44
Here are your even numbers:
20     4      8      44
value of i is currently: 20
value of i is currently: 1
value of i is currently: 4
value of i is currently: 8
value of i is currently: 9
value of i is currently: 44
Here are your even numbers:
20     4      8      44
```

Исходный код. Проект `SimpleLambdaExpressions` доступен в подкаталоге `Chapter_10`.

Лямбда-выражения с несколькими параметрами и без параметров

Показанные ранее лямбда-выражения обрабатывали единственный параметр. Тем не менее, это вовсе не обязательно, т.к. лямбда-выражения могут обрабатывать множество аргументов (или ни одного). Для демонстрации первого сценария создадим проект консольного приложения по имени `LambdaExpressionsMultipleParams` с показанной ниже версией класса `SimpleMath`:

```
public class SimpleMath
{
  public delegate void MathMessage(string msg, int result);
  private MathMessage mmDelegate;

  public void SetMathHandler(MathMessage target)
  {mmDelegate = target; }

  public void Add(int x, int y)
  {
    mmDelegate?.Invoke("Adding has completed!", x + y);
  }
}
```

Обратите внимание, что делегат `MathMessage` ожидает два параметра. Чтобы представить их в виде лямбда-выражения, метод `Main()` может быть реализован так:

```
static void Main(string[] args)
{
  // Зарегистрировать делегат как лямбда-выражение.
  SimpleMath m = new SimpleMath();
  m.SetMathHandler((msg, result) =>
    {Console.WriteLine("Message: {0}, Result: {1}", msg, result);});

  // Это приведет к выполнению лямбда-выражения.
  m.Add(10, 10);
  Console.ReadLine();
}
```

Здесь задействовано выведение типа, поскольку для простоты два параметра не были строго типизированы. Однако метод `SetMathHandler()` можно было бы вызвать следующим образом:

```
m.SetMathHandler((string msg, int result) =>
  {Console.WriteLine("Message: {0}, Result: {1}", msg, result);});
```

И, наконец, если лямбда-выражение применяется для взаимодействия с делегатом, который вообще не принимает параметров, то это можно сделать, указав в качестве параметра пару пустых круглых скобок. Таким образом, предполагая, что определен такой тип делегата:

```
public delegate string VerySimpleDelegate();
```

вот как можно было бы обработать результат вызова:

```
// Выводит на консоль строку "Enjoy your string!".
VerySimpleDelegate d = new VerySimpleDelegate( () => {return "Enjoy your string!";} );
Console.WriteLine(d());
```

Исходный код. Проект `LambdaExpressionsMultipleParams` доступен в подкаталоге `Chapter_10`.

Модернизация примера `PrimAndProperCarEvents` с использованием лямбда-выражений

С учетом того, что основной целью лямбда-выражений является предоставление способа ясного и компактного определения анонимных методов (косвенно упрощая работу с делегатами), давайте модернизируем проект `CarEventArgs`, созданный ранее в этой главе. Ниже приведена упрощенная версия метода `Main()`, в которой для перехва-

та всех событий, поступающих от объекта Car, применяется синтаксис лямбда-выражений (вместо простых делегатов):

```
static void Main(string[] args)
{
  Console.WriteLine("***** More Fun with Lambdas *****\n");

  // Создать объект Car обычным образом.
  Car c1 = new Car("SlugBug", 100, 10);

  // Привязаться к событиям с помощью лямбда-выражений.
  c1.AboutToBlow += (sender, e) => { Console.WriteLine(e.msg);};
  c1.Exploded += (sender, e) => { Console.WriteLine(e.msg); };

  // Увеличить скорость (это инициирует события).
  Console.WriteLine("\n***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);

  Console.ReadLine();
}
```

Лямбда-выражения и реализация членов с единственным оператором

Последний аспект, связанный с лямбда-операцией C#, заключается в том, что в версии .NET 4.6 для упрощения некоторых (но не всех) реализаций членов теперь допускается использовать операцию =>. В частности, если есть метод или свойство (в дополнение к специальной операции или процедуре преобразования, как показано в главе 11), реализация которого состоит в точности из одной строки кода, то определять область действия посредством фигурных скобок необязательно. Вместо этого можно задействовать лямбда-операцию.

Рассмотрим предыдущий пример кода, где производится привязка обработчиков к событиям AboutToBlow и Exploded. Обратите внимание на определение с помощью фигурных скобок области действия, которая внутри содержит вызов метода Console.WriteLine(). При желании можно было бы поступить так:

```
c1.AboutToBlow += (sender, e) => Console.WriteLine(e.msg);
c1.Exploded += (sender, e) => Console.WriteLine(e.msg);
```

Тем не менее, имейте в виду, что этот новый сокращенный синтаксис может применяться где угодно, даже когда код не имеет никакого отношения к делегатам или событиям. Таким образом, например, если вы строите элементарный класс для сложения двух чисел, то можете написать следующий код:

```
class SimpleMath
{
  public int Add(int x, int y)
  {
    return x + y;
  }

  public void PrintSum(int x, int y)
  {
    Console.WriteLine(x + y);
  }
}
```

В качестве альтернативы теперь код может выглядеть так:

```
class SimpleMath
{
  public int Add(int x, int y) =>  x + y;
  public void PrintSum(int x, int y) => Console.WriteLine(x + y);
}
```

В идеале к этому моменту вы должны уловить суть лямбда-выражений и понимать, что они предлагают "функциональный способ" работы с анонимными методами и типами делегатов. Хотя привыкание к лямбда-операции (=>) может занять некоторое время, просто запомните, что лямбда-выражение сводится к следующей формуле:

АргументыДляОбработки => ОбрабатывающиеОператоры

Или, если операция => используется для реализации члена типа с единственным оператором, то это будет выглядеть так:

ЧленТипа => ЕдинственныйОператорКода

Следует отметить, что лямбда-выражения широко задействованы также в модели программирования LINQ, помогая упрощать кодирование. Исследование LINQ начинается в главе 12.

Исходный код. Проект `CarEventsWithLambdas` доступен в подкаталоге `Chapter_10`.

Резюме

В этой главе вы получили представление о нескольких способах организации двустороннего взаимодействия для множества объектов. Во-первых, было рассмотрено ключевое слово `delegate`, которое применяется для косвенного конструирования класса, производного от `System.MulticastDelegate`. Вы узнали, что объект делегата поддерживает список методов для вызова тогда, когда ему об этом будет указано. Такие вызовы могут выполняться синхронно (с использованием метода `Invoke()`) или асинхронно (посредством методов `BeginInvoke()` и `EndInvoke()`). Асинхронная природа типов делегатов .NET будет обсуждаться в главе 19.

Во-вторых, вы ознакомились с ключевым словом `event`, которое в сочетании с типом делегата может упростить процесс отправки уведомлений ожидающим объектам. Как можно заметить в результирующем коде CIL, модель событий .NET отображается на скрытые обращения к типам `System.Delegate/System.MulticastDelegate`. В этом отношении ключевое слово `event` является совершенно необязательным, т.к. оно просто позволяет сэкономить на наборе кода. Кроме того, вы видели, что `null`-условная операция C# 6.0 упрощает безопасное инициирование событий для любой заинтересованной стороны.

В-третьих, в главе также рассматривалось средство языка C#, которое называется *анонимными методами*. С помощью такой синтаксической конструкции можно явно ассоциировать блок операторов кода с заданным событием. Как было показано, анонимные методы вполне могут игнорировать параметры, переданные событием, и имеют доступ к "внешним переменным" определяющего их метода. Вы также освоили упрощенный подход к регистрации событий с применением *групповых преобразований методов*.

И, наконец, в завершение главы мы взглянули на лямбда-операцию (=>) языка C#. Как было показано, этот синтаксис представляет собой сокращенный способ для записи анонимных методов, когда набор аргументов может быть передан на обработку группе операторов. Любой метод внутри платформы .NET, который принимает объект делегата в качестве аргумента, может быть заменен связанным лямбда-выражением, что обычно несколько упрощает кодовую базу.

ГЛАВА 11

Расширенные средства языка C#

В этой главе рассматриваются некоторые более сложные синтаксические конструкции языка программирования C#. Сначала вы узнаете, как реализовать и применять *метод индексатора*. Данный механизм C# позволяет строить специальные типы, которые обеспечивают доступ к внутренним элементам с использованием синтаксиса, подобного синтаксису массивов. Затем вы научитесь перегружать разнообразные операции (+, –, <, > и т.д.) и создавать специальные процедуры явного и неявного преобразования типов (а также ознакомитесь с причинами, по которым они могут понадобиться).

Далее обсуждаются темы, которые особенно полезны при работе с API-интерфейсами LINQ (хотя они применимы и за рамками контекста LINQ): расширяющие методы и анонимные типы.

В завершение главы вы узнаете, каким образом создавать контекст "небезопасного" кода, чтобы напрямую манипулировать неуправляемыми указателями. Хотя использование указателей в приложениях C# — довольно редкое явление, понимание того, как это делается, может пригодиться при определенных обстоятельствах, предусматривающих сложные сценарии взаимодействия.

Понятие методов индексаторов

Программистам хорошо знаком процесс доступа к индивидуальным элементам, содержащимся внутри простого массива, с применением операции индекса ([]). Вот пример:

```
static void Main(string[] args)
{
  // Организовать цикл по аргументам командной строки
  // с использованием операции индекса.
  for(int i = 0; i < args.Length; i++)
    Console.WriteLine("Args: {0}", args[i]);
  // Объявить локальный массив целочисленных значений.
  int[] myInts = { 10, 9, 100, 432, 9874};
  // Применить операцию индекса для доступа к каждому элементу.
  for(int j = 0; j < myInts.Length; j++)
    Console.WriteLine("Index {0}  = {1} ", j,  myInts[j]);
  Console.ReadLine();
}
```

Приведенный код не должен выглядеть как что-то совершенно новое. В языке C# предлагается возможность проектирования специальных классов и структур, которые

могут индексироваться подобно стандартному массиву, за счет определения *метода индексатора*. Это конкретное языковое средство наиболее полезно при создании специальных классов коллекций (обобщенных или необобщенных).

Прежде чем выяснять, каким образом реализуется специальный индексатор, давайте начнем с его исследования в действии. Предположим, что к специальному типу `PersonCollection`, разработанному в главе 9 (в проекте `IssuesWithNonGenericCollections`), добавлена поддержка метода индексатора. Хотя сам индексатор пока не добавлен, посмотрим на его использование внутри нового проекта консольного приложения по имени `SimpleIndexer`:

```
// Индексаторы позволяют обращаться к элементам в стиле массива.
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Indexers *****\n");

    PersonCollection myPeople = new PersonCollection();

    // Добавить объекты с применением синтаксиса индексатора.
    myPeople[0] = new Person("Homer", "Simpson", 40);
    myPeople[1] = new Person("Marge", "Simpson", 38);
    myPeople[2] = new Person("Lisa", "Simpson", 9);
    myPeople[3] = new Person("Bart", "Simpson", 7);
    myPeople[4] = new Person("Maggie", "Simpson", 2);

    // Получить и отобразить элементы с использованием индексатора.
    for (int i = 0; i < myPeople.Count; i++)
    {
      Console.WriteLine("Person number: {0}", i);          // номер лица
      Console.WriteLine("Name: {0} {1}",                   // имя и фамилия
        myPeople[i].FirstName, myPeople[i].LastName);
      Console.WriteLine("Age: {0}", myPeople[i].Age);      // возраст
      Console.WriteLine();
    }
  }
}
```

Как видите, индексаторы позволяют манипулировать внутренней коллекцией подобъектов подобно стандартному массиву. Но тут возникает серьезный вопрос: каким образом сконфигурировать класс `PersonCollection` (или любой другой класс либо структуру) для поддержки этой функциональности? Индексатор представлен как слегка видоизмененное определение свойства C#. В своей простейшей форме индексатор создается с применением синтаксиса `this[]`. Ниже показано необходимое обновление класса `PersonCollection`:

```
// Добавим индексатор к существующему определению класса.
public class PersonCollection : IEnumerable
{
  private ArrayList arPeople = new ArrayList();

  // Специальный индексатор для этого класса.
  public Person this[int index]
  {
    get { return (Person)arPeople[index]; }
    set { arPeople.Insert(index, value); }
  }
  ...
}
```

Если не считать использование ключевого слова this, индексатор похож на объявление любого другого свойства C#. Например, роль области get заключается в возвращении корректного объекта вызывающему коду. Здесь мы делаем это, делегируя запрос к индексатору объекта ArrayList, т.к. данный класс также поддерживает индексатор. Область set отвечает за добавление новых объектов Person, что достигается вызовом метода Insert() объекта ArrayList.

Индексаторы являются еще одной формой "синтаксического сахара", учитывая, что ту же самую функциональность можно получить с применением "нормальных" открытых методов, таких как AddPerson() или GetPerson(). Тем не менее, поддержка методов индексаторов в специальных типах коллекций обеспечивает их хорошую интеграцию с инфраструктурой библиотек базовых классов .NET.

Несмотря на то что создание методов индексаторов является довольно-таки обычным явлением при построении специальных коллекций, не забывайте, что обобщенные типы предлагают такую функциональность в готовом виде. В следующем методе используется обобщенный список List<T> объектов Person. Обратите внимание, что индексатор List<T> можно просто применять непосредственно. Например:

```
static void UseGenericListOfPeople()
{
  List<Person> myPeople = new List<Person>();
  myPeople.Add(new Person("Lisa", "Simpson", 9));
  myPeople.Add(new Person("Bart", "Simpson", 7));

  // Изменить первый объект лица с помощью индексатора.
  myPeople[0] = new Person("Maggie", "Simpson", 2);

  // Теперь получить и отобразить каждый элемент, используя индексатор.
  for (int i = 0; i < myPeople.Count; i++)
  {
    Console.WriteLine("Person number: {0}", i);
    Console.WriteLine("Name: {0} {1}", myPeople[i].FirstName,
      myPeople[i].LastName);
    Console.WriteLine("Age: {0}", myPeople[i].Age);
    Console.WriteLine();
  }
}
```

Исходный код. Проект SimpleIndexer доступен в подкаталоге Chapter_11.

Индексация данных с использованием строковых значений

В текущей версии класса PersonCollection определен индексатор, позволяющий вызывающему коду идентифицировать подэлементы с применением числовых значений. Однако вы должны понимать, что это не относится к требованиям метода индексатора. Предположим, что вы предпочитаете хранить объекты Person, используя тип System.Collections.Generic.Dictionary<TKey, TValue> вместо ArrayList. Поскольку типы Dictionary разрешают доступ к содержащимся внутри них типам с применением ключа (такого как фамилия лица), индексатор можно было бы определить следующим образом:

```
public class PersonCollection : IEnumerable
{
  private Dictionary<string, Person> listPeople =
    new Dictionary<string, Person>();
```

```
  // Этот индексатор возвращает объект лица на основе строкового индекса.
  public Person this[string name]
  {
    get { return (Person)listPeople[name]; }
    set { listPeople[name] = value; }
  }
  public void ClearPeople()
  { listPeople.Clear(); }

  public int Count
  { get { return listPeople.Count; } }

  IEnumerator IEnumerable.GetEnumerator()
  { return listPeople.GetEnumerator(); }
}
```

Теперь вызывающий код может взаимодействовать с содержащимися внутри объектами `Person`, как показано ниже:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Indexers *****\n");

  PersonCollection myPeople = new PersonCollection();

  myPeople["Homer"] = new Person("Homer", "Simpson", 40);
  myPeople["Marge"] = new Person("Marge", "Simpson", 38);

  // Получить объект лица Homer и вывести данные.
  Person homer = myPeople["Homer"];
  Console.WriteLine(homer.ToString());

  Console.ReadLine();
}
```

Опять-таки, если бы вы использовали обобщенный тип `Dictionary<TKey, TValue>` напрямую, то получили бы функциональность метода индексатора в готовом виде без построения специального необобщенного класса, поддерживающего строковый индексатор. Тем не менее, имейте в виду, что тип данных любого индексатора будет основан на том, как поддерживающий тип коллекции позволяет вызывающему коду извлекать подэлементы.

Исходный код. Проект `StringIndexer` доступен в подкаталоге `Chapter_11`.

Перегрузка методов индексаторов

Методы индексаторов могут быть перегружены в отдельном классе или структуре. Таким образом, если имеет смысл предоставить вызывающему коду возможность доступа к подэлементам с применением числового индекса *или* строкового значения, то в одном типе можно определить несколько индексаторов. Например, в ADO.NET (встроенный API-интерфейс .NET для доступа к базам данных) класс `DataSet` поддерживает свойство по имени `Tables`, которое возвращает строго типизированную коллекцию `DataTableCollection`. В свою очередь, в `DataTableCollection` определены *три* индексатора для получения и установки объектов `DataTable` — по порядковой позиции, по дружественному строковому имени и по строковому имени с дополнительным пространством имен:

```
public sealed class DataTableCollection : InternalDataCollectionBase
{
  ...
  // Перегруженные индексаторы.
  public DataTable this[int index] { get; }
  public DataTable this[string name] { get; }
  public DataTable this[string name, string tableNamespace] { get; }
}
```

Поддержка методов индексаторов обычна для типов в библиотеках базовых классов. Поэтому даже если текущий проект не требует построения специальных индексаторов для классов и структур, помните, что многие типы уже поддерживают этот синтаксис.

Многомерные индексаторы

Вы можете также создавать метод индексатора, который принимает несколько параметров. Предположим, что у вас есть специальная коллекция, хранящая подэлементы в двумерном массиве. В таком случае метод индексатора можно определить следующим образом:

```
public class SomeContainer
{
  private int[,] my2DintArray = new int[10, 10];

  public int this[int row, int column]
  { /* получить или установить значение в двумерном массиве */ }
}
```

Если только вы не строите высокоспециализированный класс коллекций, то вряд ли особо нуждаетесь в создании многомерного индексатора. И снова пример ADO.NET демонстрирует, насколько полезной может быть эта конструкция. Класс `DataTable` в ADO.NET — это по существу коллекция строк и столбцов, похожая на миллиметровку или общую электронную таблицу Microsoft Excel.

Хотя объекты `DataTable` обычно наполняются без вашего участия с использованием связанного "адаптера данных", в приведенном ниже коде показано, как вручную создать находящийся в памяти объект `DataTable`, который содержит три столбца (для имени, фамилии и возраста каждой записи). Обратите внимание на то, как после добавления одной строки в `DataTable` с помощью многомерного индексатора производится обращение ко всем столбцам первой (и единственной) строки. (Если вы собираетесь следовать примеру, то импортируйте в файл кода пространство имен `System.Data`.)

```
static void MultiIndexerWithDataTable()
{
  // Создать простой объект DataTable с тремя столбцами.
  DataTable myTable = new DataTable();
  myTable.Columns.Add(new DataColumn("FirstName"));
  myTable.Columns.Add(new DataColumn("LastName"));
  myTable.Columns.Add(new DataColumn("Age"));

  // Добавить строку в таблицу.
  myTable.Rows.Add("Mel", "Appleby", 60);

  // Использовать многомерный индексатор для вывода деталей первой строки.
  Console.WriteLine("First Name: {0}", myTable.Rows[0][0]);
  Console.WriteLine("Last Name: {0}", myTable.Rows[0][1]);
  Console.WriteLine("Age : {0}", myTable.Rows[0][2]);
}
```

Начиная с главы 21, мы продолжим рассмотрение ADO.NET, так что не пугайтесь, если что-то в приведенном выше коде выглядит незнакомым. Этот пример просто иллюстрирует, что методы индексаторов способны поддерживать множество измерений, а при правильном применении могут упростить взаимодействие с подобъектами, содержащимися в специальных коллекциях.

Определения индексаторов в интерфейсных типах

Индексаторы могут быть определены в заданном типе интерфейса .NET, чтобы позволить поддерживающим типам предоставлять специальные реализации. Ниже показан простой пример интерфейса, который определяет протокол для получения строковых объектов с использованием числового индексатора:

```
public interface IStringContainer
{
  string this[int index] { get; set; }
}
```

При таком определении интерфейса любой класс или структура, которые его реализуют, должны поддерживать индексатор с чтением/записью, манипулирующий подэлементами с применением числового значения. Вот частичная реализация примера класса:

```
class SomeClass : IStringContainer
{
  private List<string> myStrings = new List<string>();

  public string this[int index]
  {
    get { return myStrings[index]; }
    set { myStrings.Insert(index, value); }
  }
}
```

На этом первая крупная тема настоящей главы завершена. А теперь давайте займемся исследованием языкового средства, которое позволяет строить специальные классы и структуры, уникальным образом реагирующие на внутренние операции C#. Итак, займемся концепцией *перегрузки операций*.

Понятие перегрузки операций

Как и любой язык программирования, C# имеет заготовленный набор лексем, используемых для выполнения базовых операций над встроенными типами. Например, вы знаете, что операция + может применяться к двум целым числам, чтобы получить большее целое число:

```
// Операция + с целыми числами.
int a = 100;
int b = 240;
int c = a + b; // с теперь имеет значение 340
```

Опять-таки, здесь нет ничего нового, но задумывались ли вы когда-нибудь о том, что одну и ту же операцию + разрешено использовать с большинством встроенных типов данных C#? Скажем, взгляните на следующий код:

```
// Операция + со строками.
string s1 = "Hello";
string s2 = " world!";
string s3 = s1 + s2;  // s3 теперь имеет значение "Hello world!"
```

В сущности, операция + функционирует специфическим образом на основе типа предоставленных данных (в этом случае строкового или целочисленного). Когда операция + применяется к числовым типам, в результате выполняется суммирование операндов, а когда к строковым типам — то конкатенация строк.

Язык C# дает возможность строить специальные классы и структуры, которые также уникально реагируют на один и тот же набор базовых лексем (вроде операции +). Хотя не каждая операция C# может быть перегружена, перегрузку допускают многие операции (табл. 11.1).

Таблица 11.1. Возможность перегрузки операций C#

Операция C#	Возможность перегрузки
+, -, !, ~, ++, --, true, false	Эти унарные операции могут быть перегружены
+, -, *, /, %, &, |, ^, <<, >>	Эти бинарные операции могут быть перегружены
==, !=, <, >, <=, >=	Эти операции сравнения могут быть перегружены. Язык C# требует совместной перегрузки "похожих" операций (т.е. < и >, <= и >=, == и !=)
[]	Операция [] не может быть перегружена. Однако, как было показано ранее в главе, ту же самую функциональность обеспечивает индексатор
()	Операция () не может быть перегружена. Тем не менее, как вы увидите далее в главе, ту же самую функциональность предоставляют специальные методы преобразования
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Сокращенные операции присваивания не могут быть перегружены; однако вы получаете их автоматически, когда перегружаете соответствующие бинарные операции

Перегрузка бинарных операций

Чтобы проиллюстрировать процесс перегрузки бинарных операций, рассмотрим приведенный ниже простой класс Point, который определен в новом проекте консольного приложения по имени OverloadedOps:

```
// Простой будничный класс C#.
public class Point
{
  public int X {get; set;}
  public int Y {get; set;}

  public Point(int xPos, int yPos)
  {
    X = xPos;
    Y = yPos;
  }

  public override string ToString()
  {
    return string.Format("[{0}, {1}]", this.X, this.Y);
  }
}
```

Если рассуждать логически, то суммирование объектов Point имеет смысл. Например, сложение двух переменных Point должно давать новый объект Point с про-

суммированными значениями свойств X и Y. Конечно, полезно также и вычитать один объект Point из другого. В идеале желательно иметь возможность записи примерно такого кода:

```
// Сложение и вычитание двух точек?
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Overloaded Operators *****\n");

  // Создать две точки.
  Point ptOne = new Point(100, 100);
  Point ptTwo = new Point(40, 40);
  Console.WriteLine("ptOne = {0}", ptOne);
  Console.WriteLine("ptTwo = {0}", ptTwo);

  // Сложить две точки, чтобы получить большую?
  Console.WriteLine("ptOne + ptTwo: {0} ", ptOne + ptTwo);

  // Вычесть одну точку из другой, чтобы получить меньшую?
  Console.WriteLine("ptOne - ptTwo: {0} ", ptOne - ptTwo);
  Console.ReadLine();
}
```

Тем не менее, при существующем виде класса Point вы получите ошибки на этапе компиляции, потому что типу Point не известно, как реагировать на операции + и -. Для оснащения специального типа способностью уникально реагировать на встроенные операции язык C# предлагает ключевое слово operator, которое может использоваться только в сочетании с ключевым словом static. При перегрузке бинарной операции (такой как + и -) вы чаще всего будете передавать два аргумента того же типа, что и класс, определяющий операцию (Point в этом примере):

```
// Более интеллектуальный тип Point.
public class Point
{
  ...
  // Перегруженная операция +.
  public static Point operator + (Point p1, Point p2)
  {
    return new Point(p1.X + p2.X, p1.Y + p2.Y);
  }

  // Перегруженная операция -.
  public static Point operator - (Point p1, Point p2)
  {
    return new Point(p1.X - p2.X, p1.Y - p2.Y);
  }
}
```

Логика, положенная в основу операции +, предусматривает просто возвращение нового объекта Point, основанного на сложении соответствующих полей входных параметров Point. Таким образом, когда вы пишете p1 + p2, "за кулисами" происходит следующий скрытый вызов статического метода operator +:

```
// Псевдокод: Point p3 = Point.operator+ (p1, p2)
Point p3 = p1 + p2;
```

Подобным образом выражение p1 - p2 отображается так:

```
// Псевдокод: Point p4 = Point.operator- (p1, p2)
Point p4 = p1 - p2;
```

После такой модификации типа `Point` программа скомпилируется и появится возможность сложения и вычитания объектов `Point`, что подтверждает представленный далее вывод:

```
ptOne = [100, 100]
ptTwo = [40, 40]
ptOne + ptTwo: [140, 140]
ptOne - ptTwo: [60, 60]
```

При перегрузке бинарной операции передавать ей два параметра того же самого типа не обязательно. Если это имеет смысл, то один из аргументов может относиться к другому типу. Например, ниже показана перегруженная операция +, которая позволяет вызывающему коду получить новый объект `Point` на основе числовой коррекции:

```
public class Point
{
  ...
  public static Point operator + (Point p1, int change)
  {
    return new Point(p1.X + change, p1.Y + change);
  }
  public static Point operator + (int change, Point p1)
  {
    return new Point(p1.X + change, p1.Y + change);
  }
}
```

Обратите внимание, что если вы хотите передавать аргументы в любом порядке, то потребуется реализовать *обе* версии метода (т.е. нельзя просто определить один из методов и рассчитывать, что компилятор автоматически будет поддерживать другой). Теперь эти новые версии операции + можно применять следующим образом:

```
// Выводит [110, 110].
Point biggerPoint = ptOne + 10;
Console.WriteLine("ptOne + 10 = {0}", biggerPoint);
// Выводит [120, 120].
Console.WriteLine("10 + biggerPoint = {0}", 10 + biggerPoint);
Console.WriteLine();
```

А как насчет операций += и -=?

Если до перехода на C# вы имели дело с языком C++, то вас может удивить отсутствие возможности перегрузки операций сокращенного присваивания (+=, -+ и т.д.). Не беспокойтесь. В C# операции сокращенного присваивания автоматически эмулируются при перегрузке связанных бинарных операций. Таким образом, если в классе `Point` уже перегружены операции + и -, то можно написать следующий код:

```
// Перегрузка бинарных операций автоматически обеспечивает
// перегрузку сокращенных операций.
static void Main(string[] args)
{
  ...
  // Операция += перегружена автоматически.
  Point ptThree = new Point(90, 5);
  Console.WriteLine("ptThree = {0}", ptThree);
  Console.WriteLine("ptThree += ptTwo: {0}", ptThree += ptTwo);
```

```
  // Операция -= перегружена автоматически.
  Point ptFour = new Point(0, 500);
  Console.WriteLine("ptFour = {0}", ptFour);
  Console.WriteLine("ptFour -= ptThree: {0}", ptFour -= ptThree);
  Console.ReadLine();
}
```

Перегрузка унарных операций

В C# также разрешено перегружать унарные операции, такие как ++ и --. При перегрузке унарной операции также должно использоваться ключевое слово `static` с ключевым словом `operator`, но в этом случае просто передается единственный параметр того же типа, что и класс или структура, где операция определена. Например, дополним реализацию `Point` такими перегруженными операциями:

```
public class Point
{
  ...
  // Добавить 1 к значениям X/Y входного объекта Point.
  public static Point operator ++(Point p1)
  {
    return new Point(p1.X+1, p1.Y+1);
  }
  // Вычесть 1 из значений X/Y входного объекта Point.
  public static Point operator --(Point p1)
  {
    return new Point(p1.X-1, p1.Y-1);
  }
}
```

В результате появляется возможность инкрементировать и декрементировать значения `X` и `Y` класса `Point`:

```
static void Main(string[] args)
{
  ...
  // Применение унарных операций ++ и -- к объекту Point.
  Point ptFive = new Point(1, 1);
  Console.WriteLine("++ptFive = {0}", ++ptFive);     // [2, 2]
  Console.WriteLine("--ptFive = {0}", --ptFive);     // [1, 1]
  // Применение тех же операций в виде постфиксного инкремента/декремента.
  Point ptSix = new Point(20, 20);
  Console.WriteLine("ptSix++ = {0}", ptSix++);       // [20, 20]
  Console.WriteLine("ptSix-- = {0}", ptSix--);       // [21, 21]
  Console.ReadLine();
}
```

В предыдущем примере кода специальные операции ++ и -- применяются двумя разными способами. В C++ допускается перегружать операции префиксного и постфиксного инкремента/декремента по отдельности. В C# это невозможно. Однако возвращаемое значение инкремента/декремента автоматически обрабатывается корректно (т.е. для перегруженной операции ++ выражение `pt++` дает значение немодифицированного объекта, в то время как `++pt` имеет новое значение, устанавливаемое перед использованием в выражении).

Перегрузка операций эквивалентности

Как упоминалось в главе 6, метод `System.Object.Equals()` может быть перегружен для выполнения сравнений на основе значений (а не ссылок) между ссылочными типами. Если вы решили переопределить `Equals()` (часто вместе со связанным методом `System.Object.GetHashCode()`), то легко переопределите и операции проверки эквивалентности (== и !=). Взгляните на обновленный тип `Point`:

```
// В этой версии типа Point также перегружены операции == и !=.
public class Point
{
  ...
  public override bool Equals(object o)
  {
    return o.ToString() == this.ToString();
  }
  public override int GetHashCode()
  {
    return this.ToString().GetHashCode();
  }
  // Теперь перегрузить операции == и !=.
  public static bool operator ==(Point p1, Point p2)
  {
    return p1.Equals(p2);
  }
  public static bool operator !=(Point p1, Point p2)
  {
    return !p1.Equals(p2);
  }
}
```

Обратите внимание, что для выполнения всей работы в реализациях операций == и != просто вызывается перегруженный метод `Equals()`. Теперь класс `Point` можно применять следующим образом:

```
// Использование перегруженных операций эквивалентности.
static void Main(string[] args)
{
  ...
  Console.WriteLine("ptOne == ptTwo : {0}", ptOne == ptTwo);
  Console.WriteLine("ptOne != ptTwo : {0}", ptOne != ptTwo);
  Console.ReadLine();
}
```

Как видите, сравнение двух объектов с использованием хорошо знакомых операций == и != выглядит намного интуитивно понятнее, чем вызов метода `Object.Equals()`. При перегрузке операций эквивалентности для определенного класса имейте в виду, что C# требует, чтобы в случае перегрузки операции == *обязательно* перегружалась также и операция != (компилятор напомнит, если вы забудете это сделать).

Перегрузка операций сравнения

В главе 8 было показано, каким образом реализовывать интерфейс `IComparable` для сравнения двух похожих объектов. В действительности для того же самого класса можно также перегрузить операции сравнения (<, >, <= и >=). Как и в случае операций эквивалентности, язык C# требует, чтобы при перегрузке операции < обязательно пере-

гружалась также операция >. Если класс Point перегружает эти операции сравнения, то пользователь объекта может сравнивать объекты Point:

```
// Использование перегруженных операций < и >.
static void Main(string[] args)
{
  ...
  Console.WriteLine("ptOne < ptTwo : {0}", ptOne < ptTwo);
  Console.WriteLine("ptOne > ptTwo : {0}", ptOne > ptTwo);
  Console.ReadLine();
}
```

Когда интерфейс IComparable (или, что еще лучше, его обобщенный эквивалент) реализован, перегрузка операций сравнения становится тривиальной. Вот модифицированное определение класса:

```
// Объекты Point также можно сравнивать посредством операций сравнения.
public class Point : IComparable<Point>
{
  ...
  public int CompareTo(Point other)
  {
    if (this.X > other.X && this.Y > other.Y)
      return 1;
    if (this.X < other.X && this.Y < other.Y)
      return -1;
    else
      return 0;
  }

  public static bool operator <(Point p1, Point p2)
  { return (p1.CompareTo(p2) < 0); }

  public static bool operator >(Point p1, Point p2)
  { return (p1.CompareTo(p2) > 0); }

  public static bool operator <=(Point p1, Point p2)
  { return (p1.CompareTo(p2) <= 0); }

  public static bool operator >=(Point p1, Point p2)
  { return (p1.CompareTo(p2) >= 0); }
}
```

Финальные соображения относительно перегрузки операций

Как уже было указано, язык C# предоставляет возможность построения типов, которые могут уникальным образом реагировать на разнообразные встроенные хорошо известные операции. Перед добавлением поддержки такого поведения в классы вы должны удостовериться в том, что операции, которые планируется перегружать, имеют какой-нибудь смысл в реальности.

Например, пусть перегружена операция умножения для класса MiniVan, представляющего минивэн. Что по своей сути будет означать перемножение двух объектов MiniVan? В этом нет особого смысла. На самом деле коллеги по команде даже могут быть озадачены, когда увидят следующее применение класса MiniVan:

```
// Что?! Понять это непросто...
MiniVan newVan = myVan * yourVan;
```

Перегрузка операций обычно полезна только при построении атомарных типов данных. Текст, точки, прямоугольники, функции и шестиугольники — подходящие канди-

даты на перегрузку операций, но люди, менеджеры, автомобили, подключения к базе данных и веб-страницы — нет. В качестве эмпирического правила запомните, что если перегруженная операция *затрудняет* понимание пользователем функциональности типа, то не перегружайте ее. Используйте эту возможность с умом.

Исходный код. Проект `OverloadedOps` доступен в подкаталоге `Chapter_11`.

Понятие специальных преобразований типов

Давайте теперь обратимся к теме, тесно связанной с перегрузкой операций — специальные преобразования типов. Чтобы заложить фундамент для последующего обсуждения, кратко вспомним понятие явных и неявных преобразований между числовыми данными и связанными типами классов.

Повторение: числовые преобразования

В терминах встроенных числовых типов (`sbyte`, `int`, `float` и т.д.) *явное преобразование* требуется, когда вы пытаетесь сохранить большее значение в контейнере меньшего размера, т.к. подобное действие может привести к потере данных. По существу тем самым вы сообщаете компилятору, что отдаете себе отчет в том, что делаете. В противоположность этому *неявное преобразование* происходит автоматически, когда вы пытаетесь поместить меньший тип в больший целевой тип, что не должно вызвать потерю данных:

```
static void Main()
{
  int a = 123;
  long b = a;             // Неявное преобразование из int в long.
  int c = (int) b;        // Явное преобразование из long в int.
}
```

Повторение: преобразования между связанными типами классов

Как было показано в главе 6, типы классов могут быть связаны классическим наследованием (отношение "является"). В таком случае процесс преобразования C# позволяет осуществлять приведение вверх и вниз по иерархии классов. Например, производный класс всегда может быть неявно приведен к базовому классу. Тем не менее, если вы хотите сохранить объект базового класса в переменной производного класса, то должны выполнить явное приведение:

```
// Два связанных типа классов.
class Base{}
class Derived : Base{}

class Program
{
  static void Main(string[] args)
  {
    // Неявное приведение производного класса к базовому.
    Base myBaseType;
    myBaseType = new Derived();

    // Для сохранения ссылки на базовый класс в переменной
    // производного класса требуется явное преобразование.
    Derived myDerivedType = (Derived)myBaseType;
  }
}
```

Продемонстрированное явное приведение работает из-за того, что классы Base и Derived связаны классическим наследованием. Однако что если есть два типа классов в *разных иерархиях* без общего предка (кроме System.Object), которые требуют преобразований? Учитывая, что они не связаны классическим наследованием, типичные операции приведения здесь не помогут (и вдобавок компилятор сообщит об ошибке).

В качестве связанного замечания обратимся к типам значений (структурам). Предположим, что имеются две структуры .NET с именами Square и Rectangle. Поскольку они не могут задействовать классическое наследование (т.к. всегда запечатаны), не существует естественного способа выполнить приведение между этими по внешнему виду связанными типами.

Несмотря на то что можно было бы создать в структурах вспомогательные методы (наподобие Rectangle.ToSquare()), язык C# позволяет строить специальные процедуры преобразования, которые дают типам возможность реагировать на операцию приведения (). Следовательно, если корректно сконфигурировать структуры, то для явного преобразования между ними можно будет применять такой синтаксис:

```
// Преобразовать Rectangle в Square!
Rectangle rect;
rect.Width = 3;
rect.Height = 10;
Square sq = (Square)rect;
```

Создание специальных процедур преобразования

Начнем с создания нового проекта консольного приложения по имени CustomConversions. В языке C# предусмотрены два ключевых слова, explicit и implicit, которые можно использовать для управления тем, как типы должны реагировать на попытку преобразования. Предположим, что есть следующие определения структур:

```
public struct Rectangle
{
  public int Width {get; set;}
  public int Height {get; set;}

  public Rectangle(int w, int h) : this()
  {
    Width = w; Height = h;
  }

  public void Draw()
  {
    for (int i = 0; i < Height; i++)
    {
      for (int j = 0; j < Width; j++)
      {
        Console.Write("*");
      }
      Console.WriteLine();
    }
  }

  public override string ToString()
  {
    return string.Format("[Width = {0}; Height = {1}]",
      Width, Height);
  }
}
```

```
public struct Square
{
  public int Length {get; set;}
  public Square(int l) : this()
  {
    Length = l;
  }
  public void Draw()
  {
    for (int i = 0; i < Length; i++)
    {
      for (int j = 0; j < Length; j++)
      {
        Console.Write("*");
      }
      Console.WriteLine();
    }
  }
  public override string ToString()
  { return string.Format("[Length = {0}]", Length); }
  // Rectangle можно явно преобразовывать в Square.
  public static explicit operator Square(Rectangle r)
  {
    Square s = new Square();
    s.Length = r.Height;
    return s;
  }
}
```

На заметку! Вы заметите, что в конструкторах `Square` и `Rectangle` производится явное связывание в цепочку со стандартным конструктором. Дело в том, что когда в структуре применяется синтаксис автоматических свойств (как в рассматриваемом случае), внутри всех специальных конструкторов должен явно вызываться стандартный конструктор для инициализации закрытых поддерживающих полей. (Например, если структуры имеют любые дополнительные поля/свойства, то данный стандартный конструктор будет их инициализировать стандартными значениями.) Да, это необычное правило C#, но ведь настоящая глава посвящена сложным темам.

Обратите внимание, что в этой версии типа `Square` определена явная операция преобразования. Подобно перегрузке операций процедуры преобразования используют ключевое слово `operator` в сочетании с ключевым словом `explicit` или `implicit` и должны быть определены как `static`. Входным параметром является сущность, *из* которой выполняется преобразование, а типом операции — сущность, *в* которую оно производится.

В данном случае предположение заключается в том, что квадрат (будучи геометрической фигурой с четырьмя сторонами равной длины) может быть получен из высоты прямоугольника. Таким образом, вот как преобразовать `Rectangle` в `Square`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Conversions *****\n");
  // Создать экземпляр Rectangle.
  Rectangle r = new Rectangle(15, 4);
  Console.WriteLine(r.ToString());
  r.Draw();
```

```
  Console.WriteLine();
  // Преобразовать r в Square на основе высоты Rectangle.
  Square s = (Square)r;
  Console.WriteLine(s.ToString());
  s.Draw();
  Console.ReadLine();
}
```

Ниже показан вывод:

```
***** Fun with Conversions *****
[Width = 15; Height = 4]
***************
***************
***************
***************
[Length = 4]
****
****
****
****
```

Хотя преобразование `Rectangle` в `Square` в пределах одной и той же области действия может быть не особенно полезным, взгляните на следующий метод, который спроектирован для приема параметров типа `Square`:

```
// Этот метод требует параметр типа Square.
static void DrawSquare(Square sq)
{
  Console.WriteLine(sq.ToString());
  sq.Draw();
}
```

Благодаря наличию операции явного преобразования в типе `Square` этому методу на обработку можно передавать типы `Rectangle`, применяя явное приведение:

```
static void Main(string[] args)
{
  ...
  // Преобразовать Rectangle в Square для вызова метода.
  Rectangle rect = new Rectangle(10, 5);
  DrawSquare((Square)rect);
  Console.ReadLine();
}
```

Дополнительные явные преобразования для типа `Square`

Теперь, когда экземпляры `Rectangle` можно явно преобразовывать в экземпляры `Square`, давайте рассмотрим несколько дополнительных явных преобразований. Учитывая, что квадрат симметричен по всем сторонам, полезно предусмотреть процедуру преобразования, которая позволит вызывающему коду привести целочисленный тип к типу `Square` (который, естественно, будет иметь длину стороны, равную переданному целочисленному значению). Подобным же образом, что если вы захотите модифицировать `Square` так, чтобы вызывающий код мог выполнять приведение *из* `Square` в `int`? Вот как выглядит логика вызова:

```
static void Main(string[] args)
{
  ...
  // Преобразование int в Square.
  Square sq2 = (Square)90;
  Console.WriteLine("sq2 = {0}", sq2);

  // Преобразование Square в int.
  int side = (int)sq2;
  Console.WriteLine("Side length of sq2 = {0}", side);
  Console.ReadLine();
}
```

Ниже показаны изменения, внесенные в структуру `Square`:

```
public struct Square
{
  ...
  public static explicit operator Square(int sideLength)
  {
    Square newSq = new Square();
    newSq.Length = sideLength;
    return newSq;
  }

  public static explicit operator int (Square s)
  { return s.Length; }
}
```

По правде говоря, преобразование `Square` в `int` может показаться не слишком интуитивно понятной (или полезной) операцией (все же, скорее всего, вы просто передадите нужные значения конструктору). Тем не менее, это указывает на важный факт, касающийся процедур специальных преобразований: компилятор не беспокоится о том, из чего и во что происходит преобразование, до тех пор, пока вы пишете синтаксически корректный код.

Таким образом, как и с перегрузкой операций, *возможность* создания операции явного приведения для заданного типа вовсе не означает *необходимость* ее создания. Обычно этот прием будет наиболее полезным при создании типов структур .NET, учитывая, что они не могут принимать участие в классическом наследовании (где приведение обеспечивается автоматически).

Определение процедур неявного преобразования

До сих пор мы создавали различные специальные операции *явного* преобразования. Но что насчет следующего *неявного* преобразования?

```
static void Main(string[] args)
{
  ...
  Square s3 = new Square();
  s3.Length = 83;

  // Попытка сделать неявное приведение?
  Rectangle rect2 = s3;

  Console.ReadLine();
}
```

Данный код не скомпилируется, т.к. вы не предоставили процедуру неявного преобразования для типа `Rectangle`. Ловушка здесь вот в чем: определять одновременно функции явного и неявного преобразования не разрешено, если они не различаются по типу возвращаемого значения или по списку параметров. Это может показаться ограничением; однако вторая ловушка связана с тем, что когда тип определяет процедуру *неявного* преобразования, то вызывающий код вполне законно может использовать синтаксис *явного* приведения!

Запутались? Чтобы прояснить ситуацию, давайте добавим к структуре `Rectangle` процедуру неявного преобразования с применением ключевого слова `implicit` (обратите внимание, что в показанном ниже коде предполагается, что ширина результирующего прямоугольника вычисляется умножением стороны квадрата на 2):

```
public struct Rectangle
{
  ...
  public static implicit operator Rectangle(Square s)
  {
    Rectangle r = new Rectangle();
    r.Height = s.Length;
    // Предположим, что ширина нового квадрата будет равна (Length x 2).
    r.Width = s.Length * 2;
    return r;
  }
}
```

После такой модификации можно выполнять преобразование между типами:

```
static void Main(string[] args)
{
  ...
  // Неявное преобразование работает!
  Square s3 = new Square();
  s3.Length= 7;
  Rectangle rect2 = s3;
  Console.WriteLine("rect2 = {0}", rect2);
  // Синтаксис явного преобразования также работает!
  Square s4 = new Square();
  s4.Length = 3;
  Rectangle rect3 = (Rectangle)s4;
  Console.WriteLine("rect3 = {0}", rect3);
  Console.ReadLine();
}
```

На этом обзор определения операций специального преобразования завершен. Как и с перегруженными операциями, помните о том, что данный фрагмент синтаксиса представляет собой просто сокращенное обозначение для "нормальных" функций-членов, и в этом смысле всегда необязателен. Тем не менее, в случае правильного использования специальные структуры могут применяться более естественным образом, поскольку будут трактоваться как настоящие типы классов, связанные наследованием.

Исходный код. Проект `CustomConversions` доступен в подкаталоге `Chapter_11`.

Понятие расширяющих методов

В версии .NET 3.5 появилась концепция *расширяющих методов*, которая позволила добавлять новые методы или свойства к классу либо структуре, не модифицируя исходный тип непосредственно. Когда это может оказаться удобным? Рассмотрим следующие ситуации.

Для начала пусть имеется класс, находящийся в производстве. Со временем выясняется, что этот класс должен поддерживать несколько новых членов. Изменение текущего определения класса напрямую сопряжено с риском нарушения обратной совместимости со старыми кодовыми базами, использующими его, т.к. они могут не скомпилироваться с последним улучшенным определением класса. Один из способов обеспечения обратной совместимости предусматривает создание нового класса, производного от существующего, но тогда придется сопровождать два класса. Как все мы знаем, сопровождение кода является самой скучной частью деятельности разработчика программного обеспечения.

Теперь представим другую ситуацию. Предположим, что имеется структура (или, может быть, запечатанный класс), и необходимо добавить новые члены, чтобы получить полиморфное поведение в рамках системы. Поскольку структуры и запечатанные классы не могут быть расширены, единственный выбор заключается в том, чтобы добавить желаемые члены к типу, снова рискуя нарушить обратную совместимость!

За счет применения расширяющих методов появляется возможность модифицировать типы, не создавая подклассов и не изменяя код типа напрямую. По правде говоря, данный прием в действительности является иллюзией. Новая функциональность предлагается типом, только если в текущем проекте будет присутствовать ссылки на эти расширяющие методы.

Определение расширяющих методов

Первое ограничение, связанное с расширяющими методами, состоит в том, что они должны быть определены внутри статического класса (см. главу 5), и потому каждый расширяющий метод должен объявляться с ключевым словом `static`. Вторая проблема в том, что все расширяющие методы помечаются как таковые посредством ключевого слова `this` в качестве модификатора первого (и только первого) параметра заданного метода. Параметр, помеченный с помощью `this`, представляет расширяемый элемент.

В целях иллюстрации создадим новый проект консольного приложения под названием `ExtensionMethods`. Теперь предположим, что создается класс по имени `MyExtensions`, в котором определены два расширяющих метода. Первый расширяющий метод позволяет объекту любого типа взаимодействовать с новым методом `DisplayDefiningAssembly()`, который использует типы из пространства имен `System.Reflection` для отображения имени сборки, содержащей этот тип.

На заметку! API-интерфейс рефлексии формально рассматривается в главе 15. Если эта тема для вас нова, просто запомните, что рефлексия позволяет исследовать структуру сборок, типов и членов типов во время выполнения.

Второй расширяющий метод по имени `ReverseDigits()` позволяет любому значению типа `int` получить новую версию самого себя с обратным порядком следования цифр. Например, если целочисленное значение `1234` вызывает `ReverseDigits()`, то в результате возвратится `4321`. Взгляните на следующую реализацию класса (не забудьте импортировать пространство имен `System.Reflection`):

```
static class MyExtensions
{
  // Этот метод позволяет объекту любого типа
  // отобразить сборку, в которой он определен.
  public static void DisplayDefiningAssembly(this object obj)
  {
    Console.WriteLine("{0} lives here: => {1}\n", obj.GetType().Name,
      Assembly.GetAssembly(obj.GetType()).GetName().Name);
  }

  // Этот метод позволяет любому целочисленному значению изменить порядок
  // следования десятичных цифр на обратный. Например, для 56 возвратится 65.
  public static int ReverseDigits(this int i)
  {
    // Транслировать int в string и затем получить все его символы.
    char[] digits = i.ToString().ToCharArray();

    // Изменить порядок следования элементов массива.
    Array.Reverse(digits);

    // Поместить обратно в строку.
    string newDigits = new string(digits);

    // Возвратить модифицированную строку как int.
    return int.Parse(newDigits);
  }
}
```

Снова обратите внимание на то, что первый параметр каждого расширяющего метода снабжен ключевым словом `this`, находящимся перед определением типа параметра. Первый параметр расширяющего метода всегда представляет расширяемый тип. Учитывая, что `DisplayDefiningAssembly()` был прототипирован для расширения `System.Object`, этот новый член теперь имеется в каждом типе, поскольку `Object` является родительским для всех типов платформы .NET. Однако метод `ReverseDigits()` прототипирован для расширения только целочисленных типов, поэтому если к нему обращается какое-то другое значение, то возникнет ошибка на этапе компиляции.

На заметку! Запомните, что каждый расширяющий метод может иметь множество параметров, но *только* первый параметр разрешено помечать посредством `this`. Дополнительные параметры будут трактоваться как нормальные входные параметры, применяемые методом.

Вызов расширяющих методов

Располагая этими расширяющими методами, рассмотрим следующий метод `Main()` который их использует с разнообразными типами из библиотек базовых классов:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Extension Methods *****\n");
  // В int появилась новая отличительная черта!
  int myInt = 12345678;
  myInt.DisplayDefiningAssembly();

  // То же и в DataSet!
  System.Data.DataSet d = new System.Data.DataSet();
  d.DisplayDefiningAssembly();

  // И в SoundPlayer!
  System.Media.SoundPlayer sp = new System.Media.SoundPlayer();
  sp.DisplayDefiningAssembly();
```

```
    // Использовать новую функциональность int.
    Console.WriteLine("Value of myInt: {0}", myInt);
    Console.WriteLine("Reversed digits of myInt: {0}", myInt.ReverseDigits());
    Console.ReadLine();
}
```

Ниже показан вывод:

```
***** Fun with Extension Methods *****
Int32 lives here: => mscorlib
DataSet lives here: => System.Data
SoundPlayer lives here: => System
Value of myInt: 12345678
Reversed digits of myInt: 87654321
```

Импортирование расширяющих методов

Когда определяется класс, содержащий расширяющие методы, вне всяких сомнений он будет принадлежать какому-то пространству имен .NET. Если это пространство имен отличается от пространства имен, где расширяющие методы применяются, то понадобится использовать ключевое слово `using` языка C#. Оно позволяет файлу кода иметь доступ ко всем расширяющим методам интересующего типа. Об этом важно помнить, потому что если не импортировать явно корректное пространство имен, то в таком файле кода C# расширяющие методы окажутся недоступными.

Хотя на первый взгляд может показаться, что расширяющие методы глобальны по своей природе, на самом деле они ограничены пространствами имен, где они определены, или пространствами имен, которые их импортируют. Таким образом, если вы поместите класс `MyExtensions` в пространство имен `MyExtensionMethods`, как показано ниже:

```
namespace MyExtensionMethods
{
  static class MyExtensions
  {
    ...
  }
}
```

то другим пространствам имен в проекте придется явно импортировать `MyExtensionMethods` для получения расширяющих методов, определенных вашим классом.

На заметку! Обычная практика предусматривает изоляцию расширяющих методов не только в отдельном пространстве имен .NET, но также в отдельной библиотеке классов. В таком случае новые приложения могут обращаться к расширениям путем явной ссылки на подходящую библиотеку и импортирования пространства имен. В главе 14 будут представлены детали построения и применения специальных библиотек классов .NET.

Поддержка расширяющих методов средством IntelliSense

Учитывая то, что расширяющие методы не определены буквально в расширяемом типе, при чтении кода есть шанс запутаться. Например, предположим, что вы импортировали пространство имен, в котором определено несколько расширяющих методов, написанных кем-то из команды разработчиков. При написании своего кода вы можете создать переменную расширенного типа, применить операцию точки и обнаружить десятки новых методов, которые не являются членами исходного определения класса!

К счастью, средство IntelliSense в Visual Studio помечает все расширяющие методы, как показано на рис. 11.1.

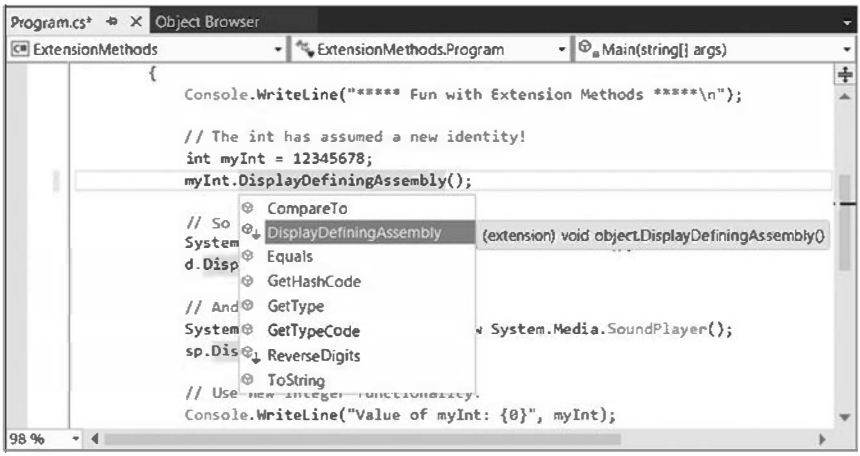


Рис. 11.1. Отображение расширяющих методов в IntelliSense

Любой метод, помеченный подобным образом, определен за пределами исходного определения класса как расширяющий метод.

Исходный код. Проект `ExtensionMethods` доступен в подкаталоге `Chapter_11`.

Расширение типов, реализующих специфичные интерфейсы

К этому моменту вы видели, как расширять классы (и косвенно структуры, которые следуют тому же синтаксису) новой функциональностью посредством расширяющих методов. Также есть возможность определить расширяющий метод, который способен расширять только класс или структуру, реализующую корректный интерфейс. Например, можно было бы заявить следующее: если класс или структура реализует интерфейс `IEnumerable<T>`, то этот тип получит новые члены. Разумеется, вполне допустимо требовать, чтобы тип поддерживал вообще любой интерфейс, включая ваши специальные интерфейсы.

Для примера создадим новый проект консольного приложения по имени `InterfaceExtensions`. Цель здесь заключается в том, чтобы добавить новый метод к любому типу, который реализует интерфейс `IEnumerable`; это охватывает все массивы и многие классы необобщенных коллекций (вспомните из главы 8, что обобщенный интерфейс `IEnumerable<T>` расширяет необобщенный интерфейс `IEnumerable`). Добавьте в проект следующий расширяющий класс:

```
static class AnnoyingExtensions
{
  public static void PrintDataAndBeep(this System.Collections.IEnumerable
iterator)
  {
    foreach (var item in iterator)
    {
      Console.WriteLine(item);
      Console.Beep();
    }
  }
}
```

Поскольку метод PrintDataAndBeep() может использоваться любым классом или структурой, реализующей интерфейс IEnumerable, мы можем протестировать его с помощью такого метода Main():

```
static void Main( string[] args )
{
  Console.WriteLine("***** Extending Interface Compatible Types *****\n");

  // System.Array реализует IEnumerable!
  string[] data = { "Wow", "this", "is", "sort", "of", "annoying",
                    "but", "in", "a", "weird", "way", "fun!"};
  data.PrintDataAndBeep();
  Console.WriteLine();

  // List<T> реализует IEnumerable!
  List<int> myInts = new List<int>() {10, 15, 20};
  myInts.PrintDataAndBeep();

  Console.ReadLine();
}
```

На этом исследование расширяющих методов C# завершено. Помните, что это конкретное языковое средство полезно, когда вам необходимо расширить функциональность типа, но вы не хотите создавать подклассы (или не можете, если тип запечатан) в целях обеспечения полиморфизма. Как вы увидите позже, расширяющие методы играют ключевую роль в API-интерфейсах LINQ. На самом деле вы узнаете, что в API-интерфейсах LINQ одним из самых часто расширяемых элементов является класс или структура, реализующая обобщенную версию интерфейса IEnumerable.

Исходный код. Проект InterfaceExtension доступен в подкаталоге Chapter_11.

Понятие анонимных типов

Программистам на объектно-ориентированных языках хорошо известны преимущества определения классов для представления состояния и функциональности заданного элемента, который требуется моделировать. Всякий раз, когда необходимо определить класс, который предназначен для многократного применения и предоставляет обширную функциональность через набор методов, событий, свойств и специальных конструкторов, устоявшейся практикой является создание нового класса C#.

Тем не менее, возникают и другие ситуации, когда желательно определять класс просто в целях моделирования набора инкапсулированных (и каким-то образом связанных) элементов данных безо всяких ассоциированных методов, событий или другой специализированной функциональности. Кроме того, что если такой тип должен использоваться только небольшим набором методов внутри программы? Было бы довольно утомительно строить полное определение класса вроде показанного ниже, если хорошо известно, что класс будет применяться только в нескольких местах. Чтобы подчеркнуть этот момент, вот примерный план того, что может понадобиться делать, когда нужно создать "простой" тип данных, который следует обычной семантике на основе значений:

```
class SomeClass
{
  // Определить набор закрытых переменных-членов...
  // Создать свойство для каждой закрытой переменной...
  // Переопределить метод ToString() для учета основных переменных-членов...
  // Переопределить методы GetHashCode() и Equals()
  // для работы с эквивалентностью на основе значений...
}
```

Как видите, задача не обязательно оказывается настолько простой. Вам потребуется не только написать большой объем кода, но еще и сопровождать дополнительный класс в системе. Для временных данных такого рода было бы удобно формировать специальный тип на лету. Например, пусть необходимо построить специальный метод, который принимает какой-то набор входных параметров. Эти параметры нужно использовать для создания нового типа данных, который будет применяться внутри области действия метода. Вдобавок желательно иметь возможность быстрого вывода данных с помощью метода `ToString()` и работы с другими членами `System.Object`. Все этого можно достичь с помощью синтаксиса анонимных типов.

Определение анонимного типа

Анонимный тип определяется с использованием ключевого слова `var` (глава 3) в сочетании с синтаксисом инициализации объектов (глава 5). Ключевое слово `var` должно применяться из-за того, что компилятор будет автоматически генерировать новое определение класса на этапе компиляции (причем имя этого класса никогда не встретится в коде C#). Синтаксис инициализации применяется для сообщения компилятору о необходимости создания в новом типе закрытых поддерживающих полей и (допускающих только чтение) свойств.

В целях иллюстрации создадим новый проект консольного приложения по имени `AnonymousTypes`. Затем добавим в класс `Program` показанный ниже метод, который формирует новый тип на лету, используя данные входных параметров:

```
static void BuildAnonType( string make, string color, int currSp )
{
  // Построить анонимный тип с применением входных аргументов.
  var car = new { Make = make, Color = color, Speed = currSp };

  // Обратите внимание, что теперь этот тип можно
  // использовать для получения данных свойств!
  Console.WriteLine("You have a {0} {1} going {2} MPH",
    car.Color, car.Make, car.Speed);

  // Анонимные типы имеют специальные реализации каждого
  // виртуального метода System.Object. Например:
  Console.WriteLine("ToString() == {0}", car.ToString());
}
```

Метод `BuildAnonType()` можно вызвать в `Main()` ожидаемым образом. Однако обратите внимание, что анонимный тип можно также создать с применением жестко закодированных значений:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Anonymous Types *****\n");

  // Создать анонимный тип, представляющий автомобиль.
  var myCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };

  // Вывести на консоль цвет и производителя.
  Console.WriteLine("My car is a {0} {1}.", myCar.Color, myCar.Make);

  // А теперь вызвать вспомогательный метод для построения
  // анонимного типа с указанием аргументов.
  BuildAnonType("BMW", "Black", 90);

  Console.ReadLine();
}
```

Итак, к настоящему моменту достаточно понимать, что анонимные типы позволяют быстро моделировать "форму" данных с очень малыми накладными расходами. Это всего лишь способ построения на лету нового типа данных, который поддерживает базовую инкапсуляцию через свойства и действует в соответствии с семантикой на основе значений. Чтобы уловить суть последнего утверждения, давайте посмотрим, каким образом компилятор C# строит анонимные типы на этапе компиляции, и в особенности — как он переопределяет члены `System.Object`.

Внутреннее представление анонимных типов

Все анонимные типы автоматически наследуются от `System.Object` и потому поддерживают все члены, предоставленные этим базовым классом. В результате можно вызывать метод `ToString()`, `GetHashCode()`, `Equals()` или `GetType()` на неявно типизированном объекте `myCar`. Предположим, что в классе `Program` определен следующий статический вспомогательный метод:

```
static void ReflectOverAnonymousType(object obj)
{
  Console.WriteLine("obj is an instance of: {0}", obj.GetType().Name);
  Console.WriteLine("Base class of {0} is {1}",
    obj.GetType().Name,
    obj.GetType().BaseType);
  Console.WriteLine("obj.ToString() == {0}", obj.ToString());
  Console.WriteLine("obj.GetHashCode() == {0}", obj.GetHashCode());
  Console.WriteLine();
}
```

Теперь вызовем этот метод в `Main()`, передав ему объект `myCar` в качестве параметра:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Anonymous Types *****\n");

  // Создать анонимный тип, представляющий автомобиль.
  var myCar = new {Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55};

  // Выполнить рефлексию того, что сгенерировал компилятор.
  ReflectOverAnonymousType(myCar);
  ...
  Console.ReadLine();
}
```

Вывод будет выглядеть приблизительно так:

```
***** Fun with Anonymous Types *****

obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() = { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() = -439083487
```

Прежде всего, в этом примере обратите внимание на то, что объект `myCar` имеет тип `<>f__AnonymousType0`3` (в вашем выводе имя типа может быть другим). Помните, что имя, назначаемое типу, полностью определяется компилятором и напрямую в коде C# недоступно.

Пожалуй, наиболее важно здесь то, что каждая пара "имя-значение", определенная с использованием синтаксиса инициализации объектов, отображается на идентично именованное свойство, доступное только для чтения, и соответствующее закрытое под-

держивающее поле, также допускающее только чтение. Приведенный ниже код C# примерно отражает сгенерированный компилятором класс, применяемый для представления объекта myCar (который можно просмотреть посредством утилиты ildasm.exe):

```
internal sealed class <>f__AnonymousType0<<Color>j__TPar,
  <Make>j__TPar, <CurrentSpeed>j__TPar>
{
  // Поля только для чтения.
  private readonly <Color>j__TPar <Color>i__Field;
  private readonly <CurrentSpeed>j__TPar <CurrentSpeed>i__Field;
  private readonly <Make>j__TPar <Make>i__Field;

  // Стандартный конструктор.
  public <>f__AnonymousType0(<Color>j__TPar Color,
    <Make>j__TPar Make, <CurrentSpeed>j__TPar CurrentSpeed);

  // Переопределенные методы.
  public override bool Equals(object value);
  public override int GetHashCode();
  public override string ToString();

  // Свойства только для чтения.
  public <Color>j__TPar Color { get; }
  public <CurrentSpeed>j__TPar CurrentSpeed { get; }
  public <Make>j__TPar Make { get; }
}
```

Реализация методов ToString() и GetHashCode()

Все анонимные типы автоматически являются производными от System.Object и предоставляют переопределенные версии методов Equals(), GetHashCode() и ToString(). Реализация ToString() просто строит строку из пар "имя-значение". Вот пример:

```
public override string ToString()
{
  StringBuilder builder = new StringBuilder();
  builder.Append("{ Color = ");
  builder.Append(this.<Color>i__Field);
  builder.Append(", Make = ");
  builder.Append(this.<Make>i__Field);
  builder.Append(", CurrentSpeed = ");
  builder.Append(this.<CurrentSpeed>i__Field);
  builder.Append(" }");
  return builder.ToString();
}
```

Реализация GetHashCode() вычисляет хеш-значение, используя каждую переменную-член анонимного типа в качестве входных данных для типа System.Collections.Generic.EqualityComparer<T>. С этой реализацией GetHashCode() два анонимных типа будут выдавать одинаковые хеш-значения тогда (и только тогда), когда они обладают одним и тем же набором свойств, которым присвоены те же самые значения. Благодаря такой реализации анонимные типы хорошо подходят для помещения внутрь контейнера Hashtable.

Семантика эквивалентности анонимных типов

Наряду с тем, что реализация переопределенных методов ToString() и GetHashCode() достаточно проста, вас может интересовать, как был реализован метод

Equals(). Например, если определены две переменные "анонимных автомобилей" с одинаковыми наборами пар "имя-значение", то должны ли эти переменные считаться эквивалентными? Чтобы увидеть результат такого сравнения, дополним класс Program следующим новым методом:

```
static void EqualityTest()
{
  // Создать два анонимных класса с идентичными наборами пар "имя-значение".
  var firstCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };
  var secondCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };

  // Считаются ли они эквивалентными, когда используется Equals()?
  if (firstCar.Equals(secondCar))
    Console.WriteLine("Same anonymous object!");
                    // Тот же самый анонимный объект
  else
    Console.WriteLine("Not the same anonymous object!");
                    // Не тот же самый анонимный объект

  // Можно ли проверить их эквивалентность с помощью операции ==?
  if (firstCar == secondCar)
    Console.WriteLine("Same anonymous object!");
  else
    Console.WriteLine("Not the same anonymous object!");

  // Имеют ли эти объекты в основе один и тот же тип?
  if (firstCar.GetType().Name == secondCar.GetType().Name)
    Console.WriteLine("We are both the same type!");
                    // Оба объекта имеют тот же самый тип
  else
    Console.WriteLine("We are different types!");
                    // Объекты относятся к разным типам

  // Отобразить все детали.
  Console.WriteLine();
  ReflectOverAnonymousType(firstCar);
  ReflectOverAnonymousType(secondCar);
}
```

В результате вызова этого метода внутри Main() получается (несколько неожиданный) вывод:

```
My car is a Bright Pink Saab.
You have a Black BMW going 90 MPH
ToString() == { Make = BMW, Color = Black, Speed = 90 }

Same anonymous object!
Not the same anonymous object!
We are both the same type!

obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() == { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() == -439083487

obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() == { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() == -439083487
```

Как видите, первая проверка, где вызывается Equals(), возвращает true, и потому на консоль выводится сообщение Same anonymous object! (тот же самый анонимный

объект). Причина в том, что сгенерированный компилятором метод `Equals()` при проверке эквивалентности применяет семантику на основе значений (т.е. проверяет значения каждого поля сравниваемых объектов).

Тем не менее, вторая проверка, в которой используется операция ==, приводит к выводу на консоль строки `Not the same anonymous object!` (не тот же самый анонимный объект), что на первый взгляд выглядит несколько нелогично. Такой результат обусловлен тем, что анонимные типы *не* получают перегруженных версий операций проверки равенства (== и !=). Поэтому при проверке эквивалентности объектов анонимных типов с применением операций равенства C# (вместо метода `Equals()`) проверяются *ссылки*, а не значения, поддерживаемые объектами.

Последнее, но не менее важное замечание: финальная проверка (где исследуется имя лежащего в основе типа) показывает, что объекты анонимных типов являются экземплярами одного и того же типа класса, сгенерированного компилятором (`<>f__AnonymousType0`3` в данном примере), т.к. `firstCar` и `secondCar` имеют одинаковые наборы свойств (`Color`, `Make` и `CurrentSpeed`).

Это иллюстрирует важный, но тонкий аспект: компилятор будет генерировать новое определение класса, только когда анонимный тип содержит *уникальные* имена свойств. Таким образом, если вы объявляете идентичные анонимные типы (в смысле с одинаковыми именами свойств) внутри сборки, то компилятор генерирует единственное определение анонимного типа.

Анонимные типы, содержащие другие анонимные типы

Разрешено создавать анонимные типы, которые состоят из других анонимных типов. Например, предположим, что требуется смоделировать заказ на приобретение, который хранит метку времени, цену и сведения о приобретаемом автомобиле. Вот новый (чуть более сложный) анонимный тип, представляющий такую сущность:

```
// Создать анонимный тип, состоящий из еще одного анонимного типа.
var purchaseItem = new {
  TimeBought = DateTime.Now,
  ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
  Price = 34.000};

ReflectOverAnonymousType(purchaseItem);
```

Сейчас вы уже должны понимать синтаксис, используемый для определения анонимных типов, но возможно все еще интересуетесь, где (и когда) применять это языковое средство. Выражаясь кратко, объявления анонимных типов следует использовать умеренно, обычно только когда применяется набор технологий LINQ (глава 12). С учетом описанных ниже многочисленных ограничений анонимных типов вы никогда не должны отказываться от использования строго типизированных классов и структур просто из-за того, что такое возможно.

- Отсутствует контроль над именами анонимных типов.
- Анонимные типы всегда расширяют `System.Object`.
- Поля и свойства анонимного типа всегда допускают только чтение.
- Анонимные типы не могут поддерживать события, специальные методы, специальные операции или специальные переопределения.
- Анонимные типы всегда неявно запечатаны.
- Экземпляры анонимных типов всегда создаются с применением стандартных конструкторов.

Однако при программировании с использованием набора технологий LINQ вы обнаружите, что во многих случаях этот синтаксис оказывается удобным, когда нужно быстро смоделировать общую *форму* сущности, а не ее функциональность.

Исходный код. Проект `AnonymousTypes` доступен в подкаталоге `Chapter_11`.

Работа с типами указателей

Последняя тема главы касается средства C#, которое в подавляющем большинстве проектов применяется реже всех остальных.

На заметку! В последующих примерах предполагается наличие у вас определенных навыков манипулирования указателями в C++. Если это не так, можете спокойно пропустить данную тему. В подавляющем большинстве приложений C# указатели не используются.

В главе 4 вы узнали, что платформа .NET определяет две крупных категории данных: типы значений и ссылочные типы. По правде говоря, в действительность есть еще и третья категория: *типы указателей*. Для работы с типами указателей доступны специфичные операции и ключевые слова (табл. 11.2), которые позволяют обойти схему управления памятью CLR и взять дело в свои руки.

Таблица 11.2. Операции и ключевые слова C#, связанные с указателями

Операция/ ключевое слово	Назначение
`*`	Эта операция применяется для создания переменной указателя (т.е. переменной, которая представляет непосредственное местоположение в памяти). Как и в языке C++, та же самая операция используется для разыменования указателя
`&`	Эта операция применяется для получения адреса переменной в памяти
`->`	Эта операция используется для доступа к полям типа, представленного указателем (небезопасная версия операции точки в C#)
`[]`	Эта операция (в небезопасном контексте) позволяет индексировать область памяти, на которую указывает переменная указателя (если вы программировали на C++, то вспомните о взаимодействии между переменной указателя и операцией `[]`)
`++, --`	В небезопасном контексте операции инкремента и декремента могут применяться к типам указателей
`+, -`	В небезопасном контексте операции сложения и вычитания могут применяться к типам указателей
`==, !=, <, >, <=, >=`	В небезопасном контексте операции сравнения и эквивалентности могут применяться к типам указателей
`stackalloc`	В небезопасном контексте ключевое слово `stackalloc` может использоваться для размещения массивов C# прямо в стеке
`fixed`	В небезопасном контексте ключевое слово `fixed` может применяться для временного закрепления переменной, чтобы впоследствии удалось найти ее адрес

Перед погружением в детали следует еще раз подчеркнуть, что вам очень *редко, если вообще когда-нибудь*, понадобится использовать типы указателей. Хотя C# позволяет опуститься на уровень манипуляций указателями, помните, что исполняющая среда .NET не имеет абсолютно никакого понятия о ваших намерениях. Соответственно, если вы неправильно управляете указателем, то сами и будете отвечать за последствия. С учетом этих предупреждений возникает вопрос: когда в принципе может понадобиться работать с типами указателей? Существуют две распространенных ситуации.

- Необходимо оптимизировать избранные части приложения, напрямую манипулируя памятью за рамками ее управления со стороны CLR.
- Требуется вызывать методы из DLL-библиотеки, написанной на C, или сервера COM, которые требуют передачи типов указателей в качестве параметров. Но даже в этом случае часто можно обойтись без применения типов указателей, отдав предпочтение типу `System.IntPtr` и членам типа `System.Runtime.InteropServices.Marshal`.

В случае если вы все же решили задействовать данное средство языка C#, то придется информировать компилятор C# (`csc.exe`) о своих намерениях, разрешив проекту поддерживать "небезопасный код". Чтобы сделать это в командной строке, при запуске компилятора укажите флаг `/unsafe`:

```
csc /unsafe *.cs
```

В среде Visual Studio перейдите на страницу свойств проекта и на вкладке Build (Сборка) отметьте флажок Allow Unsafe Code (Разрешить небезопасный код), как показано на рис. 11.2. Чтобы поэкспериментировать с типами указателей, создадим новый проект консольного приложения по имени `UnsafeCode` и разрешим небезопасный код.

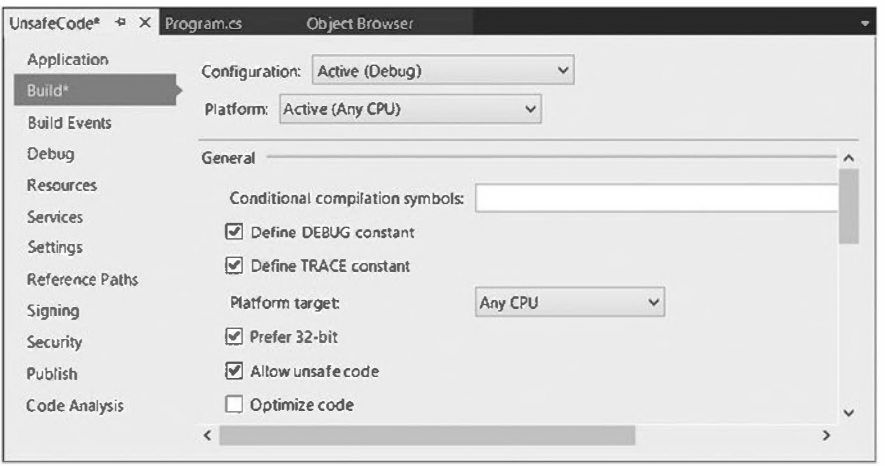


Рис. 11.2. Включение поддержки небезопасного кода в Visual Studio

Ключевое слово `unsafe`

Для работы с указателями в C# должен быть специально объявлен блок "небезопасного кода" с использованием ключевого слова `unsafe` (любой код, который не помечен ключевым словом `unsafe`, автоматически считается "безопасным"). Например, в следующем классе `Program` объявляется область небезопасного кода внутри метода `Main()`:

```
class Program
{
```

```
  static void Main(string[] args)
  {
    unsafe
    {
      // Здесь работаем с указателями!
    }
    // Здесь работа с указателями невозможна!
  }
}
```

В дополнение к объявлению области небезопасного кода внутри метода можно строить "небезопасные" структуры, классы, члены типов и параметры. Ниже приведено несколько примеров (типы Node и Node2 в текущем проекте определять не нужно):

```
// Вся эта структура является небезопасной и может
// использоваться только в небезопасном контексте.
unsafe struct Node
{
  public int Value;
  public Node* Left;
  public Node* Right;
}

// Эта структура безопасна, но члены Node2* - нет.
// Формально извне небезопасного контекста можно
// обращаться к Value, но не к Left и Right.
public struct Node2
{
  public int Value;

  // Эти члены доступны только в небезопасном контексте!
  public unsafe Node2* Left;
  public unsafe Node2* Right;
}
```

Методы (статические либо уровня экземпляра) также могут быть помечены как небезопасные. Предположим, что определенный статический метод будет использовать логику указателей. Чтобы обеспечить возможность вызова этого метода только из небезопасного контекста, его можно определить так:

```
unsafe static void SquareIntPointer(int* myIntPointer)
{
  // Возвести значение в квадрат просто для тестирования.
  *myIntPointer *= *myIntPointer;
}
```

Конфигурация метода требует, чтобы вызывающий код обращался к методу SquareIntPointer() следующим образом:

```
static void Main(string[] args)
{
  unsafe
  {
    int myInt = 10;

    // Нормально, мы находимся в небезопасном контексте.
    SquareIntPointer(&myInt);
    Console.WriteLine("myInt: {0}", myInt);
  }

  int myInt2 = 5;
```

```
  // Ошибка на этапе компиляции! Это должно делаться в небезопасном контексте!
  SquareIntPointer(&myInt2);
  Console.WriteLine("myInt: {0}", myInt2);
}
```

Если вы не хотите вынуждать вызывающий код помещать такой вызов внутрь небезопасного контекста, то можете пометить весь метод `Main()` ключевым словом `unsafe`. В этом случае приведенный ниже код скомпилируется:

```
unsafe static void Main(string[] args)
{
  int myInt2 = 5;
  SquareIntPointer(&myInt2);
  Console.WriteLine("myInt: {0}", myInt2);
}
```

Запустив на выполнение метод `Main()`, вы получите следующий вывод:

```
myInt: 25
```

Работа с операциями * и &

После установления небезопасного контекста можно строить указатели и типы данных с помощью операции `*`, а также получать адрес указываемых данных посредством операции `&`. В отличие от C или C++ в языке C# операция `*` применяется только к лежащему в основе типу, а не является префиксом имени каждой переменной указателя. Например, взгляните на показанный далее код, демонстрирующий правильный и неправильный способы объявления указателей на целочисленные переменные:

```
// Нет! В C# это некорректно!
int *pi, *pj;

// Да! Так поступают в C#.
int* pi, pj;
```

Рассмотрим следующий небезопасный метод:

```
unsafe static void PrintValueAndAddress()
{
  int myInt;

  // Определить указатель на int и присвоить ему адрес myInt.
  int* ptrToMyInt = &myInt;

  // Присвоить значение myInt, используя обращение через указатель.
  *ptrToMyInt = 123;

  // Вывести некоторые значения.
  Console.WriteLine("Value of myInt {0}", myInt);          // значение myInt
  Console.WriteLine("Address of myInt {0:X}", (int)&ptrToMyInt); //адрес myInt
}
```

Небезопасная (и безопасная) функция обмена

Разумеется, объявление указателей на локальные переменные, чтобы просто присваивать им значения (как в предыдущем примере), никогда не понадобится и к тому же неудобно. В качестве более практичного примера небезопасного кода предположим, что необходимо построить функцию обмена с использованием арифметики указателей:

```
unsafe public static void UnsafeSwap(int* i, int* j)
{
  int temp = *i;
```

```
  *i = *j;
  *j = temp;
}
```

Очень похоже на язык C, не так ли? Тем не менее, учитывая предшествующую работу, вы должны знать, что можно было бы написать безопасную версию алгоритма обмена с применением ключевого слова `ref` языка C#:

```
public static void SafeSwap(ref int i, ref int j)
{
  int temp = i;
  i = j;
  j = temp;
}
```

Функциональность обеих версий метода идентична, доказывая тем самым, что прямые манипуляции указателями в C# не являются обязательными. Ниже показана логика вызова, использующая безопасный метод `Main()`, но с небезопасным контекстом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Calling method with unsafe code *****");

  // Значения, подлежащие обмену.
  int i = 10, j = 20;

  // "Безопасный" обмен значений.
  Console.WriteLine("\n***** Safe swap *****");
  Console.WriteLine("Values before safe swap: i = {0}, j = {1}", i, j);
  SafeSwap(ref i, ref j);
  Console.WriteLine("Values after safe swap: i = {0}, j = {1}", i, j);

  // "Небезопасный" обмен значений.
  Console.WriteLine("\n***** Unsafe swap *****");
  Console.WriteLine("Values before unsafe swap: i = {0}, j = {1}", i, j);
  unsafe { UnsafeSwap(&i, &j); }

  Console.WriteLine("Values after unsafe swap: i = {0}, j = {1}", i, j);
  Console.ReadLine();
}
```

Доступ к полям через указатели (операция ->)

Теперь предположим, что определена простая безопасная структура `Point`:

```
struct Point
{
  public int x;
  public int y;

  public override string ToString()
  {
    return string.Format("({0}, {1})", x, y);
  }
}
```

В случае объявления указателя на тип `Point` для доступа к открытым членам структуры понадобится применять операцию доступа к полям (имеющую вид `->`). Как упоминалось в табл. 11.2, она представляет собой небезопасную версию стандартной (безопасной) операции точки (`.`). В сущности, используя операцию обращения к указателю (`*`), можно разыменовать указатель для применения операции точки. Взгляните на следующий небезопасный метод:

```
unsafe static void UsePointerToPoint()
{
  // Доступ к членам через указатель.
  Point point;
  Point* p = &point;
  p->x = 100;
  p->y = 200;
  Console.WriteLine(p->ToString());
  // Доступ к членам через разыменованный указатель.
  Point point2;
  Point* p2 = &point2;
  (*p2).x = 100;
  (*p2).y = 200;
  Console.WriteLine((*p2).ToString());
}
```

Ключевое слово stackalloc

В небезопасном контексте может возникнуть необходимость в объявлении локальной переменной, для которой память выделяется непосредственно в стеке вызовов (и потому она не обрабатывается сборщиком мусора .NET). Для этого в языке C# предусмотрено ключевое слово stackalloc, которое является эквивалентом функции _alloca библиотеки времени выполнения C. Вот простой пример:

```
unsafe static void UnsafeStackAlloc()
{
  char* p = stackalloc char[256];
  for (int k = 0; k < 256; k++)
    p[k] = (char)k;
}
```

Закрепление типа посредством ключевого слова fixed

В предыдущем примере вы видели, что выделение фрагмента памяти внутри небезопасного контекста может делаться с помощью ключевого слова stackalloc. Из-за природы этой операции выделенная память очищается, как только выделяющий ее метод возвращает управление (т.к. память распределена в стеке). Однако рассмотрим более сложный пример. Во время исследования операции -> создавался тип значения по имени Point. Как и все типы значений, выделяемая его экземплярам память исчезает из стека по окончании выполнения. Предположим, что тип Point взамен определен как *ссылочный*:

```
class PointRef  // <= Переименован и набран заново.
{
  public int x;
  public int y;
  public override string ToString()
  {
    return string.Format("({0}, {1})", x, y);
  }
}
```

Как вам известно, если в вызывающем коде объявляется переменная типа Point, то память для нее выделяется в куче, поддерживающей сборку мусора. И тут возникает животрепещущий вопрос: а что если небезопасный контекст пожелает взаимодействовать с этим объектом (или любым другим объектом из кучи)? Учитывая, что сборка мусора может произойти в любой момент, вы только вообразите, какие проблемы возникнут

при обращении к членам Point именно в тот момент, когда происходит реорганизация кучи. Теоретически может случиться так, что небезопасный контекст попытается взаимодействовать с членом, который больше недоступен или был перемещен в другое место кучи после ее очистки с учетом поколений (что является очевидной проблемой).

Для фиксации переменной ссылочного типа в памяти из небезопасного контекста язык C# предлагает ключевое слово fixed. Оператор fixed устанавливает указатель на управляемый тип и "закрепляет" эту переменную на время выполнения кода. Без fixed от указателей на управляемые переменные было бы мало толку, поскольку сборщик мусора может перемещать переменные в памяти непредсказуемым образом. (На самом деле компилятор C# не позволит установить указатель на управляемую переменную, если оператор fixed отсутствует.)

Таким образом, если вы создали объект Point и хотите взаимодействовать с его членами, то должны написать следующий код (либо иначе получить ошибку на этапе компиляции):

```
unsafe public static void UseAndPinPoint()
{
  PointRef pt = new PointRef ();
  pt.x = 5;
  pt.y = 6;
  // Закрепить указатель pt на месте, чтобы он не мог
  // быть перемещен или уничтожен сборщиком мусора.
  fixed (int* p = &pt.x)
  {
    // Использовать здесь переменную int*!
  }
  // Указатель pt теперь не закреплен и готов.
  // к сборке мусора после завершения метода.
  Console.WriteLine ("Point is: {0}", pt);
}
```

Короче говоря, ключевое слово fixed позволяет строить оператор, который фиксирует ссылочную переменную в памяти, чтобы ее адрес оставался постоянным на протяжении работы оператора (или блока операторов). Каждый раз, когда вы взаимодействуете со ссылочным типом из контекста небезопасного кода, закрепление ссылки обязательно.

Ключевое слово sizeof

Последнее ключевое слово C#, связанное с небезопасным кодом — это sizeof. Как и в C++, ключевое слово sizeof в C# используется для получения размера в байтах *встроенного типа данных*, но не специального типа, разве только в небезопасном контексте. Например, показанный ниже метод не нуждается в объявлении "небезопасным", т.к. все аргументы ключевого слова sizeof относятся к встроенным типам:

```
static void UseSizeOfOperator()
{
  Console.WriteLine("The size of short is {0}.", sizeof(short)); //размер short
  Console.WriteLine("The size of int is {0}.", sizeof(int));  // размер int
  Console.WriteLine("The size of long is {0}.", sizeof(long)); // размер long
}
```

Тем не менее, если вы хотите получить размер специальной структуры Point, то этот метод понадобится модифицировать (обратите внимание на добавление ключевого слова unsafe):

```
unsafe static void UseSizeOfOperator()
{
  ...
  Console.WriteLine("The size of Point is {0}.", sizeof(Point)); // размер Point
}
```

Исходный код. Проект `UnsafeCode` доступен в подкаталоге `Chapter_11`.

На этом обзор нескольких более сложных средств языка программирования C# завершен. Напоследок снова необходимо отметить, что в большинстве проектов .NET эти средства могут вообще не понадобиться (особенно указатели). Тем не менее, как будет показано в последующих главах, некоторые средства действительно полезны (а то и обязательны) при работе с API-интерфейсами LINQ, в частности — расширяющие методы и анонимные типы.

Резюме

Целью этой главы было углубление знаний языка программирования C#. Первым делом мы исследовали разнообразные более сложные конструкции в типах (методы индексаторов, перегруженные операции и специальные процедуры преобразования).

Затем мы рассмотрели роль расширяющих методов и анонимных типов. Как вы увидите в следующей главе, эти средства удобны при работе с API-интерфейсами LINQ (хотя при желании их можно применять повсеместно в коде). Вспомните, что анонимные методы позволяют быстро моделировать "форму" типа, в то время как расширяющие методы дают возможность добавлять новую функциональность к типам без необходимости в определении подклассов.

Финальная часть главы была посвящена небольшому набору менее известных ключевых слов (`sizeof`, `unsafe` и т.п.); наряду с этим рассматривалась работа с низкоуровневыми типами указателей. Как было установлено в процессе исследования типов указателей, в подавляющем большинстве приложений C# их *никогда* не придется использовать.

ГЛАВА 12

LINQ to Objects

Независимо от типа приложения, которое вы создаете с использованием платформы .NET, ваша программа определенно нуждается в доступе к какой-нибудь форме данных во время выполнения. Разумеется, данные могут находиться в многочисленных местах, включая файлы XML, реляционные базы данных, коллекции в памяти и элементарные массивы. Исторически сложилось так, что в зависимости от места хранения данных программистам приходилось применять разные и несвязанные друг с другом API-интерфейсы. Набор технологий LINQ (Language Integrated Query — язык интегрированных запросов), появившийся в версии .NET 3.5, предоставил краткий, симметричный и строго типизированный способ доступа к широкому разнообразию хранилищ данных. В этой главе изучение LINQ начинается с исследования LINQ to Objects.

Прежде чем погрузиться в LINQ to Objects, в первой части главы предлагается обзор основных программных конструкций языка C#, которые делают возможным существование LINQ. По мере чтения главы вы убедитесь, насколько полезны (а иногда и обязательны) такие средства, как неявно типизированные переменные, синтаксис инициализации объектов, лямбда-выражения, расширяющие методы и анонимные типы.

После пересмотра поддерживающей инфраструктуры в оставшемся материале главы будет представлена модель программирования LINQ и объяснена ее роль внутри платформы .NET. Вы узнаете предназначение операций и выражений запросов, позволяющих определять операторы, которые будут опрашивать источник данных для выдачи требуемого результирующего набора. Попутно будут строиться многочисленные примеры LINQ, взаимодействующие с данными в массивах и коллекциях различного типа (обобщенных и необобщенных), а также исследоваться сборки, пространства имен и типы, которые представляют API-интерфейс LINQ to Objects.

На заметку! Информация, приведенная в этой главе, послужит фундаментом для освоения материала последующих глав книги, в которых описаны дополнительные технологии LINQ, включая LINQ to XML (глава 24), Parallel LINQ (глава 19) и LINQ to Entities (глава 23).

Программные конструкции, специфичные для LINQ

На самом высоком уровне LINQ можно воспринимать как строго типизированный язык запросов, встроенный непосредственно в грамматику самого языка C#. Используя LINQ, можно создавать любое количество выражений, которые выглядят и ведут себя подобно SQL-запросам к базе данных. Однако запрос LINQ может применяться к любому числу хранилищ данных, включая хранилища, которые не имеют ничего общего с настоящими реляционными базами данных.

На заметку! Хотя запросы LINQ внешне похожи на запросы SQL, их синтаксис *не* идентичен. В действительности многие запросы LINQ имеют формат, прямо противоположный формату подобного запроса к базе данных! Если вы попытаетесь отобразить LINQ непосредственно на SQL, то определенно запутаетесь. Чтобы этого не произошло, рекомендуется воспринимать запросы LINQ как уникальные операторы, которые просто случайно оказались похожими на SQL.

Когда LINQ впервые был представлен в составе платформы .NET 3.5, языки C# и VB уже были расширены огромным количеством программных конструкций для поддержки набора технологий LINQ. В частности, язык C# использует следующие связанные с LINQ средства:

- неявно типизированные локальные переменные;
- синтаксис инициализации объектов и коллекций;
- лямбда-выражения;
- расширяющие методы;
- анонимные типы.

Эти средства уже детально рассматривались в других главах книги. Тем не менее, чтобы освежить все в памяти, давайте быстро вспомним о каждом из средств по очереди, удостоверившись в правильном их понимании.

На заметку! Из-за того, что в последующих разделах приводится обзор материала, рассматриваемого где-то в других местах книги, проект кода C# здесь не приводится.

Неявная типизация локальных переменных

В главе 3 вы узнали о ключевом слове `var` языка C#. Оно позволяет определять локальную переменную без явного указания типа данных. Однако такая переменная будет строго типизированной, потому что компилятор определит ее корректный тип данных на основе начального присваивания. Вспомните показанный ниже код примера из главы 3:

```
static void DeclareImplicitVars()
{
  // Неявно типизированные локальные переменные.
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";

  // Вывести имена лежащих в основе типов.
  Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
  Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
  Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

Это языковое средство очень удобно и зачастую обязательно, когда применяется LINQ. Как вы увидите на протяжении главы, многие запросы LINQ возвращают последовательность типов данных, которые не будут известны вплоть до этапа компиляции. Учитывая, что лежащий в основе тип данных не известен до того, как приложение скомпилируется, вполне очевидно, что явно объявить такую переменную невозможно!

Синтаксис инициализации объектов и коллекций

В главе 5 объяснялась роль синтаксиса инициализации объектов, который позволяет создавать переменную типа класса или структуры и устанавливать любое количество ее открытых свойств за один прием. В результате получается компактный (и по-прежнему легко читаемый) синтаксис, который может использоваться для подготовки объектов к употреблению. Также вспомните из главы 9, что язык C# поддерживает похожий синтаксис инициализации коллекций объектов. Взгляните на следующий фрагмент кода, где синтаксис инициализации коллекций применяется для наполнения `List<T>` объектами `Rectangle`, каждый из которых состоит из пары объектов `Point`, представляющих точку с координатами (x, y):

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
  new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
                 BottomRight = new Point { X = 200, Y = 200}},
  new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
                 BottomRight = new Point { X = 100, Y = 100}},
  new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
                 BottomRight = new Point { X = 90, Y = 75}}
};
```

Несмотря на то что использовать синтаксис инициализации коллекций или объектов совершенно не обязательно, с его помощью можно получить более компактную кодовую базу. Более того, этот синтаксис в сочетании с неявной типизацией локальных переменных позволяет объявлять анонимный тип, что удобно при создании проекций LINQ. О проекциях LINQ речь пойдет позже в настоящей главе.

Лямбда-выражения

Лямбда-операция C# (=>) была полностью описана в главе 10. Вспомните, что данная операция позволяет строить лямбда-выражение, которое может применяться в любой момент при вызове метода, требующего строго типизированный делегат в качестве аргумента. Лямбда-выражения значительно упрощают работу с делегатами .NET, т.к. сокращают объем кода, который должен быть написан вручную. Лямбда-выражения могут быть описаны следующим образом:

```
( АргументыДляОбработки ) => { ОбрабатывающиеОператоры }
```

В главе 10 было показано, как взаимодействовать с методом `FindAll()` обобщенного класса `List<T>` с использованием трех разных подходов. После работы с низкоуровневым делегатом `Predicate<T>` и анонимным методом C# мы пришли к приведенной ниже (исключительно компактной) версии, в которой использовалось лямбда-выражение:

```
static void LambdaExpressionSyntax()
{
  // Создать список целочисленных значений.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
  // Теперь использовать лямбда-выражение C#.
  List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
  // Вывести четные числа.
  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

Лямбда-выражения будут удобны при работе с объектной моделью, лежащей в основе LINQ. Как вы вскоре выясните, операции запросов LINQ в C# — это просто сокращенная запись для вызова методов класса по имени `System.Linq.Enumerable`. Эти методы обычно всегда требуют передачи в качестве параметров делегатов (в частности, делегата `Func<>`), которые применяются для обработки данных с целью выдачи корректного результирующего набора. За счет использования лямбда-выражений можно упростить код и позволить компилятору вывести нужный делегат.

Расширяющие методы

Расширяющие методы C# позволяют оснащать существующие классы новой функциональностью без необходимости в создании подклассов. Кроме того, расширяющие методы дают возможность добавления новой функциональности к запечатанным классам и структурам, которые в принципе не допускают построения подклассов. Вспомните из главы 11, что когда создается расширяющий метод, первый его параметр снабжается ключевым словом `this` и помечает расширяемый тип. Также вспомните, что расширяющие методы должны всегда определяться внутри статического класса, а потому объявляться с применением ключевого слова `static`. Вот пример:

```
namespace MyExtensions
{
  static class ObjectExtensions
  {
    // Определить расширяющий метод для System.Object.
    public static void DisplayDefiningAssembly(this object obj)
    {
      Console.WriteLine("{0} lives here:\n\t->{1}\n", obj.GetType().Name,
        Assembly.GetAssembly(obj.GetType()));
    }
  }
}
```

Чтобы использовать такое расширение, приложение должно импортировать пространство имен, определяющее расширение (и возможно добавить ссылку на внешнюю сборку). После этого можно приступать к написанию кода:

```
static void Main(string[] args)
{
  // Поскольку все типы расширяют System.Object, все классы
  // и структуры могут использовать это расширение.
  int myInt = 12345678;
  myInt.DisplayDefiningAssembly();

  System.Data.DataSet d = new System.Data.DataSet();
  d.DisplayDefiningAssembly();
  Console.ReadLine();
}
```

При работе с LINQ вам редко (если вообще когда-либо) потребуется вручную строить собственные расширяющие методы. Тем не менее, создавая выражения запросов LINQ, вы на самом деле будете применять многочисленные расширяющие методы, уже определенные Microsoft. Фактически каждая операция запроса LINQ в C# представляет собой сокращенную запись для ручного вызова лежащего в основе расширяющего метода, который обычно определен в служебном классе `System.Linq.Enumerable`.

Анонимные типы

Последним средством языка C#, описание которого мы здесь кратко повторим, являются анонимные типы, рассмотренные в главе 11. Это средство может использоваться для быстрого моделирования "формы" данных, разрешая компилятору генерировать на этапе компиляции новое определение класса, которое основано на предоставленном наборе пар "имя-значение". Вспомните, что такой тип будет составлен с применением семантики на основе значений, а каждый виртуальный метод `System.Object` будет соответствующим образом переопределен. Чтобы определить анонимный тип, понадобится объявить неявно типизированную переменную и указать форму данных с использованием синтаксиса инициализации объектов:

```
// Создать анонимный тип, состоящий из еще одного анонимного типа.
var purchaseItem = new {
  TimeBought = DateTime.Now,
  ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
  Price = 34.000};
```

Анонимные типы часто применяются в LINQ, когда необходимо проецировать новые формы данных на лету. Например, предположим, что есть коллекция объектов `Person`, и вы хотите использовать LINQ для получения информации о возрасте и номере карточки социального страхования в каждом объекте. Применяя проекцию LINQ, можно предоставить компилятору возможность генерации нового анонимного типа, содержащего интересующую информацию.

Роль LINQ

На этом краткий обзор средств языка C#, которые позволяют LINQ делать свою работу, завершен. Однако важно понимать, зачем вообще нужен язык LINQ. Любой разработчик программного обеспечения согласится с утверждением, что подавляющее большинство времени при программировании тратится на получение и манипулирование данными. Когда говорят о "данных", на ум немедленно приходит информация, хранящаяся внутри реляционных баз данных. Тем не менее, другими популярными местоположениями для данных являются документы XML или простые текстовые файлы.

Данные могут находиться в многочисленных местах помимо этих двух распространенных хранилищ информации. Например, пусть имеется массив или обобщенный тип `List<T>`, содержащий 300 целых чисел, и требуется получить подмножество, которое удовлетворяет заданному критерию (например, только четные или нечетные числа, только простые числа, только неповторяющиеся числа больше 50). Или, возможно, при использовании API-интерфейсов рефлексии необходимо получить в массиве элементов `Type` метаданные только для классов, производных от определенного родительского класса. На самом деле данные находятся *повсюду*.

До появления версии .NET 3.5 взаимодействие с отдельной разновидностью данных требовало от программистов применения совершенно несходных API-интерфейсов. В табл. 12.1 описаны некоторые популярные API-интерфейсы, используемые для доступа к разнообразным типам данных (наверняка вы в состоянии привести и другие примеры).

Разумеется, в этих подходах к манипулированию данными ничего ошибочного нет. В сущности, вы можете (и будете) работать напрямую с ADO.NET, пространствами имен XML, службами рефлексии и разнообразными типами коллекций. Однако основная проблема заключается в том, что каждый из этих API-интерфейсов является "самостоятельным островком", трудно интегрируемым с другими.

Таблица 12.1. Способы манипулирования различными типами данных

Интересующие данные	Способ получения
Реляционные данные	`System.Data.dll`, `System.Data.SqlClient.dll` и т.д.
Данные документов XML	`System.Xml.dll`
Таблицы метаданных	Пространство имен `System.Reflection`
Коллекции объектов	Пространства имен `System.Array` и `System.Collections/System.Collections.Generic`

Правда, можно (например) сохранить объект `DataSet` из ADO.NET в документ XML и затем манипулировать им посредством пространств имен `System.Xml`, но все равно манипуляции данными остаются довольно асимметричными.

В рамках API-интерфейса LINQ была предпринята попытка предложить программистам согласованный, симметричный способ получения и манипулирования "данными" в широком смысле этого понятия. Применяя LINQ, можно создавать прямо внутри языка программирования C# конструкции, которые называются *выражениями запросов*. Такие выражения запросов основаны на многочисленных операциях запросов, которые намеренно сделаны похожими внешне и по поведению (но не идентичными) на выражения SQL.

Тем не менее, уловка заключается в том, что выражение запроса может использоваться для взаимодействия с разнообразными типами данных — даже с такими данными, которые не имеют ничего общего с реляционными базами данных. Строго говоря, LINQ представляет собой термин, в целом описывающий сам подход доступа к данным. Однако в зависимости от того, где применяются запросы LINQ, вы встретите разные обозначения вроде перечисленных ниже.

- LINQ to Objects. Этот термин относится к действию по применению запросов LINQ к массивам и коллекциям.
- LINQ to XML. Этот термин относится к действию по использованию LINQ для манипулирования и запрашивания документов XML.
- LINQ to DataSet. Этот термин относится к действию по применению запросов LINQ к объектам `DataSet` из ADO.NET.
- LINQ to Entities. Этот аспект LINQ позволяет использовать запросы LINQ внутри API-интерфейса ADO.NET Entity Framework (EF).
- Parallel LINQ (он же PLINQ). Этот аспект делает возможной параллельную обработку данных, возвращаемых из запроса LINQ.

В настоящее время LINQ является неотъемлемой частью библиотек базовых классов .NET, управляемых языков и самой среды Visual Studio.

Выражения LINQ строго типизированы

Важно также отметить, что выражение запроса LINQ (в отличие от традиционного оператора SQL) *строго типизировано*. Следовательно, компилятор C# следит за этим и гарантирует, что выражения оформлены корректно с точки зрения синтаксиса. Инструменты вроде Visual Studio могут применять метаданные для поддержки удобных средств, таких как IntelliSense, автозавершение и т.д.

Основные сборки LINQ

Как упоминалось в главе 2, в диалоговом окне New Project (Новый проект) среды Visual Studio доступна возможность выбора версии платформы .NET, для которой должна проводиться компиляция. Когда компиляция осуществляется для .NET 3.5 или последующей версии, каждый шаблон проекта автоматически ссылается на основные сборки LINQ, что можно просмотреть в окне Solution Explorer. Основные сборки LINQ описаны в табл. 12.2. В оставшихся главах вы столкнетесь с дополнительными библиотеками LINQ.

Таблица 12.2. Основные сборки, связанные с LINQ

Сборка	Описание
`System.Core.dll`	Определяет типы, представляющие основной API-интерфейс LINQ. Это единственная сборка, к которой вы должны иметь доступ, если хотите использовать любые API-интерфейсы LINQ, включая LINQ to Objects
`System.Data.DataSetExtensions.dll`	Определяет набор типов для интеграции типов ADO.NET в парадигму программирования LINQ (LINQ to DataSet)
`System.Xml.Linq.dll`	Предоставляет функциональность для применения LINQ с данными документов XML (LINQ to XML)

Для работы с LINQ to Objects вы должны обеспечить, чтобы в каждом файле кода C#, который содержит запросы LINQ, было импортировано пространство имен `System.Linq` (определенное главным образом внутри сборки `System.Core.dll`). В противном случае возникнут проблемы. Если на этапе компиляции вы получили сообщение об ошибке следующего вида:

```
Error 1 Could not find an implementation of the query pattern for source
type 'int[]'. 'Where' not found. Are you missing a reference to 'System.Core.dll'
or a using directive for 'System.Linq'?
```

Ошибка 1 Не удается обнаружить реализацию шаблона запросов для исходного типа int[]. Where не найдено. Возможно, пропущена ссылка на System.Core.dll или директива using для System.Linq?

то очень высока вероятность, что в файле кода C# отсутствует нужная директива `using`:

```
using System.Linq;
```

Применение запросов LINQ к элементарным массивам

Чтобы начать исследование LINQ to Objects, давайте построим приложение, которое будет применять запросы LINQ к разнообразным объектам типа массива. Создадим проект консольного приложения под названием `LinqOverArray` и определим внутри класса `Program` статический вспомогательный метод по имени `QueryOverStrings()`. Внутри метода создадим массив типа `string`, содержащий несколько произвольных элементов (скажем, названий видеоигр). Необходимо обеспечить, чтобы хотя бы два элемента содержали числовые значения, а несколько элементов включали внутренние пробелы:

```
static void QueryOverStrings()
{
  // Предположим, что есть массив строк.
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                "Fallout 3", "Daxter", "System Shock 2"};
}
```

Теперь модифицируем метод `Main()` для вызова `QueryOverStrings()`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with LINQ to Objects *****\n");
  QueryOverStrings();
  Console.ReadLine();
}
```

При работе с любым массивом данных часто приходится извлекать из него подмножество элементов на основе определенного критерия. Возможно, требуется получить только элементы, которые содержат число (например, "System Shock 2", "Uncharted 2" и "Fallout 3"), имеют длину больше или меньше заданного количества символов, не содержат встроенные пробелы (скажем, "Morrowind" или "Daxter") и т.п. В то время как такие задачи определенно можно решать с использованием членов типа `System.Array`, прикладывая приличные усилия, выражения запросов LINQ значительно упрощают процесс.

Исходя из предположения, что из массива нужно получить только элементы, содержащие внутри себя пробел, и представить их в алфавитном порядке, можно построить следующее выражение запроса LINQ:

```
static void QueryOverStrings()
{
  // Предположим, что имеется массив строк.
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // Построить выражение запроса для нахождения
  // элементов массива, которые содержат пробелы.
  IEnumerable<string> subset = from g in currentVideoGames
                               where g.Contains(" ") orderby g select g;

  // Вывести результаты.
  foreach (string s in subset)
    Console.WriteLine("Item: {0}", s);
}
```

Обратите внимание, что в созданном здесь выражении запроса применяются операции `from`, `in`, `where`, `orderby` и `select` языка LINQ. Формальности синтаксиса выражений запросов будут подробно излагаться далее в главе. Тем не менее, даже сейчас вы в состоянии прочесть этот оператор примерно как "предоставить мне элементы из `currentVideoGames`, содержащие пробелы, в алфавитном порядке".

Каждому элементу, который соответствует критерию поиска, будет назначено имя `g` (от "game"); однако подошло бы любое допустимое имя переменной C#:

```
IEnumerable<string> subset = from game in currentVideoGames
                             where game.Contains(" ") orderby
                             game select game;
```

Возвращенная последовательность сохраняется в переменной по имени `subset`, которая имеет тип, реализующий обобщенную версию интерфейса `IEnumerable<T>`, где `T` — тип `System.String` (в конце концов, вы запрашиваете массив элементов `string`).

После получения результирующего набора его элементы затем просто выводятся на консоль с использованием стандартной конструкции `foreach`. Запустив приложение, вы получите такой вывод:

```
***** Fun with LINQ to Objects *****
Item: Fallout 3
Item: System Shock 2
Item: Uncharted 2
```

Решение без использования LINQ

Конечно, применение LINQ никогда не бывает обязательным. При желании идентичный результирующий набор можно получить без участия LINQ с помощью таких программных конструкций, как операторы `if` и циклы `for`. Ниже приведен метод, который выдает тот же самый результат, что и `QueryOverStrings()`, но в намного более многословной манере:

```
static void QueryOverStringsLongHand()
{
  // Предположим, что имеется массив строк.
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};
  string[] gamesWithSpaces = new string[5];
  for (int i = 0; i < currentVideoGames.Length; i++)
  {
    if (currentVideoGames[i].Contains(" "))
      gamesWithSpaces[i] = currentVideoGames[i];
  }
  // Отсортировать набор.
  Array.Sort(gamesWithSpaces);
  // Вывести результаты.
  foreach (string s in gamesWithSpaces)
  {
    if( s != null)
      Console.WriteLine("Item: {0}", s);
  }
  Console.WriteLine();
}
```

Несмотря на возможные пути улучшения этого метода, факт остается фактом — запросы LINQ могут радикально упростить процесс извлечения новых подмножеств данных из источника. Вместо построения вложенных циклов, сложной логики `if/else`, временных типов данных и т.п. компилятор C# сделает всю черновую работу за вас, как только вы создадите подходящий запрос LINQ.

Выполнение рефлексии результирующего набора LINQ

А теперь определим в классе `Program` дополнительный вспомогательный метод по имени `ReflectOverQueryResults()`, который выводит на консоль разнообразные детали о результирующем наборе LINQ (обратите внимание на параметр типа `System.Object`, позволяющий учитывать множество типов результирующих наборов):

```
static void ReflectOverQueryResults(object resultSet)
{
  Console.WriteLine("***** Info about your query *****");
```

```
  // Вывести тип результирующего набора.
  Console.WriteLine("resultSet is of type: {0}", resultSet.GetType().Name);

  // Вывести местоположение результирующего набора.
  Console.WriteLine("resultSet location: {0}",
    resultSet.GetType().Assembly.GetName().Name);
}
```

Вызвав этот метод внутри `QueryOverStrings()` сразу после вывода полученного подмножества и запустив приложение, легко заметить, что данное подмножество в действительности представляет собой экземпляр обобщенного типа `OrderedEnumerable<TElement, TKey>` (представленного в коде CIL как `OrderedEnumerable`2`), который является внутренним абстрактным типом, находящимся в сборке `System.Core.dll`:

```
***** Info about your query *****

resultSet is of type: OrderedEnumerable`2
resultSet location: System.Core
```

На заметку! Многие типы, представляющие результат LINQ, в браузере объектов Visual Studio скрыты. Эти низкоуровневые типы не предназначены для прямого использования в приложениях.

LINQ и неявно типизированные локальные переменные

Хотя в приведенной программе относительно легко выяснить, что результирующий набор может быть интерпретирован как перечисление объектов `string` (например, `IEnumerable<string>`), совсем *не* так очевиден тот факт, что подмножество на самом деле имеет тип `OrderedEnumerable<TElement, TKey>`.

Поскольку результирующие наборы LINQ могут быть представлены с применением порядочного количества типов из разнообразных пространств имен LINQ, было бы утомительно определять подходящий тип для хранения результирующего набора. Причина в том, что во многих случаях лежащий в основе тип не очевиден и даже напрямую не доступен в коде (и как вы увидите, в ряде ситуаций тип генерируется на этапе компиляции).

Чтобы еще более подчеркнуть это обстоятельство, ниже показан дополнительный вспомогательный метод, определенный внутри класса `Program` (который должен быть вызван из метода `Main()`):

```
static void QueryOverInts()
{
  int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
  // Вывести только элементы меньше 10.
  IEnumerable<int> subset = from i in numbers where i < 10 select i;

  foreach (int i in subset)
    Console.WriteLine("Item: {0}", i);
  ReflectOverQueryResults(subset);
}
```

В рассматриваемом случае переменная `subset` имеет совершенно другой внутренний тип. На этот раз тип, реализующий интерфейс `IEnumerable<int>`, представляет собой низкоуровневый класс по имени `WhereArrayIterator<T>`:

```
Item: 1
Item: 2
Item: 3
Item: 8
```

```
***** Info about your query *****
resultSet is of type: WhereArrayIterator`1
resultSet location: System.Core
```

Учитывая, что точный тип запроса LINQ определенно не очевиден, в первых примерах результаты запросов были представлены как переменная IEnumerable<T>, где T — тип данных в возвращенной последовательности (string, int и т.д.). Тем не менее, ситуация по-прежнему является довольно запутанной. Чтобы еще больше все усложнить, стоит упомянуть, что поскольку интерфейс IEnumerable<T> расширяет необобщенный IEnumerable, получать результат запроса LINQ допускается и так:

```
System.Collections.IEnumerable subset =
  from i in numbers where i < 10 select i;
```

К счастью, неявная типизация значительно проясняет картину при работе с запросами LINQ:

```
static void QueryOverInts()
{
  int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
  // Здесь используется неявная типизация...
  var subset = from i in numbers where i < 10 select i;
  // ...и здесь тоже.
  foreach (var i in subset)
    Console.WriteLine("Item: {0} ", i);
  ReflectOverQueryResults(subset);
}
```

В качестве эмпирического правила: при захвате результатов запроса LINQ всегда необходимо использовать неявную типизацию. Однако помните, что (в подавляющем большинстве случаев) *действительное* возвращаемое значение имеет тип, реализующий интерфейс IEnumerable<T>.

Какой точно тип кроется за этим (OrderedEnumerable<TElement, TKey>, WhereArrayIterator<T> и т.п.) к делу не относится, и определять его вовсе не обязательно. Как было показано в предыдущем примере кода, можно просто применить ключевое слово var внутри конструкции foreach для прохода по извлеченным данным.

LINQ и расширяющие методы

Несмотря на то что в текущем примере вовсе не требуется напрямую писать какие-то расширяющие методы, на самом деле они благополучно используются на заднем плане. Выражения запросов LINQ могут применяться для прохода по содержимому контейнеров данных, которые реализуют обобщенный интерфейс IEnumerable<T>. Тем не менее, класс System.Array в .NET (используемый для представления массива строк и массива целых чисел) *не* реализует этот контракт:

```
// Тип System.Array, похоже, не реализует корректную
// инфраструктуру для выражений запросов!
public abstract class Array : ICloneable, IList, ICollection,
  IEnumerable, IStructuralComparable, IStructuralEquatable
{
  ...
}
```

Хотя класс System.Array не реализует напрямую интерфейс IEnumerable<T>, он косвенно получает необходимую функциональность этого типа (а также многие другие члены, связанные с LINQ) через статический тип класса System.Linq.Enumerable.

В этом служебном классе определено множество обобщенных расширяющих методов (таких как `Aggregate<T>()`, `First<T>()`, `Max<T>()` и т.д.), которые класс `System.Array` (и другие типы) получают в свое распоряжение на заднем плане. Таким образом, если вы примените операцию точки к локальной переменной `currentVideoGames`, то обнаружите большое количество членов, которые *отсутствуют* в формальном определении `System.Array` (рис. 12.1).

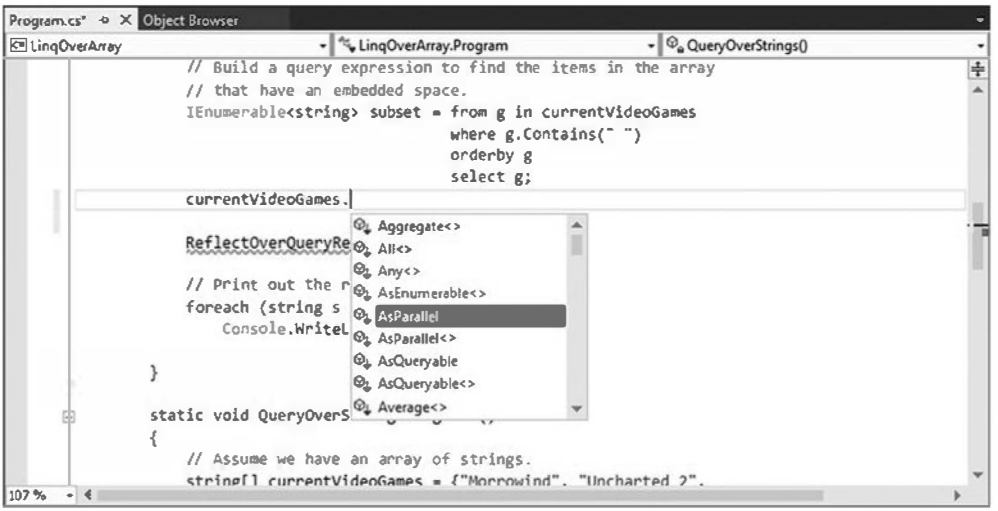


Рис. 12.1. Тип `System.Array` был расширен членами типа `System.Linq.Enumerable`

Роль отложенного выполнения

Еще один важный момент, касающийся выражений запросов LINQ, заключается в том, что в действительности они не оцениваются до тех пор, пока не начнется итерация по последовательности. Формально это называется *отложенным выполнением*. Преимущество такого подхода связано с возможностью применения одного и того же запроса LINQ многократно к тому же самому контейнеру и полной гарантией получения актуальных результатов. Взгляните на следующее обновление метода `QueryOverInts()`:

```
static void QueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  // Получить числа меньше 10.
  var subset = from i in numbers where i < 10 select i;

  // Оператор LINQ здесь оценивается!
  foreach (var i in subset)
    Console.WriteLine("{0} < 10", i);
  Console.WriteLine();
  // Изменить некоторые данные в массиве.
  numbers[0] = 4;

  // Снова производится оценка!
  foreach (var j in subset)
    Console.WriteLine("{0} < 10", j);

  Console.WriteLine();
  ReflectOverQueryResults(subset);
}
```

Ниже показан вывод, полученный в результате запуска программы. Обратите внимание, что во второй итерации по запрошенной последовательности появился дополнительный член, т.к. для первого элемента массива было установлено значение меньше 10:

```
1 < 10
2 < 10
3 < 10
8 < 10

4 < 10
1 < 10
2 < 10
3 < 10
8 < 10
```

Среда Visual Studio обладает одним полезным аспектом: если вы поместите точку останова перед оценкой запроса LINQ, то получите возможность просматривать содержимое во время сеанса отладки. Просто наведите курсор мыши на переменную результирующего набора LINQ (`subset` на рис. 12.2) и вам будет предложено выполнить запрос, развернув узел Results View (Представление результатов).

Рис. 12.2. Отладка выражений LINQ

Роль немедленного выполнения

Когда требуется оценить выражение LINQ за пределами логики `foreach`, можно вызывать любое количество расширяющих методов, определенных в типе `Enumerable`, таких как `ToArray<T>`, `ToDictionary<TSource,TKey>()` и `ToList<T>()`. Все эти методы приводят к выполнению запроса LINQ в момент их вызова для получения снимка данных. Затем полученным снимком данных можно манипулировать независимым образом.

```
static void ImmediateExecution()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  // Получить данные НЕМЕДЛЕННО как int[].
  int[] subsetAsIntArray =
    (from i in numbers where i < 10 select i).ToArray<int>();

  // Получить данные НЕМЕДЛЕННО как List<int>.
  List<int> subsetAsListOfInts =
    (from i in numbers where i < 10 select i).ToList<int>();
}
```

Обратите внимание, что для вызова методов `Enumerable` выражение LINQ целиком помещено в круглые скобки для приведения к корректному внутреннему типу (каким бы он ни был).

Вспомните из главы 9, что если компилятор C# может однозначно определить параметр типа обобщенного элемента, то вы не обязаны указывать этот параметр типа. Следовательно, `ToArray<T>` (или `ToList<T>`) можно было бы вызвать так:

```
int[] subsetAsIntArray =
  (from i in numbers where i < 10 select i).ToArray();
```

Полезность немедленного выполнения очевидна, когда нужно возвратить результаты запроса LINQ внешнему вызывающему коду. Это и будет темой следующего раздела главы.

Исходный код. Проект `LinqOverArray` доступен в подкаталоге `Chapter_12`.

Возвращение результатов запроса LINQ

Внутри класса (или структуры) можно определить поле, значением которого будет результат запроса LINQ. Однако для этого нельзя использовать неявную типизацию (т.к. ключевое слово `var` не может применяться к полям), и целью запроса LINQ не могут быть данные уровня экземпляра, так что он должен быть статическим. С учетом указанных ограничений необходимость в написании кода следующего вида будет возникать редко:

```
class LINQBasedFieldsAreClunky
{
  private static string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // Здесь нельзя использовать неявную типизацию! Тип subset должен быть известен!
  private IEnumerable<string> subset = from g in currentVideoGames
    where g.Contains(" ") orderby g select g;

  public void PrintGames()
  {
    foreach (var item in subset)
    {
      Console.WriteLine(item);
    }
  }
}
```

Запросы LINQ очень часто определяются внутри области действия метода или свойства. Кроме того, для упрощения программирования результирующий набор будет храниться в неявно типизированной локальной переменной, использующей ключевое слово `var`. Вспомните из главы 3, что неявно типизированные переменные не могут применяться для определения параметров, возвращаемых значений, а также полей класса или структуры.

Итак, вас наверняка интересует, каким образом возвратить результат запроса внешнему коду. Смотря по обстоятельствам. Если у вас есть результирующий набор, состоящий из строго типизированных данных, такой как массив строк или список `List<T>` объектов `Car`, то вы могли бы отказаться от использования ключевого слова `var` и указать подходящий тип `IEnumerable<T>` либо `IEnumerable` (т.к. `IEnumerable<T>` расширяет `IEnumerable`). Ниже приведен пример класса `Program` в новом проекте консольного приложения по имени `LinqRetValues`:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** LINQ Return Values *****\n");
    IEnumerable<string> subset = GetStringSubset();

    foreach (string item in subset)
    {
      Console.WriteLine(item);
    }
    Console.ReadLine();
  }
  static IEnumerable<string> GetStringSubset()
  {
    string[] colors = {"Light Red", "Green",
      "Yellow", "Dark Red", "Red", "Purple"};

    // Обратите внимание, что subset является
    // совместимым с IEnumerable<string> объектом.
    IEnumerable<string> theRedColors = from c in colors
      where c.Contains("Red") select c;

    return theRedColors;
  }
}
```

Результат выглядит вполне ожидаемо:

```
Light Red
Dark Red
Red
```

Возвращение результатов LINQ посредством немедленного выполнения

Этот пример работает ожидаемым образом только потому, что возвращаемое значение `GetStringSubset()` и запрос LINQ внутри этого метода были строго типизированными. Если применить ключевое слово `var` для определения переменной `subset`, то возвращать значение будет разрешено, *только* если метод по-прежнему прототипирован с возвращаемым типом `IEnumerable<string>` (и если неявно типизированная локальная переменная на самом деле совместима с указанным возвращаемым типом).

Поскольку оперировать с типом `IEnumerable<T>` несколько неудобно, можно задействовать немедленное выполнение. Скажем, вместо возвращения `IEnumerable<string>` можно было бы возвратить просто `string[]` при условии трансформации последовательности в строго типизированный массив. Новый метод класса `Program` именно это и делает:

```
static string[] GetStringSubsetAsArray()
{
  string[] colors = {"Light Red", "Green",
    "Yellow", "Dark Red", "Red", "Purple"};

  var theRedColors = from c in colors
    where c.Contains("Red") select c;

  // Отобразить результаты в массив.
  return theRedColors.ToArray();
}
```

При этом вызывающему коду абсолютно не известно, что полученный им результат поступил от запроса LINQ, и он просто работает с массивом строк вполне ожидаемым образом. Вот пример:

```
foreach (string item in GetStringSubsetAsArray())
{
  Console.WriteLine(item);
}
```

Немедленное выполнение также важно при попытке возвратить вызывающему коду результаты проекции LINQ. Мы исследуем эту тему чуть позже в главе. А сейчас давайте посмотрим, как применять запросы LINQ к обобщенным и необобщенным объектам коллекций.

Исходный код. Проект `LinqRetValues` доступен в подкаталоге `Chapter_12`.

Применение запросов LINQ к объектам коллекций

Помимо извлечения результатов из простого массива данных выражения запросов LINQ могут также манипулировать данными внутри классов из пространства имен `System.Collections.Generic`, таких как `List<T>`. Создадим новый проект консольного приложения по имени `ListOverCollections` и определим базовый класс `Car`, который поддерживает текущую скорость, цвет, производителя и дружественное имя:

```
class Car
{
  public string PetName {get; set;} = "";
  public string Color {get; set;} = "";
  public int Speed {get; set;}
  public string Make {get; set;} = "";
}
```

Теперь определим в методе `Main()` локальную переменную типа `List<T>` для хранения элементов типа `Car` и с помощью синтаксиса инициализации объектов заполним список несколькими новыми объектами `Car`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** LINQ over Generic Collections *****\n");

  // Создать список List<> объектов Car.
  List<Car> myCars = new List<Car>() {
    new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
    new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
    new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
    new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
    new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
  };

  Console.ReadLine();
}
```

Доступ к содержащимся в контейнере подобъектам

Применение запроса LINQ к обобщенному контейнеру ничем не отличается от такого же действия в отношении простого массива, т.к. LINQ to Objects может использоваться с любым типом, реализующим интерфейс `IEnumerable<T>`. На этот раз цель заключает-

ся в построении выражения запроса для выборки из списка myCars только тех объектов Car, у которых значение скорости больше 55.

После получения подмножества на консоль будет выведено имя каждого объекта Car за счет обращения к его свойству PetName. Предположим, что определен следующий вспомогательный метод (принимающий параметр List<Car>), который вызывается внутри Main():

```
static void GetFastCars(List<Car> myCars)
{
  // Найти в List<> все объекты Car, у которых значение Speed больше 55.
  var fastCars = from c in myCars where c.Speed > 55 select c;
  foreach (var car in fastCars)
  {
    Console.WriteLine("{0} is going too fast!", car.PetName);
  }
}
```

Обратите внимание, что выражение запроса захватывает из List<T> только те элементы, у которых значение Speed больше 55. Запустив приложение, вы увидите, что критерию поиска отвечают только два элемента — Henry и Daisy.

Чтобы построить более сложный запрос, можно искать только автомобили марки BMW со значением Speed больше 90. Для этого нужно просто создать составной булевский оператор с применением операции && языка C#:

```
static void GetFastBMWs(List<Car> myCars)
{
  // Найти быстрые автомобили BMW!
  var fastCars = from c in myCars where c.Speed > 90 && c.Make == "BMW" select c;
  foreach (var car in fastCars)
  {
    Console.WriteLine("{0} is going too fast!", car.PetName);
  }
}
```

На этот раз выводится только одно имя Henry.

Применение запросов LINQ к необобщенным коллекциям

Вспомните, что операции запросов LINQ спроектированы для работы с любым типом, реализующим интерфейс IEnumerable<T> (как напрямую, так и через расширяющие методы). Учитывая то, что класс System.Array оснащен всей необходимой инфраструктурой, может оказаться сюрпризом, что унаследованные (необобщенные) контейнеры в пространстве имен System.Collections такой поддержкой не обладают. К счастью, итерация по данным, содержащимся внутри необобщенных коллекций, по-прежнему возможна с использованием обобщенного расширяющего метода Enumerable.OfType<T>().

При вызове метода OfType<T>() на объекте необобщенной коллекции (наподобие ArrayList) нужно просто указать тип элемента внутри контейнера, чтобы извлечь совместимый с IEnumerable<T> объект. Сохранить этот элемент данных в коде можно посредством неявно типизированной переменной.

Взгляните на показанный ниже новый метод, который заполняет ArrayList набором объектов Car (не забудьте импортировать пространство имен System.Collections в файл Program.cs):

```
static void LINQOverArrayList()
{
  Console.WriteLine("***** LINQ over ArrayList *****");

  // Необобщенная коллекция объектов Car.
  ArrayList myCars = new ArrayList() {
    new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
    new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
    new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
    new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
    new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
  };

  // Трансформировать ArrayList в тип, совместимый с IEnumerable<T>.
  var myCarsEnum = myCars.OfType<Car>();

  // Создать выражение запроса, нацеленное на совместимый с IEnumerable<T> тип.
  var fastCars = from c in myCarsEnum where c.Speed > 55 select c;

  foreach (var car in fastCars)
  {
    Console.WriteLine("{0} is going too fast!", car.PetName);
  }
}
```

Аналогично предыдущим примерам этот метод, вызванный в `Main()`, отобразит только имена `Henry` и `Daisy`, основываясь на формате запроса LINQ.

Фильтрация данных с использованием метода `OfType<T>()`

Как вы уже знаете, необобщенные типы способны содержать любые комбинации элементов, поскольку члены этих контейнеров (вроде `ArrayList`) прототипированы для приема `System.Object`. Например, предположим, что `ArrayList` содержит разные элементы, часть которых являются числовыми. Получить подмножество, состоящее только из числовых данных, можно с помощью метода `OfType<T>()`, т.к. во время итерации он отфильтрует элементы, тип которых отличается от заданного:

```
static void OfTypeAsFilter()
{
  // Извлечь из ArrayList целочисленные значения.
  ArrayList myStuff = new ArrayList();
  myStuff.AddRange(new object[] { 10, 400, 8, false, new Car(), "string data" });
  var myInts = myStuff.OfType<int>();

  // Выводит 10, 400 и 8.
  foreach (int i in myInts)
  {
    Console.WriteLine("Int value: {0}", i);
  }
}
```

К этому моменту вы уже умеете применять запросы LINQ к массивам, обобщенным и необобщенным коллекциям. Такие контейнеры содержат элементарные типы C# (целочисленные и строковые данные), а также специальные классы. Следующей задачей будет изучение многочисленных дополнительных операций LINQ, которые могут использоваться для построения более сложных и полезных запросов.

Исходный код. Проект `LinqOverCollectons` доступен в подкаталоге `Chapter_12`.

Исследование операций запросов LINQ

В языке C# предопределено порядочное число операций запросов. Некоторые часто применяемые из них перечислены в табл. 12.3.

На заметку! Документация .NET Framework SDK содержит подробные сведения по каждой операции LINQ языка C# (ищите раздел "LINQ General Programming Guide" ("Общее руководство по программированию на LINQ")).

Таблица 12.3. Распространенные операции запросов LINQ

Операции запросов	Описание
`from, in`	Используются для определения основы любого выражения LINQ, которая позволяет извлекать подмножество данных из подходящего контейнера
`where`	Применяется для определения ограничений относительно того, какие элементы должны извлекаться из контейнера
`select`	Используется для выборки последовательности из контейнера
`join, on, equals, into`	Выполняют соединения на основе указанного ключа. Помните, что эти "соединения" ничего не обязаны делать с данными в реляционной базе данных
`orderby, ascending, descending`	Позволяют упорядочить результирующий набор по возрастанию или убыванию
`group, by`	Выдают подмножество с данными, сгруппированными по указанному значению

В дополнение к неполному списку операций, приведенному в табл. 12.3, класс `System.Linq.Enumerable` предлагает набор методов, которые не имеют прямого сокращенного обозначения в виде операций запросов C#, а доступны как расширяющие методы. Эти обобщенные методы можно вызывать для трансформации результирующего набора разными способами (`Reverse<>()`, `ToArray<>()`, `ToList<>()` и т.д.). Некоторые из них применяются для извлечения одиночных элементов из результирующего набора, другие выполняют разнообразные операции над множествами (`Distinct<>()`, `Union<>()`, `Intersect<>()` и т.п.), а еще одни агрегируют результаты (`Count<>()`, `Sum<>()`, `Min<>()`, `Max<>()` и т.д.).

Чтобы приступить к исследованию более замысловатых запросов LINQ, создадим новый проект консольного приложения по имени `FunWithLinqExpressions`. Затем определим массив или коллекцию некоторых выборочных данных. В этом проекте мы создадим массив объектов `ProductInfo`, определенный следующим образом:

```
class ProductInfo
{
  public string Name {get; set;} = "";
  public string Description {get; set;} = "";
  public int NumberInStock {get; set;} = 0;
  public override string ToString()
  {
    return string.Format("Name={0}, Description={1}, Number in Stock={2}",
                         Name, Description, NumberInStock);
  }
}
```

Теперь заполним массив объектами ProductInfo внутри метода Main():

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Query Expressions *****\n");

  // Этот массив будет основой для тестирования...
  ProductInfo[] itemsInStock = new[] {
    new ProductInfo{ Name = "Mac's Coffee",
                     Description = "Coffee with TEETH",
                     NumberInStock = 24},
    new ProductInfo{ Name = "Milk Maid Milk",
                     Description = "Milk cow's love",
                     NumberInStock = 100},
    new ProductInfo{ Name = "Pure Silk Tofu",
                     Description = "Bland as Possible",
                     NumberInStock = 120},
    new ProductInfo{ Name = "Cruchy Pops",
                     Description = "Cheezy, peppery goodness",
                     NumberInStock = 2},
    new ProductInfo{ Name = "RipOff Water",
                     Description = "From the tap to your wallet",
                     NumberInStock = 100},
    new ProductInfo{ Name = "Classic Valpo Pizza",
                     Description = "Everyone loves pizza!",
                     NumberInStock = 73}
    };
  // Здесь мы будем вызывать разнообразные методы!
  Console.ReadLine();
}
```

Базовый синтаксис выборки

Поскольку синтаксическая корректность выражения запроса LINQ проверяется на этапе компиляции, вы должны помнить, что порядок следования операций критически важен. В простейшем виде каждый запрос LINQ строится с использованием операций from, in и select. Вот базовый шаблон, который нужно соблюдать:

```
var результат =
     from сопоставляемыйЭлемент in контейнер select сопоставляемыйЭлемент;
```

Элемент после операции from представляет элемент, соответствующий критерию запроса LINQ; именовать его можно по своему усмотрению. Элемент после операции in представляет контейнер данных, в котором производится поиск (массив, коллекция, документ XML и т.д.).

Рассмотрим простой запрос, не делающий ничего кроме извлечения каждого элемента контейнера (по поведению похожий на SQL-оператор SELECT * в базе данных):

```
static void SelectEverything(ProductInfo[] products)
{
  // Получить все!
  Console.WriteLine("All product details:");
  var allProducts = from p in products select p;
  foreach (var prod in allProducts)
  {
    Console.WriteLine(prod.ToString());
  }
}
```

По правде говоря, это выражение запроса не особенно полезно, т.к. выдаст подмножество, идентичное содержимому входного параметра. При желании можно извлечь только значения Name каждого товара, применив следующий синтаксис выборки:

```
static void ListProductNames(ProductInfo[] products)
{
  // Теперь получить только наименования товаров.
  Console.WriteLine("Only product names:");
  var names = from p in products select p.Name;

  foreach (var n in names)
  {
    Console.WriteLine("Name: {0}", n);
  }
}
```

Получение подмножества данных

Чтобы получить определенное подмножество из контейнера, можно использовать операцию where. Общий шаблон запроса становится таким:

```
var результат =
        from элемент in контейнер where булевскоеВыражение select элемент;
```

Обратите внимание, что операция where ожидает выражения, результатом вычисления которого является булевское значение. Например, чтобы извлечь из аргумента ProductInfo[] только те товарные позиции, складские запасы которых составляют более 25 единиц, можно написать следующий код:

```
static void GetOverstock(ProductInfo[] products)
{
  Console.WriteLine("The overstock items!");

  // Получить только товары со складским запасом более 25 единиц.
  var overstock = from p in products where p.NumberInStock > 25 select p;

  foreach (ProductInfo c in overstock)
  {
    Console.WriteLine(c.ToString());
  }
}
```

Как было показано ранее в главе, при указании конструкции where разрешено применять любые операции C# для построения сложных выражений. Например, вспомните запрос, который извлекал только автомобили марки BMW, движущиеся со скоростью выше 100 миль в час:

```
// Получить автомобили BMW, движущиеся со скоростью свыше 100 миль в час.
var onlyFastBMWs = from c in myCars
                   where c.Make == "BMW" && c.Speed >= 100 select c;
foreach (Car c in onlyFastBMWs)
{
  Console.WriteLine("{0} is going {1} MPH", c.PetName, c.Speed);
}
```

Проецирование новых типов данных

Новые формы данных также можно проецировать из существующего источника данных. Давайте предположим, что необходимо принять входной параметр ProductInfo[] и получить результирующий набор, который учитывает только имя и описание каждого

товара. Для этого понадобится определить оператор select, динамически выдающий новый анонимный тип:

```
static void GetNamesAndDescriptions(ProductInfo[] products)
{
  Console.WriteLine("Names and Descriptions:");
  var nameDesc = from p in products select new { p.Name, p.Description };
  foreach (var item in nameDesc)
  {
    // Можно было бы также использовать свойства Name и Description напрямую.
    Console.WriteLine(item.ToString());
  }
}
```

Не забывайте, что когда запрос LINQ использует проекцию, нет никакого способа узнать лежащий в ее основе тип данных, т.к. он определяется на этапе компиляции. В этих случаях ключевое слово var является обязательным. Кроме того, вспомните о невозможности создания методов с неявно типизированными возвращаемыми значениями. Таким образом, следующий метод не скомпилируется:

```
static var GetProjectedSubset(ProductInfo[] products)
{
  var nameDesc = from p in products select new { p.Name, p.Description };
  return nameDesc;  // Так поступать нельзя!
}
```

Если необходимо возвратить спроецированные данные вызывающему коду, то один из подходов предусматривает трансформацию результата запроса в объект System.Array из .NET с применением расширяющего метода ToArray(). Следовательно, модифицировав выражение запроса, как показано ниже:

```
// Теперь возвращаемым значением является объект Array.
static Array GetProjectedSubset(ProductInfo[] products)
{
  var nameDesc = from p in products select new { p.Name, p.Description };
  // Отобразить набор анонимных объектов на объект Array.
  return nameDesc.ToArray();
}
```

метод GetProjectedSubset() можно вызвать в методе Main() и обработать возвращенные им данные:

```
Array objs = GetProjectedSubset(itemsInStock);
foreach (object o in objs)
{
  Console.WriteLine(o); // Вызывает метод ToString() на каждом анонимном объекте.
}
```

Как видите, здесь должен использоваться буквальный объект System.Array и невозможно применять синтаксис объявления массива C#, учитывая, что лежащий в основе проекции тип неизвестен, поскольку речь идет об анонимном классе, который сгенерирован компилятором. Кроме того, параметр типа для обобщенного метода ToArry<>T() не указывается, потому что он тоже не известен вплоть до этапа компиляции.

Очевидная проблема связана с утратой строгой типизации, т.к. каждый элемент в объекте Array считается относящимся к типу Object. Тем не менее, когда нужно возвратить результирующий набор LINQ, полученный посредством операции проецирования, трансформация данных в тип Array (или другой подходящий контейнер через другие члены типа Enumerable) обязательна.

Подсчет количества с использованием класса Enumerable

При проецировании новых пакетов данных вам может возникнуть необходимость в выяснении количества элементов, возвращаемых внутри последовательности. Для определения числа элементов, которые возвращаются из выражения запроса LINQ, можно применять расширяющий метод Count() класса Enumerable. Например, следующий метод будет искать в локальном массиве все объекты string, которые имеют длину, превышающую шесть символов, и выводить их количество:

```
static void GetCountFromQuery()
{
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                "Fallout 3", "Daxter", "System Shock 2"};
  // Получить количество элементов из запроса.
  int numb =
    (from g in currentVideoGames where g.Length > 6 select g).Count();
  // Вывести количество элементов.
  Console.WriteLine("{0} items honor the LINQ query.", numb);
}
```

Изменение на противоположный порядка следования элементов в результирующих наборах

Изменить порядок следования элементов в результирующем наборе на противоположный довольно легко с помощью расширяющего метода Reverse<T>() класса Enumerable. Например, в показанном далее методе выбираются все элементы из входного параметра ProductInfo[] в обратном порядке:

```
static void ReverseEverything(ProductInfo[] products)
{
  Console.WriteLine("Product in reverse:");
  var allProducts = from p in products select p;
  foreach (var prod in allProducts.Reverse())
  {
    Console.WriteLine(prod.ToString());
  }
}
```

Выражения сортировки

В начальных примерах настоящей главы вы видели, что в выражении запроса может использоваться операция orderby для сортировки элементов в подмножестве по заданному значению. По умолчанию принят порядок по возрастанию, поэтому строки сортируются в алфавитном порядке, числовые значения — от меньшего к большему и т.д. Если вы хотите просматривать результаты в порядке по убыванию, просто включите в выражение запроса операцию descending. Взгляните на следующий метод:

```
static void AlphabetizeProductNames(ProductInfo[] products)
{
  // Получить наименования товаров в алфавитном порядке.
  var subset = from p in products orderby p.Name select p;
  Console.WriteLine("Ordered by Name:");
  foreach (var p in subset)
  {
    Console.WriteLine(p.ToString());
  }
}
```

Хотя порядок по возрастанию является стандартным, свои намерения можно прояснить, явно указав операцию ascending:

```
var subset = from p in products orderby p.Name ascending select p;
```

Для получения элементов в порядке убывания служит операция descending:

```
var subset = from p in products orderby p.Name descending select p;
```

LINQ как лучшее средство построения диаграмм Венна

Класс Enumerable поддерживает набор расширяющих методов, которые позволяют применять два (или более) запроса LINQ в качестве основы для нахождения объединений, разностей, конкатенаций и пересечений данных. Первым мы рассмотрим расширяющий метод Except(). Он возвращает результирующий набор LINQ, содержащий разность между двумя контейнерами, которой в этом случае является значение Yugo:

```
static void DisplayDiff()
{
  List<string> myCars = new List<String> {"Yugo", "Aztec", "BMW"};
  List<string> yourCars = new List<String>{"BMW", "Saab", "Aztec" };

  var carDiff =(from c in myCars select c)
    .Except(from c2 in yourCars select c2);

  Console.WriteLine("Here is what you don't have, but I do:");
  foreach (string s in carDiff)
    Console.WriteLine(s);  // Выводит Yugo.
}
```

Метод Intersect() возвращает результирующий набор, который содержит общие элементы данных в наборе контейнеров. Например, следующий метод возвращает последовательность из Aztec и BMW:

```
static void DisplayIntersection()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

  // Получить общие члены.
  var carIntersect = (from c in myCars select c)
    .Intersect(from c2 in yourCars select c2);

  Console.WriteLine("Here is what we have in common:");
  foreach (string s in carIntersect)
    Console.WriteLine(s);  // Выводит Aztec и BMW.
}
```

Метод Union() возвращает результирующий набор, который включает все члены множества запросов LINQ. Подобно любому объединению, даже если общий член встречается более одного раза, повторяющихся значений в результирующем наборе не будет. Следовательно, показанный ниже метод выведет на консоль значения Yugo, Aztec, BMW и Saab:

```
static void DisplayUnion()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

  // Получить объединение двух контейнеров.
  var carUnion = (from c in myCars select c)
  .Union(from c2 in yourCars select c2);
```

```
  Console.WriteLine("Here is everything:");
  foreach (string s in carUnion)
    Console.WriteLine(s);  // Выводит все общие члены.
}
```

Наконец, расширяющий метод `Concat()` возвращает результирующий набор, который является прямой конкатенацией результирующих наборов LINQ. Например, следующий метод выводит на консоль результаты `Yugo`, `Aztec`, `BMW`, `BMW`, `Saab` и `Aztec`:

```
static void DisplayConcat()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
  var carConcat = (from c in myCars select c)
    .Concat(from c2 in yourCars select c2);

  // Выводит:
  // Yugo Aztec BMW BMW Saab Aztec.
  foreach (string s in carConcat)
    Console.WriteLine(s);
}
```

Устранение дубликатов

При вызове расширяющего метода `Concat()` очень легко получить в результате избыточные элементы, что в некоторых случаях может быть именно тем, что и нужно. Однако в других случаях может понадобиться удалить дублированные элементы данных. Для этого необходимо просто вызвать расширяющий метод `Distinct()`:

```
static void DisplayConcatNoDups()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
  var carConcat = (from c in myCars select c)
    .Concat(from c2 in yourCars select c2);

  // Выводит:
  // Yugo Aztec BMW Saab Aztec.
  foreach (string s in carConcat.Distinct())
    Console.WriteLine(s);
}
```

Операции агрегирования LINQ

Запросы LINQ могут также быть спроектированы для выполнения разнообразных операций агрегирования над результирующим набором. Одним из примеров может служить расширяющий метод `Count()`. Другие возможности включают получение среднего, максимального, минимального или суммы значений с использованием членов `Average()`, `Max()`, `Min()` либо `Sum()` класса `Enumerable`. Вот простой пример:

```
static void AggregateOps()
{
  double[] winterTemps = { 2.0, -21.3, 8, -4, 0, 8.2 };

  // Разнообразные примеры агрегации.
  Console.WriteLine("Max temp: {0}",
    (from t in winterTemps select t).Max()); // выводит максимальную температуру
  Console.WriteLine("Min temp: {0}",
   (from t in winterTemps select t).Min()); // выводит минимальную температуру
```

```
    Console.WriteLine("Average temp: {0}",
     (from t in winterTemps select t).Average()); // выводит среднюю температуру
    Console.WriteLine("Sum of all temps: {0}",
     (from t in winterTemps select t).Sum()); // выводит сумму всех температур
}
```

Эти примеры должны предоставить достаточный объем сведений, чтобы вы освоились с процессом построения выражений запросов LINQ. Хотя существуют дополнительные операции, которые пока еще не рассматривались, вы увидите примеры позже в книге, когда речь пойдет о связанных технологиях LINQ. В завершение вводного экскурса в LINQ оставшиеся материалы главы посвящены подробностям отношений между операциями запросов LINQ и лежащей в основе объектной моделью.

Исходный код. Проект `FunWithLinqExpressions` доступен в подкаталоге `Chapter_12`.

Внутреннее представление операторов запросов LINQ

К настоящему моменту вы уже знакомы с процессом построения выражений запросов с применением разнообразных операций запросов C# (таких как `from`, `in`, `where`, `orderby` и `select`). Вдобавок вы узнали, что определенная функциональность API-интерфейса LINQ to Objects доступна только через вызов расширяющих методов класса `Enumerable`. В действительности компилятор C# транслирует все операции запросов LINQ в вызовы методов класса `Enumerable`.

Огромное количество методов класса `Enumerable` прототипированы для приема делегатов в качестве аргументов. В частности, многие методы требуют обобщенный делегат по имени `Func<>`, который был представлен во время рассмотрения обобщенных делегатов в главе 9. Взгляните на метод `Where()` класса `Enumerable`, который вызывается автоматически в случае использования операции `where`:

```
// Перегруженные версии метода Enumerable.Where<T>().
// Обратите внимание, что второй параметр имеет тип System.Func<>.
public static IEnumerable<TSource> Where<TSource>(this
IEnumerable<TSource> source,
    System.Func<TSource,int,bool> predicate)

public static IEnumerable<TSource> Where<TSource>(this
IEnumerable<TSource> source,
    System.Func<TSource,bool> predicate)
```

Делегат `Func<>` представляет шаблон фиксированной функции с набором до 16 аргументов и возвращаемым значением. Если вы исследуете этот тип в браузере объектов Visual Studio, то заметите разнообразные формы делегата `Func<>`. Например:

```
// Различные формы делегата Func<>.
public delegate TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2,
  T3 arg3, T4 arg4)
public delegate TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)
public delegate TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)
public delegate TResult Func<T1,TResult>(T1 arg1)
public delegate TResult Func<TResult>()
```

Учитывая, что многие члены класса `System.Linq.Enumerable` при вызове ожидают получить делегат, можно вручную создать новый тип делегата и написать для него необходимые целевые методы, применить анонимный метод C# или определить подходя-

щее лямбда-выражение. Независимо от выбранного подхода конечный результат будет одним и тем же. Хотя использование операций запросов LINQ является, несомненно, самым простым способом построения запросов LINQ, давайте взглянем на все возможные подходы, чтобы увидеть связь между операциями запросов C# и лежащим в основе типом Enumerable.

Построение выражений запросов с применением операций запросов

Для начала создадим новый проект консольного приложения по имени LinqUsingEnumerable. В классе Program будут определены статические вспомогательные методы (вызываемые из Main()) для иллюстрации разнообразных подходов к построению выражений запросов LINQ.

Первый метод, QueryStringsWithOperators(), предлагает наиболее прямолинейный способ создания выражений запросов и идентичен коду примера LinqOverArray, который приводился ранее в главе:

```
static void QueryStringWithOperators()
{
  Console.WriteLine("***** Using Query Operators *****");
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};
  var subset = from game in currentVideoGames
               where game.Contains(" ") orderby game select game;
  foreach (string s in subset)
    Console.WriteLine("Item: {0}", s);
}
```

Очевидное преимущество использования операций запросов C# при построении выражений запросов заключается в том, что делегаты Func<> и вызовы методов Enumerable остаются вне поля зрения и внимания, т.к. выполнение необходимой трансляции возлагается на компилятор C#. Бесспорно, создание выражений LINQ с применением различных операций запросов (from, in, where или orderby) является наиболее распространенным и простым подходом.

Построение выражений запросов с использованием типа Enumerable и лямбда-выражений

Имейте в виду, что применяемые здесь операции запросов LINQ представляют собой сокращенные версии вызова расширяющих методов, определенных в типе Enumerable. Рассмотрим показанный ниже метод QueryStringsWithEnumerableAndLambdas(), который обрабатывает локальный массив строк, но на этот раз в нем напрямую используются расширяющие методы Enumerable:

```
static void QueryStringsWithEnumerableAndLambdas()
{
  Console.WriteLine("***** Using Enumerable / Lambda Expressions *****");
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};
  // Построить выражение запроса с использованием расширяющих методов,
  // предоставленных типу Array через тип Enumerable.
  var subset = currentVideoGames.Where(game => game.Contains(" "))
    .OrderBy(game => game).Select(game => game);
```

```
  // Вывести результаты.
  foreach (var game in subset)
    Console.WriteLine("Item: {0}", game);
  Console.WriteLine();
}
```

Сначала вызывается расширяющий метод `Where()` на строковом массиве `currentVideoGames`. Вспомните, что класс `Array` получает этот метод от класса `Enumerable`. Метод `Enumerable.Where()` требует параметра типа делегата `System.Func<T1, TResult>`. Первый параметр типа этого делегата представляет совместимые с интерфейсом `IEnumerable<T>` данные для обработки (массив строк в рассматриваемом случае), а второй — результирующие данные метода, которые получаются от единственного оператора, вставленного в лямбда-выражение.

Возвращаемое значение метода `Where()` в этом примере кода скрыто от глаз, но "за кулисами" работа происходит с типом `OrderedEnumerable`. На объекте данного типа вызывается обобщенный метод `OrderBy()`, который также принимает параметр типа делегата `Func<>`. В этот раз производится передача всех элементов по очереди посредством подходящего лямбда-выражения. Результатом вызова `OrderBy()` является новая упорядоченная последовательность первоначальных данных.

И, наконец, осуществляется вызов метода `Select()` на последовательности, возвращенной `OrderBy()`, который в итоге дает окончательный набор данных, сохраняемый в неявно типизированной переменной по имени `subset`.

Конечно, такой "длинный" запрос LINQ несколько сложнее для восприятия, чем предыдущий пример с операциями запросов LINQ. Часть этой сложности, без сомнения, связана с объединением в цепочку вызовов посредством операции точки. Вот тот же самый запрос с выделением каждого шага в отдельный фрагмент (разбивать запрос на части можно разными способами):

```
static void QueryStringsWithEnumerableAndLambdas2()
{
  Console.WriteLine("***** Using Enumerable / Lambda Expressions *****");
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // Разбить на части.
  var gamesWithSpaces = currentVideoGames.Where(game => game.Contains(" "));
  var orderedGames = gamesWithSpaces.OrderBy(game => game);
  var subset = orderedGames.Select(game => game);

  foreach (var game in subset)
    Console.WriteLine("Item: {0}", game);
  Console.WriteLine();
}
```

Как видите, построение выражения запроса LINQ с применением методов класса `Enumerable` напрямую приводит к намного более многословному запросу, чем в случае использования операций запросов C#. Кроме того, поскольку методы `Enumerable` требуют передачи делегатов в качестве параметров, обычно необходимо писать лямбда-выражения, чтобы обеспечить обработку входных данных внутренней целью делегата.

Построение выражений запросов с использованием типа `Enumerable` и анонимных методов

Учитывая, что лямбда-выражения C# — это просто сокращенный способ работы с анонимными методами, рассмотрим третье выражение запроса внутри вспомогательного метода `QueryStringsWithAnonymousMethods()`:

```
static void QueryStringsWithAnonymousMethods()
{
  Console.WriteLine("***** Using Anonymous Methods *****");

  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // Построить необходимые делегаты Func<> с использованием анонимных методов.
  Func<string, bool> searchFilter =
     delegate(string game) { return game.Contains(" "); };
  Func<string, string> itemToProcess = delegate(string s) { return s; };

  // Передать делегаты в методы класса Enumerable.
  var subset = currentVideoGames.Where(searchFilter)
     .OrderBy(itemToProcess).Select(itemToProcess);

  // Вывести результаты.
  foreach (var game in subset)
    Console.WriteLine("Item: {0}", game);
  Console.WriteLine();
}
```

Такой вариант выражения запроса оказывается еще более многословным из-за создания вручную делегатов `Func<>`, применяемых методами `Where()`, `OrderBy()` и `Select()` класса `Enumerable`. Положительная сторона данного подхода связана с тем, что синтаксис анонимных методов позволяет заключить всю обработку, выполняемую делегатами, в единственное определение метода. Тем не менее, этот метод функционально эквивалентен методам `QueryStringsWithEnumerableAndLambdas()` и `QueryStringsWithOperators()`, созданным в предшествующих разделах.

Построение выражений запросов с использованием типа `Enumerable` и низкоуровневых делегатов

Наконец, если вы хотите строить выражение запроса с применением *по-настоящему многословного подхода*, то можете отказаться от использования синтаксиса лямбда-выражений и анонимных методов и напрямую создавать цели делегатов для каждого типа `Func<>`. Ниже показана финальная версия выражения запроса, смоделированная внутри нового типа класса по имени `VeryComplexQueryExpression`:

```
class VeryComplexQueryExpression
{
  public static void QueryStringsWithRawDelegates()
  {
    Console.WriteLine("***** Using Raw Delegates *****");

    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
      "Fallout 3", "Daxter", "System Shock 2"};

    // Построить необходимые делегаты Func<>.
    Func<string, bool> searchFilter = new Func<string, bool>(Filter);
    Func<string, string> itemToProcess = new Func<string,string>(ProcessItem);

    // Передать делегаты в методы класса Enumerable.
    var subset = currentVideoGames
      .Where(searchFilter).OrderBy(itemToProcess).Select(itemToProcess);

    // Вывести результаты.
    foreach (var game in subset)
      Console.WriteLine("Item: {0}", game);
    Console.WriteLine();
  }
```

```
  // Цели делегатов.
  public static bool Filter(string game) {return game.Contains(" ");}
  public static string ProcessItem(string game) { return game; }
}
```

Чтобы протестировать эту версию логики обработки строк, понадобится вызвать метод `QueryStringsWithRawDelegates()` внутри метода `Main()` класса `Program`:

```
VeryComplexQueryExpression.QueryStringsWithRawDelegates();
```

Если теперь запустить приложение, чтобы опробовать все возможные подходы, вывод окажется идентичным независимо от выбранного пути. Запомните перечисленные ниже моменты относительно выражений запросов и их внутреннего представления.

- Выражения запросов создаются с применением разнообразных операций запросов C#.
- Операции запросов — это просто сокращенное обозначение для вызова расширяющих методов, определенных в типе `System.Linq.Enumerable`.
- Многие методы класса `Enumerable` требуют передачи делегатов (в частности, `Func<>`) в качестве параметров.
- Любой метод, ожидающий параметра типа делегата, может принимать вместо него лямбда-выражение.
- Лямбда-выражения являются просто замаскированными анонимными методами (и значительно улучшают читабельность).
- Анонимные методы представляют собой сокращенные обозначения для размещения экземпляра низкоуровневого делегата и ручного построения целевого метода делегата.

Хотя здесь мы погрузились в детали чуть глубже, чем возможно хотелось, приведенное обсуждение должно поспособствовать пониманию того, что фактически делают "за кулисами" дружественные к пользователю операции запросов C#.

Исходный код. Проект `LinqUsingEnumerable` доступен в подкаталоге `Chapter_12`.

Резюме

LINQ — это набор взаимосвязанных технологий, которые были разработаны для предоставления единого и симметричного стиля взаимодействия с данными несходных форм. Как объяснялось в главе, LINQ может взаимодействовать с любым типом, реализующим интерфейс `IEnumerable<T>`, в том числе с простыми массивами, а также с обобщенными и необобщенными коллекциями данных.

Было показано, что работа с технологиями LINQ обеспечивается несколькими средствами языка C#. Например, учитывая тот факт, что выражения запросов LINQ могут возвращать любое количество результирующих наборов, для представления лежащего в основе типа данных принято использовать ключевое слово `var`. Кроме того, для построения функциональных и компактных запросов LINQ могут применяться лямбда-выражения, синтаксис инициализации объектов и анонимные типы.

Более важно то, что вы увидели, что операции запросов C# LINQ в действительности являются просто сокращенными обозначениями для обращения к статическим членам типа `System.Linq.Enumerable`. Вы узнали, что большинство членов класса `Enumerable` оперируют с типами делегатов `Func<T>`, которые могут принимать адреса существующих методов, анонимные методы или лямбда-выражения для выполнения запроса.

ГЛАВА 13

Время существования объектов

К этому моменту вы уже научились создавать специальные типы классов в C#. Теперь вы узнаете, каким образом среда CLR управляет размещенными экземплярами классов (т.е. объектами) посредством *сборки мусора*. Программистам на C# никогда не приходится непосредственно удалять управляемый объект из памяти (вспомните, что в языке C# даже нет ключевого слова вроде `delete`). Вместо этого объекты .NET размещаются в области памяти, которая называется *управляемой кучей*, где они автоматически уничтожаются сборщиком мусора "когда-нибудь в будущем".

После изложения основных деталей, касающихся процесса сборки мусора, будет показано, как программно взаимодействовать со сборщиком мусора, используя класс `System.GC` (что в большинстве проектов .NET обычно не требуется). Мы рассмотрим, как с применением виртуального метода `System.Object.Finalize()` и интерфейса `IDisposable` строить классы, которые своевременно и предсказуемо освобождают внутренние *неуправляемые ресурсы*.

Кроме того, будут описаны некоторые функциональные возможности сборщика мусора, появившиеся в версии .NET 4.0, включая фоновую сборку мусора и ленивое (отложенное) создание объектов с использованием обобщенного класса `System.Lazy<>`. После освоения материалов настоящей главы вы должны хорошо понимать, каким образом среда CLR управляет объектами .NET.

Классы, объекты и ссылки

Прежде чем приступить к исследованию главных тем главы, важно дополнительно прояснить отличие между классами, объектами и ссылочными переменными. Вспомните, что класс — это всего лишь модель, которая описывает то, как экземпляр такого типа будет выглядеть и вести себя в памяти. Разумеется, классы определяются внутри файлов кода (которым по соглашению назначается расширение `*.cs`). Взгляните на следующий простой класс `Car`, определенный в новом проекте консольного приложения C# по имени `SimpleGC`:

```
// Car.cs
public class Car
{
  public int CurrentSpeed {get; set;}
  public string PetName {get; set;}

  public Car(){}
```

```
  public Car(string name, int speed)
  {
    PetName = name;
    CurrentSpeed = speed;
  }
  public override string ToString()
  {
    return string.Format("{0} is going {1} MPH",
      PetName, CurrentSpeed);
  }
}
```

После того как класс определен, можно размещать в памяти любое количество его объектов с применением ключевого слова `new` языка C#. Однако следует иметь в виду, что ключевое слово `new` возвращает *ссылку* на объект в куче, а не действительный объект. Если ссылочная переменная объявляется как локальная переменная в области действия метода, то она сохраняется в стеке для дальнейшего использования внутри приложения. Для доступа к членам объекта необходимо применять к сохраненной ссылке операцию точки C#:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** GC Basics *****");
    // Создать новый объект Car в управляемой куче.
    // Возвращается ссылка на этот объект (refToMyCar).
    Car refToMyCar = new Car("Zippy", 50);
    // Операция точки (.) используется для обращения к членам
    // объекта с применением ссылочной переменной.
    Console.WriteLine(refToMyCar.ToString());
    Console.ReadLine();
  }
}
```

На рис. 13.1 показаны отношения между классами, объектами и ссылками.

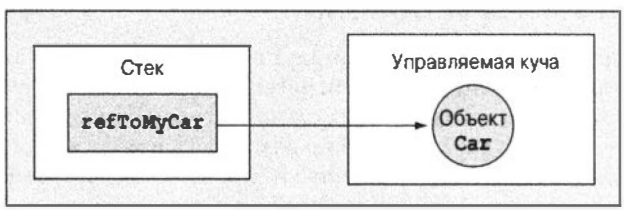


Рис. 13.1. Ссылки на объекты в управляемой куче

На заметку! Вспомните из главы 4, что структуры являются *типами значений*, которые всегда размещаются прямо в стеке и никогда не попадают в управляемую кучу .NET. Размещение в куче происходит только при создании экземпляров классов.

Базовые сведения о времени жизни объектов

При создании приложений C# корректно допускать, что исполняющая среда .NET (также известная как CLR) позаботится об управляемой куче без вашего прямого вмешательства. В действительности "золотое правило" по управлению памятью в .NET выглядит просто.

Правило. Размещайте экземпляр класса в управляемой куче с использованием ключевого слова `new` и забывайте о нем.

После создания объект будет автоматически удален сборщиком мусора, когда в нем отпадет необходимость. Конечно, возникает вполне закономерный вопрос о том, каким образом сборщик мусора выясняет, что объект больше не нужен? Краткий (т.е. неполный) ответ можно сформулировать так: сборщик мусора удаляет объект из кучи, только если он является *недостижимым* ни в одной части кодовой базы. Добавим в класс `Program` метод, который размещает в памяти локальный объект `Car`:

```
static void MakeACar()
{
  // Если myCar - единственная ссылка на объект Car, то после
  // завершения этого метода объект Car *может* быть уничтожен.
  Car myCar = new Car();
}
```

Обратите внимание, что ссылка на объект `Car` (`myCar`) была создана непосредственно внутри метода `MakeACar()` и не передавалась за пределы определяющей области действия (через возвращаемое значение или параметр `ref/out`). Таким образом, после завершения данного метода ссылка `myCar` становится недостижимой, и объект `Car` теперь является кандидатом на удаление сборщиком мусора. Тем не менее, важно понимать, что восстановление памяти, занимаемой этим объектом, немедленно после завершения метода `MakeACar()` гарантировать нельзя. В данный момент можно предполагать лишь то, что когда среда CLR выполнит следующую сборку мусора, объект `myCar` может быть безопасно уничтожен.

Как вы наверняка обнаружите, программирование в среде со сборкой мусора значительно облегчает разработку приложений. По контрасту с этим программистам на языке C++ хорошо известно, что если они не позаботятся о ручном удалении размещенных в куче объектов, то утечки памяти не заставят себя долго ждать. На самом деле отслеживание утечек памяти — один из требующих самых больших затрат времени (и утомительных) аспектов программирования в неуправляемых средах. За счет того, что сборщику мусора разрешено взять на себя заботу об уничтожении объектов, обязанности по управлению памятью перекладываются с программистов на среду CLR.

Код CIL для ключевого слова `new`

Когда компилятор C# сталкивается с ключевым словом `new`, он вставляет в реализацию метода инструкцию `newobj` языка CIL. Если вы скомпилируете текущий пример кода и заглянете в полученную сборку с помощью утилиты `ildasm.exe`, то найдете внутри метода `MakeACar()` следующие операторы CIL:

```
.method private hidebysig static void MakeACar() cil managed
{
  // Code size 8 (0x8)
  .maxstack 1
  .locals init ([0] class SimpleGC.Car myCar)
  IL_0000: nop
```

```
  IL_0001: newobj instance void SimpleGC.Car::.ctor()
  IL_0006: stloc.0
  IL_0007: ret
} // end of method Program::MakeACar
```

Прежде чем ознакомиться с точными правилами, которые определяют момент, когда объект должен удаляться из управляемой кучи, давайте более подробно рассмотрим роль CIL-инструкции `newobj`. Первым делом важно понимать, что управляемая куча представляет собой нечто большее, чем просто произвольную область памяти, к которой CLR имеет доступ. Сборщик мусора .NET "убирает" кучу довольно тщательно, причем при необходимости он даже сжимает пустые блоки памяти в целях оптимизации.

Для содействия в его усилиях в управляемой куче поддерживается указатель (обычно называемый *указателем на следующий объект* или *указателем на новый объект*), который идентифицирует точное местоположение, куда будет помещен следующий объект. Таким образом, инструкция `newobj` заставляет среду CLR выполнить перечисленные ниже основные операции.

1. Вычислить общий объем памяти, требуемой для размещения объекта (включая память, необходимую для членов данных и базовых классов).
2. Выяснить, действительно ли в управляемой куче имеется достаточно пространства для сохранения размещаемого объекта. Если места хватает, то указанный конструктор вызывается, и вызывающий код в конечном итоге получает ссылку на новый объект в памяти, адрес которого совпадает с последней позицией указателя на следующий объект.
3. Наконец, перед возвращением ссылки вызывающему коду продвинуть указатель на следующий объект, чтобы он указывал на следующую доступную область в управляемой куче.

Описанный процесс проиллюстрирован на рис. 13.2.

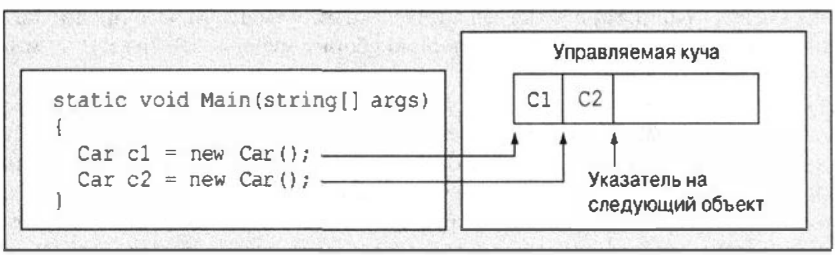


Рис. 13.2. Детали размещения объектов в управляемой куче

По мере интенсивного размещения объектов в приложении пространство в управляемой куче может со временем заполниться. Если при обработке инструкции `newobj` среда CLR определяет, что в управляемой куче недостаточно места для размещения объекта запрашиваемого типа, то она выполнит сборку мусора в попытке освободить память. Соответственно, следующее правило сборки мусора также выглядит довольно простым.

Правило. Если в управляемой куче не хватает пространства для размещения требуемого объекта, то произойдет сборка мусора.

Однако то, *как* происходит сборка мусора, зависит от версии платформы .NET, под управлением которой выполняется приложение. Различия будут описаны далее в главе.

Установка объектных ссылок в null

Программисты на C/C++ часто устанавливают переменные указателей в null, гарантируя тем самым, что они больше не ссылаются на какие-то местоположения в неуправляемой памяти. Учитывая это, вас может интересовать, что происходит в результате установки в null ссылок на объекты в C#. Для примера изменим метод MakeACar() следующим образом:

```
static void MakeACar()
{
  Car myCar = new Car();
  myCar = null;
}
```

Когда ссылки на объекты устанавливаются в null, компилятор C# генерирует код CIL, который гарантирует, что ссылка (myCar в данном примере) больше не указывает на какой-либо объект. Если теперь снова с помощью утилиты ildasm.exe просмотреть код CIL модифицированного метода MakeACar(), то можно обнаружить в нем код операции ldnull (заталкивает значение null в виртуальный стек выполнения), за которым следует код операции stloc.0 (устанавливает для переменной ссылку null):

```
.method private hidebysig static void MakeACar() cil managed
{
  // Code size 10 (0xa)
  .maxstack 1
  .locals init ([0] class SimpleGC.Car myCar)
  IL_0000:  nop
  IL_0001:  newobj instance void SimpleGC.Car::.ctor()
  IL_0006:  stloc.0
  IL_0007:  ldnull
  IL_0008:  stloc.0
  IL_0009:  ret
} // end of method Program::MakeACar
```

Тем не менее, вы должны понимать, что присваивание ссылке значения null ни в коей мере не вынуждает сборщик мусора немедленно запуститься и удалить объект из кучи. Единственное, что при этом достигается — явный разрыв связи между ссылкой и объектом, на который она ранее указывала. Таким образом, установка ссылок в null в C# имеет гораздо меньше последствий, чем в других языках, основанных на C; однако никакого вреда она определенно не причиняет.

Роль корневых элементов приложения

Теперь вернемся к вопросу о том, как сборщик мусора определяет момент, когда объект больше не нужен. Чтобы понять все детали, вы должны ознакомиться с понятием *корневых элементов приложения*. Выражаясь просто, *корневой элемент* — это местоположение в памяти, содержащее ссылку на объект в управляемой куче. Строго говоря, корневой элемент может относиться к любой из следующих категорий:

- ссылки на глобальные объекты (хотя в C# они не допускаются, код CIL разрешает размещать глобальные объекты);
- ссылки на любые статические объекты и статические поля;
- ссылки на локальные объекты внутри кодовой базы приложения;
- ссылки на объектные параметры, передаваемые методу;
- ссылки на объекты, ожидающие финализации (обсуждаются далее в главе);
- любой регистр центрального процессора, который ссылается на объект.

Во время процесса сборки мусора исполняющая среда будет исследовать объекты в управляемой куче с целью выяснения, являются ли они по-прежнему достижимыми (т.е. корневыми) для приложения. Для этого CLR будет строить *граф объектов*, который представляет каждый достижимый объект в куче. Более подробно графы объектов объясняются во время рассмотрения сериализации объектов в главе 20. Пока достаточно знать, что такие графы применяются для документирования всех достижимых объектов. Кроме того, имейте в виду, что сборщик мусора никогда не будет создавать граф для того же самого объекта дважды, избегая необходимости в выполнении утомительного подсчета циклических ссылок, который характерен для программирования COM.

Предположим, что в управляемой куче находится набор объектов с именами A, B, C, D, E, F и G. Во время сборки мусора эти объекты (а также любые внутренние объектные ссылки, которые они могут содержать) будут проверяться на предмет активных корневых элементов. После построения графа недостижимые объекты (пусть ими будут объекты C и F) помечаются как являющиеся мусором. На рис. 13.3 показан возможный граф объектов для только что описанного сценария (линии со стрелками можно читать как "зависит от" или "требует", т.е. E зависит от G и B, A не зависит ни от чего и т.д.).

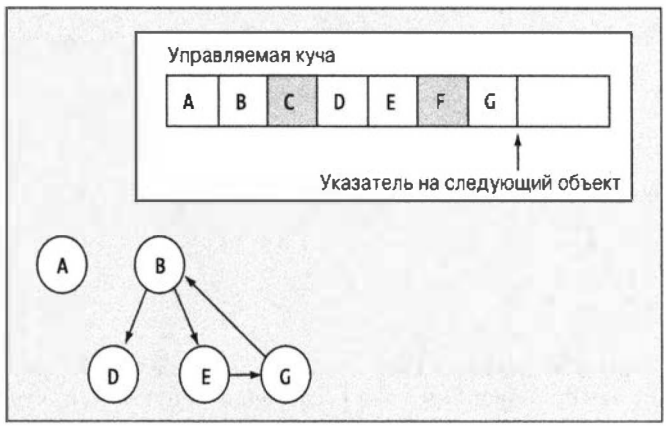


Рис. 13.3. Графы объектов строятся с целью определения объектов, достижимых для корневых элементов приложения

После того как объекты помечены для уничтожения (в данном случае это объекты C и F, т.к. они не учтены в графе объектов), они удаляются из памяти. Оставшееся пространство в куче сжимается, что в свою очередь вынуждает среду CLR модифицировать набор активных корневых элементов приложения (и лежащих в основе указателей), чтобы они ссылались на корректные местоположения в памяти (это делается автоматически и прозрачно). И последнее, но не менее важное действие — указатель на следующий объект перенастраивается так, чтобы указывать на следующую доступную область памяти. Конечный результат описанных изменений представлен на рис. 13.4.

На заметку! Строго говоря, сборщик мусора использует две отдельные кучи, одна из которых предназначена специально для хранения крупных объектов. Во время сборки мусора обращение к этой куче производится менее часто из-за возможного снижения производительности, связанного с перемещением больших объектов. Невзирая на данный факт, вполне безопасно считать управляемую кучу единственной областью памяти.

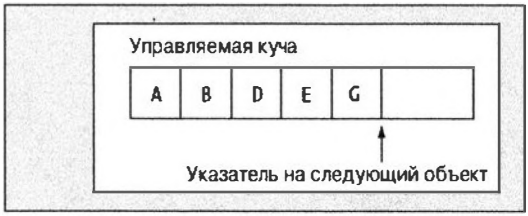


Рис. 13.4. Очищенная и сжатая куча

Понятие поколений объектов

Когда среда CLR пытается найти недостижимые объекты, она не проверяет буквально каждый объект, помещенный в управляемую кучу. Очевидно, это потребовало бы значительного времени, особенно в крупных (т.е. реальных) приложениях.

Для содействия оптимизации процесса каждому объекту в куче назначается специфичное "поколение". Идея, лежащая в основе поколений, проста: чем дольше объект существует в куче, тем выше вероятность того, что он там будет оставаться. Например, класс, который определяет главное окно настольного приложения, будет находиться в памяти вплоть до завершения приложения. В противоположность этому объекты, которые были помещены в кучу только недавно (такие как объект, размещенный внутри области действия метода), скорее всего, довольно быстро станут недостижимыми. Исходя из таких предположений, каждый объект в куче относится к одному из перечисленных ниже поколений.

- *Поколение 0.* Идентифицирует новый размещенный в памяти объект, который еще никогда не помечался как подлежащий сборке мусора.
- *Поколение 1.* Идентифицирует объект, который уже пережил одну сборку мусора (т.е. он был помечен как подлежащий сборке мусора, но не был удален из-за наличия достаточного пространства в куче).
- *Поколение 2.* Идентифицирует объект, которому удалось пережить более одной очистки сборщиком мусора.

На заметку! Поколения 0 и 1 называются *эфемерными* (недолговечными). В следующем разделе будет показано, что процесс сборки мусора трактует эфемерные поколения по-разному.

Первыми сборщик мусора будет исследовать все объекты, относящиеся к поколению 0. Если пометка и удаление (или освобождение) этих объектов дает в результате требуемый объем свободной памяти, то любые уцелевшие объекты повышаются до поколения 1. Чтобы увидеть, каким образом поколение объекта влияет на процесс сборки мусора, взгляните на рис. 13.5, где схематически показано, как набор уцелевших объектов поколения 0 (A, B и E) повышается до следующего поколения после восстановления требуемого объема памяти.

Если все объекты поколения 0 проверены, но по-прежнему требуется дополнительная память, начинают исследоваться на предмет достижимости и подвергаться сборке мусора объекты поколения 1. Уцелевшие объекты поколения 1 повышаются до поколения 2. Если же сборщику мусора *все еще* требуется дополнительная память, то начинают проверяться объекты поколения 2. На этом этапе объекты поколения 2, которым удается пережить сборку мусора, остаются объектами того же поколения 2, учитывая заранее определенный верхний предел поколений объектов. В заключение следует отметить, что за счет назначения объектам в куче определенного поколения более новые объекты (такие как локальные переменные) будут удаляться быстрее, в то время как более старые (наподобие главного окна приложения) будут существовать дольше.

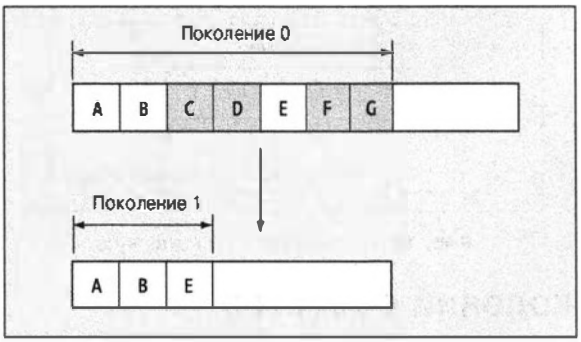


Рис. 13.5. Объекты поколения 0, которые уцелели после сборки мусора, повышаются до поколения 1

Параллельная сборка мусора до версии .NET 4.0

До выхода версии .NET 4.0 исполняющая среда производила очистку неиспользуемых объектов с применением приема, который назывался *параллельной сборкой мусора*. В рамках этой модели, когда происходила сборка мусора для любых объектов поколения 0 или поколения 1 (вспомните, что они являются *эфемерными поколениями*), сборщик мусора временно приостанавливал все активные *потоки* внутри текущего процесса, гарантируя тем самым, что приложение не будет обращаться к управляемой куче во время протекания процесса сборки.

Тема потоков раскрывается в главе 19; пока что рассматривайте поток просто как путь выполнения внутри функционирующей программы. После завершения цикла сборки мусора приостановленным потокам разрешалось продолжить свою работу. К счастью, сборщик мусора в .NET 3.5 (и предшествующих версиях) был хорошо оптимизирован и потому такие короткие перерывы в работе приложения были заметны редко (если вообще когда-либо).

В качестве оптимизации параллельная сборка мусора позволяла производить очистку объектов, которые не были обнаружены в одном из эфемерных поколений, в отдельном потоке. Это сокращало (но не устраняло) необходимость в приостановке активных потоков исполняющей средой .NET. Более того, параллельная сборка мусора разрешала программам продолжать размещение объектов в куче во время проведения сборки мусора для объектов неэфемерных поколений.

Фоновая сборка мусора в .NET 4.0 и последующих версиях

Начиная с версии .NET 4.0 сборщик мусора способен решать вопрос с приостановкой потоков при очистке объектов в управляемой куче, используя *фоновую сборку мусора*. Несмотря на название приема, это вовсе не означает, что вся сборка мусора теперь происходит в дополнительных фоновых потоках выполнения. На самом деле, если фоновая сборка мусора производится для объектов, принадлежащих к неэфемерному поколению, то исполняющая среда .NET теперь может выполнять сборку мусора в отношении объектов эфемерных поколений внутри отдельного фонового потока.

В качестве связанного замечания: механизм сборки мусора в .NET 4.0 и последующих версиях был усовершенствован с целью дальнейшего сокращения времени приостановки заданного потока, которая связана со сборкой мусора. Конечным результатом таких изменений стало то, что процесс очистки неиспользуемых объектов поколения 0

или поколения 1 был оптимизирован и позволяет обеспечить более высокую производительность для приложений (что действительно важно для систем реального времени, которые требуют небольшие и предсказуемые перерывы на сборку мусора).

Тем не менее, важно понимать, что ввод новой модели сборки мусора совершенно не повлиял на способ построения приложений .NET. С практической точки зрения вы можете просто разрешить сборщику мусора .NET выполнять свою работу без непосредственного вмешательства с вашей стороны (и радоваться тому, что разработчики в Microsoft продолжают улучшать процесс сборки мусора в прозрачной манере).

Тип System.GC

Сборка mscorlib.dll предлагает класс по имени System.GC, который позволяет программно взаимодействовать со сборщиком мусора с применением набора статических членов. Имейте в виду, что необходимость в прямом взаимодействии с этим классом внутри разрабатываемого кода возникает редко (если вообще возникает). Обычно единственной ситуацией, когда будут использоваться члены System.GC, является создание классов, которые внутренне работают с *неуправляемыми ресурсами*. Например, может строиться класс, в котором присутствуют вызовы API-интерфейса Windows, основанного на C, с применением протокола обращения к платформе .NET, или какая-то низкоуровневая и сложная логика взаимодействия с COM. В табл. 13.1 приведено краткое описание некоторых наиболее интересных членов класса System.GC (полные сведения можно найти в документации .NET Framework SDK).

Таблица 13.1. Избранные члены типа System.GC

Члены System.GC	Описание
AddMemoryPressure(), RemoveMemoryPressure()	Позволяют указывать числовое значение, которое представляет "уровень срочности" (или давление) вызывающего объекта относительно процесса сборки мусора. Имейте в виду, что эти методы должны изменять уровень давления *в тандеме*; следовательно, нельзя удалять более высокое значение давления, чем то, которое было добавлено
Collect()	Заставляет сборщик мусора выполнить сборку мусора. Этот метод перегружен для указания поколения, подлежащего сборке, а также режима сборки (посредством перечисления GCCollectionMode)
CollectionCount()	Возвращает числовое значение, которое показывает, сколько раз производилась сборка мусора для заданного поколения
GetGeneration()	Возвращает поколение, к которому относится объект в текущий момент
GetTotalMemory()	Возвращает оценочный объем памяти (в байтах), выделенной в управляемой куче в текущий момент. Булевский параметр указывает, должен ли вызов дождаться выполнения сборки мусора перед возвращением
MaxGeneration	Возвращает максимальное количество поколений, поддерживаемое целевой системой. Начиная с версии .NET 4.0 есть три возможных поколения: 0, 1 и 2
SuppressFinalize()	Устанавливает флаг, который указывает, что заданный объект не должен вызывать свой метод Finalize()
WaitForPendingFinalizers()	Приостанавливает текущий поток до тех пор, пока не будут финализированы все финализируемые объекты. Обычно вызывается сразу после вызова метода GC.Collect()

Чтобы проиллюстрировать использование System.GC для получения разнообразных деталей, связанных со сборкой мусора, создадим следующий метод Main(), в котором применяются некоторые члены System.GC:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with System.GC *****");

  // Вывести оценочное количество байтов, выделенных в куче.
  Console.WriteLine("Estimated bytes on heap: {0}",
    GC.GetTotalMemory(false));

  // Значения MaxGeneration начинаются с 0, поэтому при выводе добавить 1.
  Console.WriteLine("This OS has {0} object generations.\n",
    (GC.MaxGeneration + 1));

  Car refToMyCar = new Car("Zippy", 100);
  Console.WriteLine(refToMyCar.ToString());

  // Вывести поколение объекта refToMyCar.
  Console.WriteLine("Generation of refToMyCar is: {0}",
    GC.GetGeneration(refToMyCar));
  Console.ReadLine();
}
```

Принудительный запуск сборщика мусора

Повторим еще раз: основное предназначение сборщика мусора .NET заключается в управлении памятью вместо программистов. Однако в некоторых редких обстоятельствах может быть полезно реализовать в коде принудительный запуск сборщика мусора, используя метод GC.Collect(). Взаимодействие с процессом сборки мусора может требоваться в двух ситуациях.

- Приложение собирается войти в блок кода, который не должен прерываться возможной сборкой мусора.
- Приложение только что закончило размещение исключительно большого количества объектов и нужно насколько возможно скоро освободить большой объем выделенной памяти.

Если выясняется, что принудительная проверка сборщиком мусора наличия недостижимых объектов может принести пользу, то следует явно инициировать процесс сборки мусора:

```
static void Main(string[] args)
{
  ...
  // Принудительно запустить сборку мусора и
  // ожидать финализации каждого объекта.
  GC.Collect();
  GC.WaitForPendingFinalizers();
  ...
}
```

Когда сборка мусора запускается вручную, всегда должен вызываться метод GC.WaitForPendingFinalizers(). Благодаря такому подходу можно иметь уверенность в том, что все *финализируемые объекты* (описанные в следующем разделе) получат шанс выполнить любую необходимую очистку перед продолжением работы программы. "За кулисами" метод GC.WaitForPendingFinalizers() приостановит вызывающий поток на время прохождения сборки мусора. Это очень хорошо, т.к. гарантирует невозможность обращения в коде к методам объекта, который в текущий момент уничтожается.

Методу GC.Collect() можно также предоставить числовое значение, идентифицирующее самое старое поколение, для которого будет выполняться сборка мусора. Например, чтобы инструктировать среду CLR о необходимости исследования только объектов поколения 0, можно написать следующий код:

```
static void Main(string[] args)
{
  ...
  // Исследовать только объекты поколения 0.
  GC.Collect(0);
  GC.WaitForPendingFinalizers();
  ...
}
```

Кроме того, методу Collect() можно передать во втором параметре значение перечисления GCCollectionMode для точной настройки способа, которым исполняющая среда должна принудительно инициировать сборку мусора. Ниже показаны значения, определенные этим перечислением:

```
public enum GCCollectionMode
{
  Default,  // Текущим стандартным значением является Forced.
  Forced,   // Указывает исполняющей среде начать сборку мусора немедленно.
  Optimized // Позволяет исполняющей среде выяснить, оптимален ли
            // текущий момент для удаления объектов.
}
```

Как и при любой сборке мусора, в результате вызова GC.Collect() уцелевшие объекты переводятся в более высокие поколения. В целях иллюстрации предположим, что метод Main() модифицирован следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with System.GC *****");

  // Вывести оценочное количество байтов, выделенных в куче.
  Console.WriteLine("Estimated bytes on heap: {0}",
    GC.GetTotalMemory(false));

  // Значения MaxGeneration начинаются с 0.
  Console.WriteLine("This OS has {0} object generations.\n",
    (GC.MaxGeneration + 1));
  Car refToMyCar = new Car("Zippy", 100);
  Console.WriteLine(refToMyCar.ToString());

  // Вывести поколение refToMyCar.
  Console.WriteLine("\nGeneration of refToMyCar is: {0}",
    GC.GetGeneration(refToMyCar));

  // Создать большое количество объектов в целях тестирования.
  object[] tonsOfObjects = new object[50000];
  for (int i = 0; i < 50000; i++)
    tonsOfObjects[i] = new object();

  // Выполнить сборку мусора только для объектов поколения 0.
  GC.Collect(0, GCCollectionMode.Forced);
  GC.WaitForPendingFinalizers();

  // Вывести поколение refToMyCar.
  Console.WriteLine("Generation of refToMyCar is: {0}",
  GC.GetGeneration(refToMyCar));
```

```
    // Посмотреть, существует ли еще tonsOfObjects[9000].
    if (tonsOfObjects[9000] != null)
    {
      Console.WriteLine("Generation of tonsOfObjects[9000] is: {0}",
        GC.GetGeneration(tonsOfObjects[9000]));
    }
    else
      Console.WriteLine("tonsOfObjects[9000] is no longer alive.");

    // Вывести количество проведенных сборок мусора для разных поколений.
    Console.WriteLine("\nGen 0 has been swept {0} times",
      GC.CollectionCount(0));
    Console.WriteLine("Gen 1 has been swept {0} times",
      GC.CollectionCount(1));
    Console.WriteLine("Gen 2 has been swept {0} times",
      GC.CollectionCount(2));
    Console.ReadLine();
  }
```

Здесь в целях тестирования преднамеренно был создан большой массив типа `object` (состоящий из 50 000 элементов). В показанном ниже выводе видно, что хотя в методе `Main()` был сделан только один явный запрос на проведение сборки мусора (посредством метода `GC.Collect()`), среда CLR в фоновом режиме выполнила несколько сборок:

```
***** Fun with System.GC *****
Estimated bytes on heap: 70240
This OS has 3 object generations.

Zippy is going 100 MPH

Generation of refToMyCar is: 0
Generation of refToMyCar is: 1
Generation of tonsOfObjects[9000] is: 1

Gen 0 has been swept 1 times
Gen 1 has been swept 0 times
Gen 2 has been swept 0 times
```

К этому моменту вы должны лучше понимать детали жизненного цикла объектов. В следующем разделе мы продолжим изучение процесса сборки мусора, обратившись к теме создания *финализируемых объектов* и *освобождаемых объектов*. Имейте в виду, что описываемые далее приемы обычно необходимы только при построении классов C#, которые поддерживают внутренние неуправляемые ресурсы.

Исходный код. Проект `SimpleGC` доступен в подкаталоге `Chapter_13`.

Построение финализируемых объектов

В главе 6 вы узнали, что в самом главном базовом классе .NET, `System.Object`, определен виртуальный метод по имени `Finalize()`. В своей стандартной реализации он ничего не делает:

```
// System.Object
public class Object
{
  ...
  protected virtual void Finalize() {}
}
```

За счет переопределения метода `Finalize()` в специальных классах устанавливается специфическое место для выполнения любой логики очистки, необходимой данному типу. Учитывая, что метод `Finalize()` определен как защищенный, вызывать его напрямую из экземпляра класса через операцию точки нельзя. Взамен метод `Finalize()`, если он поддерживается, будет вызываться *сборщиком мусора* перед удалением объекта из памяти.

На заметку! Переопределять метод `Finalize()` в типах структур не допускается. Это вполне логичное ограничение, поскольку структуры являются типами значений, которые изначально никогда не размещаются в куче и, следовательно, никогда не подвергаются сборке мусора. Тем не менее, при создании структуры, которая содержит неуправляемые ресурсы, нуждающиеся в очистке, можно реализовать интерфейс `IDisposable` (вскоре он будет описан).

Разумеется, вызов метода `Finalize()` будет происходить (в итоге) во время "естественной" сборки мусора или в случае ее принудительного запуска внутри кода с помощью `GC.Collect()`. Вдобавок метод финализатора типа будет автоматически вызываться, когда домен, в котором обслуживается приложение, выгружается из памяти. В зависимости от опыта работы с .NET вам уже может быть известно, что домены приложений применяются для обслуживания исполняемой сборки и любых необходимых внешних библиотек кода. Если вы еще не знакомы с этой концепцией .NET, то необходимые сведения будут даны в главе 17. Пока запомните, что при выгрузке домена приложения из памяти среда CLR автоматически вызывает финализаторы всех финализируемых объектов, созданных за время существования домена приложения.

О чем бы ни говорили ваши инстинкты разработчика, подавляющее большинство классов C# не требует написания явной логики очистки или специального финализатора. Причина проста: если в классах используются лишь другие управляемые объекты, то все они, в конечном счете, будут подвергнуты сборке мусора. Единственная ситуация, когда может возникнуть потребность спроектировать класс, способный выполнять после себя очистку, предусматривает работу с *неуправляемыми* ресурсами (такими как низкоуровневые файловые дескрипторы операционной системы, низкоуровневые неуправляемые подключения к базам данных, фрагменты неуправляемой памяти и т.д.). В рамках платформы .NET неуправляемые ресурсы получаются путем прямого обращения к API-интерфейсу операционной системы с применением служб вызова платформы (Platform Invocation Services — PInvoke) или в сложных сценариях взаимодействия с COM. С учетом этого можно сформулировать еще одно правило сборки мусора.

Правило. Единственная серьезная причина для переопределения метода `Finalize()` связана с использованием в классе C# неуправляемых ресурсов через PInvoke или сложные задачи взаимодействия с COM (обычно посредством разнообразных членов типа `System.Runtime.InteropServices.Marshal`). Это объясняется тем, что в таких сценариях производится манипулирование памятью, которой среда CLR управлять не может.

Переопределение метода `System.Object.Finalize()`

В том редком случае, когда строится класс C#, в котором применяются неуправляемые ресурсы, вы вполне очевидно захотите обеспечить предсказуемое освобождение занимаемой памяти. Для примера создадим новый проект консольного приложения C# по имени `SimpleFinalize` и вставим в него класс `MyResourceWrapper`, в котором используется неуправляемый ресурс (каким бы он ни был). Теперь необходимо переопределить метод `Finalize()`. Как ни странно, для этого нельзя применять ключевое слово `override` языка C#:

```
class MyResourceWrapper
{
  // Ошибка на этапе компиляции!
  protected override void Finalize(){ }
}
```

На самом деле для достижения того же самого эффекта используется синтаксис деструктора (подобный C++). Причина такой альтернативной формы переопределения виртуального метода заключается в том, что при обработке синтаксиса финализатора компилятор автоматически добавляет внутрь неявно переопределяемого метода `Finalize()` много обязательных элементов инфраструктуры (как вскоре будет показано).

Финализаторы C# выглядят похожими на конструкторы тем, что именуются идентично классу, в котором определены. Вдобавок они снабжаются префиксом в виде тильды (~). Однако в отличие от конструкторов финализаторы никогда не получают модификатор доступа (они всегда неявно защищенные), не принимают параметров и не могут быть перегружены (в каждом классе допускается только один финализатор).

Ниже приведен специальный финализатор для класса `MyResourceWrapper`, который при вызове выдает звуковой сигнал. Очевидно, этот пример предназначен только для демонстрационных целей. В реальном приложении финализатор будет только освобождать любые неуправляемые ресурсы, и *не* будет взаимодействовать с другими управляемыми объектами, даже с теми, на которые ссылается текущий объект, т.к. нельзя предполагать, что они все еще существуют на момент вызова этого метода `Finalize()` сборщиком мусора.

```
// Переопределить System.Object.Finalize() посредством синтаксиса
финализатора.
class MyResourceWrapper
{
  ~MyResourceWrapper()
  {
    // Очистить неуправляемые ресурсы.
    // Выдать звуковой сигнал при уничтожении (только в целях тестирования).
    Console.Beep();
  }
}
```

Если теперь просмотреть код CIL данного деструктора с помощью утилиты `ildasm.exe`, то обнаружится, что компилятор добавил необходимый код для проверки ошибок. Первым делом операторы внутри области действия метода `Finalize()` помещены в блок `try` (см. главу 7). Связанный с ним блок `finally` гарантирует, что методы `Finalize()` базовых классов будут всегда выполняться независимо от любых исключений, возникших в области `try`.

```
.method family hidebysig virtual instance void
  Finalize() cil managed
{
  // Code size       13 (0xd)
  .maxstack  1
  .try
  {
    IL_0000:  ldc.i4 0x4e20
    IL_0005:  ldc.i4 0x3e8
    IL_000a:  call
    void [mscorlib]System.Console::Beep(int32, int32)
    IL_000f:  nop
    IL_0010:  nop
    IL_0011:  leave.s IL_001b
  }  // end .try
```

```
  finally
  {
    IL_0013:  ldarg.0
    IL_0014:
      call instance void [mscorlib]System.Object::Finalize()
    IL_0019:  nop
    IL_001a:  endfinally
  } // end handler
  IL_001b:  nop
  IL_001c:  ret
} // end of method MyResourceWrapper::Finalize
```

Тестирование класса `MyResourceWrapper` показывает, что звуковой сигнал выдается во время завершения приложения, потому что среда CLR автоматически вызывает финализаторы при выгрузке домена приложения:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Finalizers *****\n");
  Console.WriteLine("Hit the return key to shut down this app");
  Console.WriteLine("and force the GC to invoke Finalize()");
  Console.WriteLine("for finalizable objects created in this AppDomain.");
  // Нажмите клавишу <Enter>, чтобы завершить приложение
  // и заставить сборщик мусора вызвать метод Finalize()
  // для всех финализируемых объектов, которые
  // были созданы в домене этого приложения.
  Console.ReadLine();
  MyResourceWrapper rw = new MyResourceWrapper();
}
```

Исходный код. Проект `SimpleFinalize` доступен в подкаталоге `Chapter_13`.

Подробности процесса финализации

Чтобы не стараться понапрасну, всегда помните о том, что роль метода `Finalize()` состоит в обеспечении того, что объект .NET сможет освободить неуправляемые ресурсы, когда он подвергается сборке мусора. Таким образом, если вы строите класс, в котором неуправляемая память не применяется (общепризнанно самый распространенный случай), то финализация малопригодна. На самом деле по возможности вы должны проектировать свои типы так, чтобы избегать в них поддержки метода `Finalize()` по той простой причине, что финализация занимает время.

При размещении объекта в управляемой куче исполняющая среда автоматически определяет, поддерживает ли он специальный метод `Finalize()`. Если да, то объект помечается как *финализируемый*, а указатель на него сохраняется во внутренней очереди, называемой *очередью финализации*. Очередь финализации — это таблица, обслуживаемая сборщиком мусора, в которой содержатся указатели на все объекты, подлежащие финализации перед удалением из кучи.

Когда сборщик мусора решает, что наступило время высвободить объект из памяти, он просматривает каждую запись в очереди финализации и копирует объект из кучи в еще одну управляемую структуру под названием *таблица объектов, доступных для финализации*. На этой стадии порождается отдельный поток для вызова метода `Finalize()` на каждом объекте из упомянутой таблицы *при следующей сборке мусора*. Итак, действительная финализация объекта требует, по меньшей мере, две сборки мусора.

Подводя итоги, следует отметить, что хотя финализация объекта гарантирует ему возможность освобождения неуправляемых ресурсов, она все равно остается недетерминированной по своей природе, а из-за незаметной дополнительной обработки протекает значительно медленнее.

Построение освобождаемых объектов

Как вы уже видели, финализаторы могут использоваться для освобождения неуправляемых ресурсов при запуске сборщика мусора. Тем не менее, учитывая тот факт, что многие неуправляемые объекты являются "ценными элементами" (вроде низкоуровневых дескрипторов для файлов или подключений к базам данных), зачастую полезно их освобождать как можно раньше, не дожидаясь наступления сборки мусора. В качестве альтернативы переопределению метода `Finalize()` класс может реализовать интерфейс `IDisposable`, в котором определен единственный метод по имени `Dispose()`:

```
public interface IDisposable
{
  void Dispose();
}
```

При реализации интерфейса `IDisposable` предполагается, что когда *пользователь объекта* завершает работу с ним, он вручную вызывает метод `Dispose()` перед тем, как позволить объектной ссылке покинуть область действия. Таким способом объект может производить любую необходимую очистку неуправляемых ресурсов без помещения в очередь финализации и ожидания, пока сборщик мусора запустит логику финализации класса.

На заметку! Интерфейс `IDisposable` может быть реализован и структурами, и классами (в отличие от переопределения метода `Finalize()`, что допускается только для классов), т.к. метод `Dispose()` вызывает пользователь объекта, а не сборщик мусора.

Чтобы проиллюстрировать применение этого интерфейса, создадим новый проект консольного приложения C# по имени `SimpleDispose`. Ниже приведен модифицированный класс `MyResourceWrapper`, который теперь реализует интерфейс `IDisposable` взамен переопределения метода `System.Object.Finalize()`:

```
// Реализация интерфейса IDisposable.
class MyResourceWrapper : IDisposable
{
  // После окончания работы с объектом пользователь
  // объекта должен вызывать этот метод.
  public void Dispose()
  {
    // Очистить неуправляемые ресурсы...

    // Освободить другие освобождаемые объекты, содержащиеся внутри.

    // Только для целей тестирования.
    Console.WriteLine("***** In Dispose! *****");
  }
}
```

Обратите внимание, что метод `Dispose()` отвечает не только за освобождение неуправляемых ресурсов самого типа, но может также вызывать методы `Dispose()` для любых других освобождаемых объектов, которые содержатся внутри типа. В отличие от `Finalize()` в методе `Dispose()` вполне безопасно взаимодействовать с другими управ-

ляемыми объектами. Причина проста: сборщик мусора не имеет понятия об интерфейсе IDisposable, и потому никогда не будет вызывать метод Dispose(). Следовательно, когда пользователь объекта вызывает данный метод, объект все еще существует в управляемой куче и имеет доступ ко всем остальным находящимся там объектам. Логика вызова этого метода прямолинейна:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Dispose *****\n");
    // Создать освобождаемый объект и вызвать метод Dispose()
    // для освобождения любых внутренних ресурсов.
    MyResourceWrapper rw = new MyResourceWrapper();
    rw.Dispose();
    Console.ReadLine();
  }
}
```

Конечно, перед попыткой вызова метода Dispose() на объекте понадобится проверить, поддерживает ли тип интерфейс IDisposable. Хотя всегда можно выяснить, какие типы в библиотеках базовых классов реализуют IDisposable, заглянув в документацию .NET Framework SDK, программная проверка производится с помощью ключевого слова is или as (см. главу 6):

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Dispose *****\n");
    MyResourceWrapper rw = new MyResourceWrapper();
    if (rw is IDisposable)
      rw.Dispose();
    Console.ReadLine();
  }
}
```

Этот пример раскрывает очередное правило, касающееся управления памятью.

Правило. Неплохо вызывать метод Dispose() на любом создаваемом напрямую объекте, если он поддерживает интерфейс IDisposable. Предположение заключается в том, что когда проектировщик типа решил реализовать метод Dispose(), то тип должен выполнять какую-то очистку. Если вы забудете вызвать Dispose(), то память в конечном итоге будет очищена (так что можно не переживать), но это может занять больше времени, чем необходимо.

С предыдущим правилом связано одно предостережение. Несколько типов в библиотеках базовых классов, которые реализуют интерфейс IDisposable, предоставляют (кое в чем сбивающий с толку) псевдоним для метода Dispose() в попытке сделать имя метода очистки более естественным для определяющего его типа. В качестве примера можно взять класс System.IO.FileStream, который реализует интерфейс IDisposable (и поэтому поддерживает метод Dispose()), но также определяет следующий метод Close(), предназначенный для той же самой цели:

```
// Предполагается, что было импортировано пространство имен System.IO.
static void DisposeFileStream()
{
  FileStream fs = new FileStream("myFile.txt", FileMode.OpenOrCreate);
```

```
  // Мягко выражаясь, сбивает с толку!
  // Вызовы этих методов делают одно и то же!
  fs.Close();
  fs.Dispose();
}
```

В то время как "закрытие" (close) файла выглядит более естественным, чем его "освобождение" (dispose), подобное дублирование методов очистки может запутывать. При работе с типами, предлагающими псевдонимы, просто помните о том, что если тип реализует интерфейс `IDisposable`, то вызов метода `Dispose()` всегда является безопасным способом действия.

Повторное использование ключевого слова `using` в C#

Имея дело с управляемым объектом, который реализует интерфейс `IDisposable`, довольно часто приходится применять структурированную обработку исключений, гарантируя тем самым, что метод `Dispose()` типа будет вызываться даже в случае генерации исключения во время выполнения:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Dispose *****\n");
  MyResourceWrapper rw = new MyResourceWrapper ();
  try
  {
    // Использовать члены rw.
  }
  finally
  {
    // Всегда вызывать Dispose(), возникла ошибка или нет.
    rw.Dispose();
  }
}
```

Хотя это является хорошим примером защитного программирования, в действительности лишь немногих разработчиков привлекает перспектива помещения каждого освобождаемого типа внутрь блока `try/finally` просто для того, чтобы гарантировать вызов метода `Dispose()`. Того же самого результата можно достичь гораздо менее навязчивым способом, используя специальный фрагмент синтаксиса C#, который выглядит следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Dispose *****\n");
  // Метод Dispose() вызывается автоматически
  // при выходе за пределы области действия using.
  using(MyResourceWrapper rw = new MyResourceWrapper())
  {
    // Использовать объект rw.
  }
}
```

Если вы просмотрите код CIL метода `Main()` посредством `ildasm.exe`, то обнаружите, что синтаксис `using` на самом деле расширяется до логики `try/finally` с вполне ожидаемым вызовом `Dispose()`:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  ...
```

```
  .try
  {
    ...
  } // end .try
  finally
  {
  ...
  IL_0012: callvirt instance void
    SimpleFinalize.MyResourceWrapper::Dispose()
  } // end handler
  ...
} // end of method Program::Main
```

На заметку! Попытка применения `using` к объекту, который не реализует интерфейс `IDisposable`, приводит к ошибке на этапе компиляции.

Несмотря на то что этот синтаксис устраняет необходимость вручную помещать освобождаемые объекты внутрь блоков `try/finally`, к сожалению, теперь ключевое слово `using` в C# имеет двойной смысл (импортирование пространств имен и вызов метода `Dispose()`). Однако при работе с типами.NET, которые поддерживают интерфейс `IDisposable`, такая синтаксическая конструкция будет гарантировать, что используемый объект автоматический вызовет свой метод `Dispose()` по завершении блока `using`.

Кроме того, имейте в виду, что внутри `using` допускается объявлять несколько объектов *одного и того же типа*. Как и можно было ожидать, компилятор вставит код для вызова `Dispose()` на каждом объявленном объекте.

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Dispose *****\n");

  // Использование списка с разделителями-запятыми для объявления
  // нескольких объектов, подлежащих освобождению.
  using(MyResourceWrapper rw = new MyResourceWrapper(),
                          rw2 = new MyResourceWrapper())
  {
    // Работать с объектами rw и rw2.
  }
}
```

Исходный код. Проект `SimpleDispose` доступен в подкаталоге `Chapter_13`.

Создание финализируемых и освобождаемых типов

К настоящему моменту вы видели два разных подхода к конструированию класса, который очищает внутренние неуправляемые ресурсы. С одной стороны, можно применять финализатор. Использование такого подхода дает уверенность в том, что объект будет очищать себя сам во время сборки мусора (когда бы она ни произошла) без вмешательства со стороны пользователя. С другой стороны, можно реализовать интерфейс `IDisposable` и предоставить пользователю объекта способ очистки объекта по окончании работы с ним. Тем не менее, если пользователь объекта забудет вызвать метод `Dispose()`, то неуправляемые ресурсы могут оставаться в памяти неопределенно долго.

Нетрудно догадаться, что в одном определении класса можно смешивать оба подхода. Это позволит извлечь лучшее от обеих моделей. Если пользователь объекта не забыл вызвать метод `Dispose()`, то можно проинформировать сборщик мусора о пропуске процесса финализации, вызвав метод `GC.SuppressFinalize()`. Если же пользователь объекта забыл вызвать `Dispose()`, то объект со временем будет финализирован и получит шанс освободить внутренние ресурсы. Преимущество здесь в том, что внутренние неуправляемые ресурсы будут освобождены тем или иным способом.

Ниже представлена очередная версия класса `MyResourceWrapper`, который теперь является финализируемым и освобождаемым; она определена в проекте консольного приложения C# по имени `FinalizableDisposableClass`:

```
// Усовершенствованная оболочка для ресурсов.
public class MyResourceWrapper : IDisposable
{
  // Сборщик мусора будет вызывать этот метод, если
  // пользователь объекта забыл вызвать Dispose().
  ~MyResourceWrapper()
  {
    // Очистить любые внутренние неуправляемые ресурсы.
    // **Не** вызывать Dispose() на управляемых объектах.
  }
  // Пользователь объекта будет вызывать этот метод
  // для как можно более скорой очистки ресурсов.
  public void Dispose()
  {
    // Очистить неуправляемые ресурсы.
    // Вызвать Dispose() для других освобождаемых объектов, содержащихся внутри.
    // Если пользователь вызвал Dispose(), то
    // финализация не нужна, поэтому подавить ее.
    GC.SuppressFinalize(this);
  }
}
```

Обратите внимание, что этот метод `Dispose()` был модифицирован для вызова метода `GC.SuppressFinalize()`, который информирует среду CLR о том, что вызывать деструктор при обработке данного объекта сборщиком мусора больше не обязательно, т.к. неуправляемые ресурсы уже освобождены посредством логики `Dispose()`.

Формализованный шаблон освобождения

Текущая реализация класса `MyResourceWrapper` работает довольно хорошо, но осталось еще несколько небольших недостатков. Во-первых, методы `Finalize()` и `Dispose()` должны освобождать те же самые неуправляемые ресурсы. Это может привести к появлению дублированного кода, что существенно усложнит сопровождение. В идеале следовало бы определить закрытый вспомогательный метод и вызывать его внутри указанных методов.

Во-вторых, желательно удостовериться в том, что метод `Finalize()` не пытается освободить любые управляемые объекты, когда это должен делать метод `Dispose()`. В-третьих, имеет смысл также позаботиться о том, чтобы пользователь объекта мог безопасно вызывать метод `Dispose()` много раз без возникновения ошибки. В настоящий момент защита подобного рода в методе `Dispose()` отсутствует.

Для решения таких проектных задач в Microsoft определили формальный шаблон освобождения, который соблюдает баланс между надежностью, возможностью сопровождения и производительностью. Вот окончательная версия класса `MyResourceWrapper`, в которой применяется этот официальный шаблон:

```
class MyResourceWrapper : IDisposable
{
  // Используется для выяснения, вызывался ли метод Dispose().
  private bool disposed = false;

  public void Dispose()
  {
    // Вызвать вспомогательный метод.
    // Указание true означает, что очистку
    // запустил пользователь объекта.
    CleanUp(true);

    // Подавить финализацию.
    GC.SuppressFinalize(this);
  }

  private void CleanUp(bool disposing)
  {
    // Удостовериться, не выполнялось ли уже освобождение.
    if (!this.disposed)
    {
      // Если disposing равно true, освободить
      // все управляемые ресурсы.
      if (disposing)
      {
        // Освободить управляемые ресурсы.
      }
      // Очистить неуправляемые ресурсы.
    }
    disposed = true;
  }
  ~MyResourceWrapper()
  {
    // Вызвать вспомогательный метод.
    // Указание false означает, что
    // очистку запустил сборщик мусора.
    CleanUp(false);
  }
}
```

Обратите внимание, что в `MyResourceWrapper` теперь определен закрытый вспомогательный метод по имени `CleanUp()`. Передавая ему `true` в качестве аргумента, мы указываем, что очистку инициировал пользователь объекта, поэтому должны быть очищены все управляемые *и* неуправляемые ресурсы. Однако когда очистка инициируется сборщиком мусора, при вызове методу `CleanUp()` передается значение `false`, чтобы внутренние освобождаемые объекты *не* освобождались (поскольку нельзя рассчитывать на то, что они все еще присутствуют в памяти). И, наконец, перед выходом из `CleanUp()` переменная-член `disposed` типа `bool` устанавливается в `true`, что дает возможность вызывать метод `Dispose()` много раз без возникновения ошибки.

На заметку! После того как объект был "освобожден", клиент по-прежнему может обращаться к его членам, т.к. объект пока еще находится в памяти. Следовательно, в надежном классе оболочки для ресурсов каждый член также необходимо снабдить дополнительной логикой, которая бы сообщала: "если объект освобожден, то ничего не делать, а просто возвратить управление".

Чтобы протестировать финальную версию класса `MyResourceWrapper`, добавим внутрь финализатора вызов `Console.Beep()`:

```
~MyResourceWrapper()
{
  Console.Beep();
  // Вызвать вспомогательный метод.
  // Указание false означает, что
  // очистку запустил сборщик мусора.
  CleanUp(false);
}
```

Далее обновим метод `Main()`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Dispose() / Destructor Combo Platter *****");

  // Вызвать метод Dispose() вручную. Это не приведет к вызову финализатора.
  MyResourceWrapper rw = new MyResourceWrapper();
  rw.Dispose();

  // Не вызывать метод Dispose(). Это приведет к вызову финализатора
  // и выдаче звукового сигнала.
  MyResourceWrapper rw2 = new MyResourceWrapper();
}
```

Обратите внимание, что мы явно вызываем метод `Dispose()` на объекте `rw`, поэтому вызов деструктора подавляется. Тем не менее, мы "забываем" вызвать метод `Dispose()` на объекте `rw2`, так что по окончании выполнения приложения возникает звуковой сигнал. Если закомментировать вызов `Dispose()` на объекте `rw`, то звуковых сигналов будет два.

Исходный код. Проект `FinalizableDisposableClass` доступен в подкаталоге `Chapter_13`.

На этом исследование особенностей управления объектами со стороны среды CLR через сборку мусора завершено. Хотя дополнительные (довольно экзотические) детали, касающиеся процесса сборки мусора (такие как слабые ссылки и восстановление объектов), здесь не рассматривались, полученных сведений должно быть вполне достаточно, чтобы продолжить изучение самостоятельно. В завершение главы мы взглянем на программное средство под названием *ленивое (отложенное) создание* объектов.

Ленивое создание объектов

При создании классов иногда приходится учитывать, что отдельная переменная-член на самом деле может никогда не понадобиться из-за того, что пользователь объекта не будет обращаться к методу (или свойству), в котором она используется. Действительно, подобное нередко происходит. Однако проблема может возникнуть, если создание такой переменной-члена сопряжено с выделением большого объема памяти.

Для примера предположим, что строится класс, который инкапсулирует операции цифрового музыкального проигрывателя. В дополнение к ожидаемым методам вроде `Play()`, `Pause()` и `Stop()` вы также хотите обеспечить возможность возвращения коллекции объектов `Song` (посредством класса по имени `AllTracks`), которая представляет все имеющиеся на устройстве цифровые музыкальные файлы.

Давайте создадим новый проект консольного приложения по имени `LazyObjectInstantiation` и определим в нем следующие типы классов:

```
// Представляет одиночную композицию.
class Song
{
  public string Artist { get; set; }
  public string TrackName { get; set; }
  public double TrackLength { get; set; }
}
// Представляет все композиции в проигрывателе.
class AllTracks
{
  // Наш проигрыватель может содержать
  // максимум 10 000 композиций.
  private Song[] allSongs = new Song[10000];

  public AllTracks()
  {
    // Предположим, что здесь производится
    // заполнение массива объектов Song.
    Console.WriteLine("Filling up the songs!");
  }
}
// Объект MediaPlayer имеет объекты AllTracks.
class MediaPlayer
{
  // Предположим, что эти методы делают что-то полезное.
  public void Play()  { /* Воспроизведение композиции */ }
  public void Pause() { /* Пауза в воспроизведении */    }
  public void Stop()  { /* Останов воспроизведения */    }
  private AllTracks allSongs = new AllTracks();

  public AllTracks GetAllTracks()
  {
    // Возвратить все композиции.
    return allSongs;
  }
}
```

В текущей реализации `MediaPlayer` предполагается, что пользователь объекта пожелает получать список объектов с помощью метода `GetAllTracks()`. Хорошо, а что если пользователю объекта *не* нужен такой список? В этой реализации память под переменную-член `AllTracks` по-прежнему будет выделяться, приводя тем самым к созданию 10 000 объектов `Song` в памяти:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Lazy Instantiation *****\n");

  // В этом вызывающем коде получение всех композиций не производится,
  // но косвенно все равно создаются 10 000 объектов!
  MediaPlayer myPlayer = new MediaPlayer();
  myPlayer.Play();

  Console.ReadLine();
}
```

Безусловно, лучше не создавать 10 000 объектов, с которыми никто не будет работать, потому что это намного увеличивает нагрузку на сборщик мусора .NET. В то время как можно вручную добавить код, который обеспечит создание объекта `allSongs` толь-

ко в случае, если он применяется (скажем, используя шаблон фабричного метода), есть более простой путь.

Библиотеки базовых классов предоставляют удобный обобщенный класс по имени `Lazy<>`, который определен в пространстве имен `System` внутри сборки `mscorlib.dll`. Этот класс позволяет определять данные, которые *не* будут создаваться до тех пор, пока они действительно не начнут применяться в коде. Поскольку класс является обобщенным, при первом его использовании вы должны явно указать тип создаваемого элемента, которым может быть любой тип из библиотек базовых классов .NET или специальный тип, построенный вами самостоятельно. Чтобы включить отложенную инициализацию переменной-члена `AllTracks`, просто замените показанный ниже код:

```
// Объект MediaPlayer имеет объект AllTracks.
class MediaPlayer
{
  ...
  private AllTracks allSongs = new AllTracks();

  public AllTracks GetAllTracks()
  {
    // Возвратить все композиции.
    return allSongs;
  }
}
```

следующим кодом:

```
// Объект MediaPlayer имеет объект Lazy<AllTracks>.
class MediaPlayer
{
  ...
  private Lazy<AllTracks> allSongs = new Lazy<AllTracks>();
  public AllTracks GetAllTracks()
  {
    // Возвратить все композиции.
    return allSongs.Value;
  }
}
```

Помимо того факта, что переменная-член `AllTrack` теперь имеет тип `Lazy<>`, важно обратить внимание на изменение также и реализации показанного выше метода `GetAllTracks()`. В частности, для получения актуальных сохраненных данных (в этом случае объекта `AllTracks`, поддерживающего 10 000 объектов `Song`) должно применяться доступное только для чтения свойство `Value` класса `Lazy<>`.

Взгляните, как благодаря этому простому изменению приведенный далее модифицированный метод `Main()` будет косвенно размещать объекты `Song` в памяти, только если метод `GetAllTracks()` действительно был вызван:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Lazy Instantiation *****\n");

  // Память под объект AllTracks здесь не выделяется!
  MediaPlayer myPlayer = new MediaPlayer();
  myPlayer.Play();

  // Размещение объекта AllTracks происходит
  // только в случае вызова метода GetAllTracks().
  MediaPlayer yourPlayer = new MediaPlayer();
  AllTracks yourMusic = yourPlayer.GetAllTracks();
  Console.ReadLine();
}
```

На заметку! Ленивое создание объектов полезно не только для уменьшения количества выделений памяти под ненужные объекты. Этот прием можно также использовать в ситуации, когда для создания члена применяется затратный в плане ресурсов код, такой как вызов удаленного метода, взаимодействие с реляционной базой данных и т.п.

Настройка процесса создания данных Lazy<>

При объявлении переменной Lazy<> действительный внутренний тип данных создается с использованием стандартного конструктора:

```
// При использовании переменной Lazy<> вызывается
// стандартный конструктор класса AllTracks.
private Lazy<AllTracks> allSongs = new Lazy<AllTracks>();
```

В некоторых случаях это может оказаться подходящим, но что если класс AllTracks имеет дополнительные конструкторы, и нужно обеспечить вызов подходящего из них? Более того, что если при создании переменной Lazy() должна выполняться какая-то специальная работа (кроме простого создания объекта AllTracks)? К счастью, класс Lazy() позволяет указывать в качестве необязательного параметра обобщенный делегат, который задает метод для вызова во время создания находящегося внутри типа.

Таким обобщенным делегатом является тип System.Func<>, который может указывать на метод, возвращающий тот же тип данных, что создается связанной переменной Lazy<>, и способный принимать вплоть до 16 аргументов (типизированных с применением обобщенных параметров типа). В большинстве случаев никаких параметров для передачи методу, на который указывает Func<>, задавать не придется. Вдобавок, чтобы значительно упростить работу с типом Func<>, рекомендуется использовать лямбда-выражения (отношения между делегатами и лямбда-выражениями были подробно освещены в главе 10).

С учетом всего этого ниже показана окончательная версия MediaPlayer, в которой добавлен небольшой специальный код, выполняемый при создании внутреннего объекта AllTracks. Не забывайте, что перед завершением метод должен возвратить новый экземпляр типа, помещенного в Lazy<>, причем применять можно любой конструктор по своему выбору (здесь по-прежнему вызывается стандартный конструктор AllTracks).

```
class MediaPlayer
{
  ...
  // Использовать лямбда-выражение для добавления дополнительного
  // кода, который выполняется при создании объекта AllTracks.
  private Lazy<AllTracks> allSongs = new Lazy<AllTracks>( () =>
    {
      Console.WriteLine("Creating AllTracks object!");
      return new AllTracks();
    }
  );
  public AllTracks GetAllTracks()
  {
    // Возвратить все композиции.
    return allSongs.Value;
  }
}
```

Итак, вы наверняка смогли оценить полезность класса `Lazy<>`. По существу этот обобщенный класс позволяет гарантировать, что затратные в плане ресурсов объекты будут размещены в памяти, только когда они требуются их пользователю. Если данная тема вас заинтересовала, загляните в раздел документации .NET Framework SDK, посвященный классу `System.Lazy<>`, и ознакомьтесь с дополнительными примерами реализации ленивого создания.

Исходный код. Проект `LazyObjectInstantiation` доступен в подкаталоге `Chapter_13`.

Резюме

Целью настоящей главы было прояснение процесса сборки мусора. Вы видели, что сборщик мусора запускается, только если не удается получить необходимый объем памяти из управляемой кучи (или когда происходит выгрузка из памяти соответствующего домена приложения). Помните, что разработанный в Microsoft алгоритм сборки мусора хорошо оптимизирован и предусматривает использование поколений объектов, дополнительных потоков для финализации объектов и управляемой кучи для обслуживания крупных объектов.

В главе также было показано, каким образом программно взаимодействовать со сборщиком мусора с применением класса `System.GC`. Как отмечалось, единственным случаем, когда в этом может возникнуть необходимость, является построение финализируемых или освобождаемых классов, которые имеют дело с неуправляемыми ресурсами.

Вспомните, что финализируемые типы — это классы, которые предоставляют деструктор (переопределяя метод `Finalize()`) для очистки неуправляемых ресурсов во время сборки мусора. С другой стороны, освобождаемые объекты являются классами (или структурами), реализующими интерфейс `IDisposable`, к которому пользователь объекта должен обращаться по окончании работы с ними. Наконец, вы изучили официальный шаблон освобождения, в котором смешаны оба подхода.

В завершение главы был рассмотрен обобщенный класс по имени `Lazy<>`. Вы узнали, что данный класс позволяет отложить создание затратных (в смысле потребления памяти) объектов до тех пор, пока вызывающая сторона действительно не затребует их. Это помогает сократить количество объектов, хранящихся в управляемой куче, и также обеспечивает создание затратных объектов только тогда, когда они действительно требуются в вызывающем коде.

ЧАСТЬ V

Программирование с использованием сборок .NET

В этой части

ГЛАВА 14

Построение и конфигурирование библиотек классов

На протяжении первых четырех частей книги было создано несколько "автономных" исполняемых приложений, в которых вся программная логика упаковывалась в единственный исполняемый файл (`*.exe`). Такие исполняемые сборки использовали в основном главную библиотеку классов .NET, т.е. `mscorlib.dll`. В то время как многие простые программы .NET могут быть сконструированы с применением только библиотек базовых классов .NET, многократно используемая программная логика нередко изолируется в *специальные* библиотеки классов (файлы `*.dll`), которые могут разделяться между приложениями.

В этой главе вы ознакомитесь с различными способами упаковки типов в специальные библиотеки кода. Для начала вы узнаете о разнесении типов по пространствам имен .NET. После этого вы исследуете шаблоны проектов библиотеки классов Visual Studio и разберетесь с разницей между закрытыми и разделяемыми сборками.

Затем вы изучите, как исполняющая среда определяет местонахождение сборки, и освоите глобальный кеш сборок (Global Assembly Cache — GAC), XML-файлы конфигурации приложений (файлы `*.config`), сборки политик издателя и пространство имен `System.Configuration`.

Определение специальных пространств имен

Прежде чем погружаться в детали развертывания и конфигурирования библиотек, сначала необходимо узнать, каким образом упаковывать свои специальные типы в пространства имен .NET. Вплоть до этого места в книге создавались небольшие тестовые программы, которые задействовали существующие пространства имен из мира .NET (в частности, `System`). Однако когда строится крупное приложение со многими типами, может оказаться удобным группирование связанных типов в специальные пространства имен. В C# это достигается с применением ключевого слова `namespace`. Явное определение специальных пространств имен становится еще более важным при построении .NET-сборок `*.dll`, т.к. другие разработчики будут нуждаться в ссылке на вашу библиотеку и импортировании специальных пространств имен для использования типов, содержащихся внутри них.

Чтобы исследовать все аспекты непосредственно, начнем с создания нового проекта консольного приложения под названием `CustomNamespaces`. Предположим, что требуется разработать коллекцию геометрических классов с именами `Square` (квадрат),

Circle (круг) и Hexagon (шестиугольник). Учитывая сходные между ними черты, было бы желательно сгруппировать их вместе в уникальном пространстве имен MyShapes внутри сборки CustomNamespaces.exe. На выбор доступны два базовых подхода. Первый из них предусматривает определение всех классов в единственном файле C# (ShapesLib.cs):

```
// ShapesLib.cs
using System;

namespace MyShapes
{
  // Класс Circle.
  public class Circle { /* Интересные члены... */ }

  // Класс Hexagon.
  public class Hexagon { /* Более интересные члены... */ }

  // Класс Square.
  public class Square { /* Даже еще более интересные члены... */ }
}
```

Хотя компилятор C# без проблем воспримет единственный файл кода C#, содержащий множество типов, могут возникнуть сложности, когда появится желание повторно использовать определения классов в новых проектах. Например, пусть разрабатывается новый проект, в котором необходимо взаимодействовать только с классом Circle. Если все типы определены в единственном файле кода, то вы так или иначе привязаны к целому набору типов. Следовательно, в качестве альтернативы вы можете разнести одно пространство имен по нескольким файлам кода C#. Чтобы обеспечить упаковку типов в ту же самую логическую группу, нужно просто помещать определения заданных классов в область действия одного и того же пространства имен:

```
// Circle.cs
using System;

namespace MyShapes
{
  // Класс Circle.
  public class Circle { /* Интересные методы... */ }
}

// Hexagon.cs
using System;

namespace MyShapes
{
  // Класс Hexagon.
  public class Hexagon { /* Более интересные методы... */ }
}

// Square.cs
using System;

namespace MyShapes
{
  // Класс Square.
  public class Square { /* Даже еще более интересные методы... */ }
}
```

В обоих случаях обратите внимание на то, что пространство MyShapes действует в качестве концептуального "контейнера" для этих классов. Когда в другом пространстве имен (таком как CustomNamespaces) необходимо работать с типами из отдельного пространства имен, вы применяете ключевое слово using, как поступали бы в случае использования пространств имен из библиотек базовых классов .NET:

```
// Обратиться к пространству имен из библиотек базовых классов.
using System;

// Использовать типы, определенные в пространстве имен MyShapes.
using MyShapes;

namespace CustomNamespaces
{
  public class Program
  {
    static void Main(string[] args)
    {
      Hexagon h = new Hexagon();
      Circle c = new Circle();
      Square s = new Square();
    }
  }
}
```

В рассматриваемом примере предполагается, что файл или файлы C#, где определено пространство имен `MyShapes`, являются частью того же самого проекта консольного приложения, что и файл с определением пространства имен `CustomNamespaces`; другими словами, все эти файлы задействованы для компиляции единственной исполняемой сборки .NET. Если пространство имен `MyShapes` определено во внешней сборке, то для успешной компиляции потребуется также добавить ссылку на данную библиотеку. Все детали построения приложений, взаимодействующих с внешними библиотеками, вы изучите на протяжении настоящей главы.

Разрешение конфликтов имен с помощью полностью заданных имен

Говоря формально, вы не обязаны применять ключевое слово `using` языка C# при ссылках на типы, определенные во внешних пространствах имен. Вы можете использовать *полностью заданные имена* типов, которые, как упоминалось в главе 1, представляют собой имена типов, предваренные названиями пространств имен, где типы определены. Например:

```
// Обратите внимание, что пространство имен MyShapes больше не импортируется!
using System;

namespace CustomNamespaces
{
  public class Program
  {
    static void Main(string[] args)
    {
      MyShapes.Hexagon h = new MyShapes.Hexagon();
      MyShapes.Circle c = new MyShapes.Circle();
      MyShapes.Square s = new MyShapes.Square();
    }
  }
}
```

Обычно необходимость в применении полностью заданных имен отсутствует. Они требуют большего объема клавиатурного ввода, но никак не влияют на размер кода и скорость выполнения. На самом деле в коде CIL типы *всегда* определяются с полностью заданными именами. С этой точки зрения ключевое слово `using` языка C# является просто средством экономии времени.

Тем не менее, полностью заданные имена могут быть полезными (а иногда и необходимыми) для избегания потенциальных конфликтов имен при использовании множества пространств имен, которые содержат идентично названные типы. Предположим, что есть новое пространство имен `My3DShapes`, где определены три класса, которые способны визуализировать фигуры в трехмерном формате:

```
// Еще одно пространство имен для работы с фигурами.
using System;
namespace My3DShapes
{
  // Класс для представления трехмерного круга.
  public class Circle { }

  // Класс для представления трехмерного шестиугольника.
  public class Hexagon { }

  // Класс для представления трехмерного квадрата.
  public class Square { }
}
```

Если теперь модифицировать класс `Program` так, как показано ниже, то компилятор выдаст набор сообщений об ошибках, потому что в обоих пространствах имен определены одинаково именованные классы:

```
// Масса неоднозначностей!
using System;
using MyShapes;
using My3DShapes;

namespace CustomNamespaces
{
  public class Program
  {
    static void Main(string[] args)
    {
      // На какое пространство имен производится ссылка?
      Hexagon h = new Hexagon();    // Ошибка на этапе компиляции!
      Circle c = new Circle();      // Ошибка на этапе компиляции!
      Square s = new Square();      // Ошибка на этапе компиляции!
    }
  }
}
```

Разрешить неоднозначности можно за счет применения полностью заданных имен:

```
// Теперь неоднозначности устранены.
static void Main(string[] args)
{
  My3DShapes.Hexagon h = new My3DShapes.Hexagon();
  My3DShapes.Circle c = new My3DShapes.Circle();
  MyShapes.Square s = new MyShapes.Square();
}
```

Разрешение конфликтов имен с помощью псевдонимов

Ключевое слово `using` языка C# также позволяет создавать псевдоним для полностью заданного имени типа. В этом случае определяется метка, которая на этапе компиляции заменяется полностью заданным именем типа. Определение псевдонимов предоставляет второй способ разрешения конфликтов имен. Вот пример:

```
using System;
using MyShapes;
using My3DShapes;

// Устранить неоднозначность, используя специальный псевдоним.
using The3DHexagon = My3DShapes.Hexagon;

namespace CustomNamespaces
{
  class Program
  {
    static void Main(string[] args)
    {
      // На самом деле здесь создается экземпляр класса My3DShapes.Hexagon.
      The3DHexagon h2 = new The3DHexagon();
      ...
    }
  }
}
```

Посредством такого альтернативного синтаксиса `using` можно также создавать псевдонимы для пространств имен с очень длинными названиями. Одним из самых длинных пространств имен в библиотеках базовых классов является `System.Runtime.Serialization.Formatters.Binary`, которое содержит член по имени `BinaryFormatter`. При желании экземпляр класса `BinaryFormatter` можно создать следующим образом:

```
using bfHome = System.Runtime.Serialization.Formatters.Binary;

namespace MyApp
{
  class ShapeTester
  {
    static void Main(string[] args)
    {
      bfHome.BinaryFormatter b = new bfHome.BinaryFormatter();
      ...
    }
  }
}
```

А так это делается с использованием традиционной директивы `using`:

```
using System.Runtime.Serialization.Formatters.Binary;

namespace MyApp
{
  class ShapeTester
  {
    static void Main(string[] args)
    {
      BinaryFormatter b = new BinaryFormatter();
      ...
    }
  }
}
```

На этом этапе не следует беспокоиться о том, для чего служит класс `BinaryFormatter` (он исследуется в главе 20). Сейчас просто запомните, что ключевое слово `using` в C# позволяет создавать псевдонимы для очень длинных полностью заданных имен или, как случается более часто, для разрешения конфликтов имен, которые могут возникать при импорте пространств имен, определяющих типы с идентичными названиями.

На заметку! Имейте в виду, что чрезмерное применение псевдонимов C# в результате может привести к получению запутанной кодовой базы. Если другие программисты в команде не знают о ваших специальных псевдонимах, то они могут полагать, что псевдонимы ссылаются на типы из библиотек базовых классов .NET, и прийти в замешательство, не обнаружив их описания в документации .NET Framework SDK.

Создание вложенных пространств имен

При организации типов допускается определять пространства имен внутри других пространств имен. В библиотеках базовых классов .NET подобное встречается во многих местах и обеспечивает размещение типов на более глубоких уровнях. Например, пространство имен `IO` вложено внутрь пространства имен `System`, давая `System.IO`. Чтобы создать корневое пространство имен, содержащее внутри себя существующее пространство имен `My3DShapes`, необходимо модифицировать код, как показано ниже:

```
// Вложение пространства имен.
namespace Chapter14
{
  namespace My3DShapes
  {
    // Класс для представления трехмерного круга.
    public class Circle{ }

    // Класс для представления трехмерного шестиугольника.
    public class Hexagon{ }

    // Класс для представления трехмерного квадрата.
    public class Square{ }
  }
}
```

Зачастую роль корневого пространства имен заключается просто в предоставлении дополнительного уровня области действия; следовательно, непосредственно в этой области действия определения каких-либо типов могут отсутствовать (как в случае пространства имен `Chapter14`). В такой ситуации вложенное пространство имен может определяться с использованием следующей компактной формы:

```
// Вложение пространства имен (другой способ).
namespace Chapter14.My3DShapes
{
  // Класс для представления трехмерного круга.
  public class Circle{ }

  // Класс для представления трехмерного шестиугольника.
  public class Hexagon{ }

  // Класс для представления трехмерного квадрата.
  public class Square{ }
}
```

Учитывая, что теперь пространство `My3DShapes` вложено внутрь корневого пространства имен `Chapter14`, понадобится обновить все существующие директивы `using` и псевдонимы типов:

```
using Chapter14.My3DShapes;
using The3DHexagon = Chapter14.My3DShapes.Hexagon;
```

Стандартное пространство имен Visual Studio

Относительно пространств имен осталось еще отметить, что по умолчанию при создании нового проекта C# в среде Visual Studio название стандартного пространства имен приложения будет идентичным имени проекта. После этого, когда в проект вставляются новые файлы кода с применением пункта меню Project⇨Add New Item (Проект⇨Добавить новый элемент), типы будут автоматически помещаться внутрь стандартного пространства имен. Если вы хотите изменить название стандартного пространства имен, то откройте окно свойств проекта, перейдите в нем на вкладку Application (Приложение) и введите желаемое имя в поле Default namespace (Стандартное пространство имен), как показано на рис. 14.1.

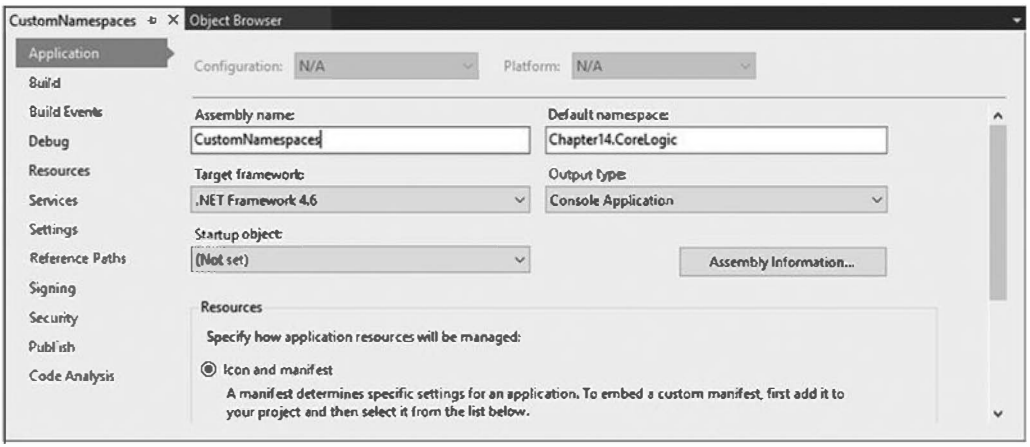


Рис. 14.1. Конфигурирование стандартного пространства имен

После такого изменения любой новый элемент, добавляемый в проект, будет помещаться внутрь пространства имен `Chapter14.CoreLogic` (и вполне очевидно, что для использования этих типов в другом пространстве имен должна применяться корректная директива `using`).

Теперь, когда вы ознакомились с некоторыми деталями упаковки специальных типов в четко организованные пространства имен, давайте кратко рассмотрим преимущества и формат сборки .NET. После этого мы углубимся в подробности создания, развертывания и конфигурирования специальных библиотек классов.

Исходный код. Проект `CustomNamespaces` доступен в подкаталоге `Chapter_14`.

Роль сборок .NET

Приложения .NET конструируются путем соединения в одно целое любого количества *сборок*. Попросту говоря, сборка представляет собой поддерживающий версии самоописательный двоичный файл, обслуживаемый средой CLR. Несмотря на то что сборки .NET имеют такие же файловые расширения (`*.exe` или `*.dll`), как и старые двоичные файлы Windows, внутри у них мало общего. Таким образом, для начала давайте выясним, какие преимущества предлагает формат сборки.

Сборки содействуют многократному использованию кода

При построении проектов консольных приложений в предшествующих главах могло показаться, что *вся* функциональность приложений содержалась внутри конструируемых исполняемых сборок. В действительности во всех этих приложениях были задействованы многочисленные типы из всегда доступной библиотеки кода .NET по имени `mscorlib.dll` (вспомните, что компилятор C# ссылается на `mscorlib.dll` автоматически), а в некоторых примерах также из библиотеки `System.Core.dll`.

Возможно, вы уже знаете, что *библиотека кода* (также называемая *библиотекой классов*) — это файл `*.dll`, который содержит типы, предназначенные для применения внешними приложениями. При построении исполняемых сборок вы без сомнения будете использовать много системных и специальных библиотек кода по мере создания приложений. Однако имейте в виду, что библиотека кода необязательно должна получать файловое расширение `*.dll`. Вполне допускается (хотя нечасто), чтобы исполняемая сборка работала с типами, определенными внутри внешнего исполняемого файла. В таком случае ссылаемый файл `*.exe` также может считаться библиотекой кода.

Независимо от того, как упакована библиотека кода, платформа .NET позволяет многократно применять типы в независимой от языка манере. Например, вы могли бы создать библиотеку кода на C# и повторно использовать ее при написании кода на другом языке программирования .NET. Возможно не только выделение памяти под экземпляры типов между языками, но также и наследование от них. Базовый класс, определенный в C#, может быть расширен классом, написанным на Visual Basic. Интерфейсы, определенные в F#, могут быть реализованы структурами, определенными в C#, и т.д. Важно понимать, что за счет разбиения единственного монолитного исполняемого файла на несколько сборок .NET достигается возможность многократного использования кода в форме, *нейтральной к языку*.

Сборки устанавливают границы типов

Вспомните, что *полностью заданное имя* типа получается путем предварения имени этого типа (скажем, `Console`) названием пространства имен, где он определен (`System`). Тем не менее, выражаясь строго, удостоверение типа дополнительно устанавливается сборкой, в которой он находится. Например, если есть две уникально именованные сборки (скажем, `MyCars.dll` и `YourCars.dll`), которые определяют пространство имен (`CarLibrary`), содержащее класс по имени `SportsCar`, то в мире .NET эти типы `SportsCar` будут рассматриваться как уникальные.

Сборки являются единицами, поддерживающими версии

Сборкам .NET назначается состоящий из четырех частей числовой номер версии в форме *<старший номер>.<младший номер>.<номер сборки>.<номер редакции>*. (Если номер версии явно не указан, то сборке автоматически назначается версия 1.0.0.0 из-за стандартных настроек проекта в Visual Studio.) Этот номер в сочетании с необязательным *значением открытого ключа* позволяет множеству версий той же самой сборки свободно сосуществовать на одной машине. Формально сборки, которые предоставляют информацию открытого ключа, называются *строго именованными*. Как вы увидите далее в главе, при наличии строгого имени среда CLR способна обеспечивать загрузку корректной версии в интересах вызывающего клиента.

Сборки являются самоописательными

Сборки считаются *самоописательными* частично потому, что содержат информацию обо всех внешних сборках, к которым они должны иметь доступ для корректно-

го функционирования. Таким образом, если сборке требуются библиотеки `System.Windows.Forms.dll` и `System.Core.dll`, то это будет документировано в *манифесте* сборки. Вспомните из главы 1, что манифест представляет собой блок метаданных, которые описывают саму сборку (имя, версия, обязательные внешние сборки и т.д.).

В дополнение к данным манифеста сборка содержит метаданные, которые описывают структуру каждого содержащегося в ней типа (имена членов, реализуемые интерфейсы, базовые классы, конструкторы и т.п.). Благодаря тому, что сборка настолько детально документирована, среда CLR *не* нуждается в обращении к реестру Windows для выяснения ее местонахождения (что радикально отличается от унаследованной модели программирования COM от Microsoft). Как вы узнаете в этой главе, для получения информации о местонахождении внешних библиотек кода среда CLR применяет совершенно новую схему.

Сборки являются конфигурируемыми

Сборки могут развертываться как "закрытые" или как "разделяемые". Закрытые сборки размещаются в том же каталоге (или возможно в подкаталоге), что и клиентское приложение, которое их использует. С другой стороны, разделяемые сборки — это библиотеки, предназначенные для потребления многочисленными приложениями на одиночной машине, которые развертываются в специальном каталоге, называемом *глобальным кешем сборок* (Global Assembly Cache — GAC).

Независимо от того, как развертываются сборки, для них могут быть созданы конфигурационные файлы, основанные на XML. Применяя такие конфигурационные файлы, среду CLR можно заставить "зондировать" сборки в специфичном местоположении, загружать конкретную версию ссылаемой сборки для определенного клиента либо обращаться в произвольный каталог на локальной машине, сетевой ресурс или URL-адрес. Вы узнаете больше о конфигурационных XML-файлах далее в главе.

Формат сборки .NET

Теперь, когда вы узнали о многих преимуществах сборок .NET, давайте более детально рассмотрим, как эти сборки устроены внутри. С точки зрения структуры сборка .NET (`*.dll` или `*.exe`) состоит из следующих элементов:

- заголовок файла Windows;
- заголовок файла CLR;
- код CIL;
- метаданные типов;
- манифест сборки;
- дополнительные встроенные ресурсы.

Несмотря на то что первые два элемента (заголовки Windows и CLR) представляют собой блоки данных, которые обычно можно игнорировать, краткого рассмотрения они все-таки заслуживают. Ниже приведен обзор всех перечисленных элементов.

Заголовок файла Windows

Заголовок файла Widows устанавливает факт того, что сборку можно загружать и манипулировать ею в средах операционных систем семейства Windows. Этот заголовок данных также идентифицирует тип приложения (консольное, с графическим пользовательским интерфейсом или библиотека кода `*.dll`), которое должно обслуживаться Windows. Чтобы просмотреть информацию заголовка Windows сборки, необходи-

мо открыть сборку .NET с помощью утилиты dumpbin.exe (в окне командной строки Windows), указав ей флаг /headers:

```
dumpbin /headers CarLibrary.dll
```

Ниже показана (неполная) информация заголовка Windows для сборки CarLibrary.dll, которая будет построена чуть позже в главе (можете запустить dumpbin.exe с именем любого ранее созданного файла *.dll или *.exe вместо CarLibrary.dll):

```
Dump of file CarLibrary.dll
PE signature found
File Type: DLL
FILE HEADER VALUES
            14C machine (x86)
              3 number of sections
       4B37DCD8 time date stamp Sun Dec 27 16:16:56 2011
              0 file pointer to symbol table
              0 number of symbols
             E0 size of optional header
           2102 characteristics
                   Executable
                   32 bit word machine
                   DLL
OPTIONAL HEADER VALUES
            10B magic # (PE32)
           8.00 linker version
            E00 size of code
            600 size of initialized data
              0 size of uninitialized data
           2CDE entry point (00402CDE)
           2000 base of code
           4000 base of data
         400000 image base (00400000 to 00407FFF)
           2000 section alignment
            200 file alignment
           4.00 operating system version
           0.00 image version
           4.00 subsystem version
              0 Win32 version
           8000 size of image
            200 size of headers
              0 checksum
              3 subsystem (Windows CUI)
...
Дамп файла CarLibrary.dll
Обнаружена подпись PE
Тип файла: DLL
Значения заголовка файла
            14C машина (x86)
              3 количество разделов
       4B37DCD8 дата и время Sun Dec 27 16:16:56 2011
              0 файловый указатель на таблицу символов
              0 количество символов
             E0 размер необязательного заголовка
           2102 характеристики
                        Исполняемый файл
                        Машина с 32-битным словом
                        DLL
```

```
Необязательные значения заголовка
             10B магический номер (PE32)
            8.00 версия редактора связей
             E00 размер кода
             600 размер инициализированных данных
               0 размер неинициализированных данных
            2CDE точка входа (00402CDE)
            2000 база кода
            4000 база данных
          400000 база образа (от 00400000 до 00407FFF)
            2000 выравнивание раздела
             200 выравнивание файла
            4.00 версия операционной системы
            0.00 версия образа
            4.00 версия подсистемы
               0 версия Win32
            8000 размер образа
             200 размер заголовков
               0 контрольная сумма
               3 подсистема (Windows CUI)
...
```

Запомните, что подавляющему большинству программистов .NET никогда не придется беспокоиться о формате данных заголовков, встроенных в сборку .NET. Если только вы не занимаетесь разработкой нового компилятора языка .NET (в таком случае вы *должны* позаботиться об такой информации), то можете не вникать в тонкие детали заголовков. Однако помните, что эта информация используется "за кулисами", когда операционная система Windows загружает двоичный образ в память.

Заголовок файла CLR

Заголовок CLR — это блок данных, который должны поддерживать все сборки .NET (и они поддерживают благодаря компилятору C#), чтобы обслуживаться средой CLR. В двух словах, в заголовке CLR определены многочисленные флаги, которые позволяют исполняющей среде воспринимать компоновку управляемого файла. Например, существуют флаги, идентифицирующие местоположение метаданных и ресурсов внутри файла, версию исполняющей среды, для которой была построена сборка, значение (необязательного) открытого ключа и т.д. Чтобы просмотреть данные заголовка CLR для сборки .NET, необходимо открыть сборку в утилите `dumpbin.exe`, указав флаг `/clrheader`:

```
dumpbin /clrheader CarLibrary.dll
```

Вот как выглядит заголовок CLR:

```
Dump of file CarLibrary.dll

File Type: DLL

  clr Header:

            48 cb
          2.05 runtime version
          2164 [    A74] RVA [size] of MetaData Directory
             1 flags
                IL Only
             0 entry point token
             0 [      0] RVA [size] of Resources Directory
             0 [      0] RVA [size] of StrongNameSignature Directory
             0 [      0] RVA [size] of CodeManagerTable Directory
```

```
            0 [      0] RVA [size] of VTableFixups Directory
            0 [      0] RVA [size] of ExportAddressTableJumps Directory
            0 [      0] RVA [size] of ManagedNativeHeader Directory
  Summary
       2000 .reloc
       2000 .rsrc
       2000 .text
Дамп файла CarLibrary.dll
Тип файла: DLL
  Заголовок clr:
            48 cb
          2.05 версия исполняющей среды
          2164 [    A74] RVA [размер] каталога MetaData
            1 флаги
                Только IL
            0 маркер записи
            0 [      0] RVA [размер] каталога Resources
            0 [      0] RVA [размер] каталога StrongNameSignature
            0 [      0] RVA [размер] каталога CodeManagerTable
            0 [      0] RVA [размер] каталога VTableFixups
            0 [      0] RVA [размер] каталога ExportAddressTableJumps
            0 [      0] RVA [размер] каталога ManagedNativeHeader
  Сводка
       2000 .reloc
       2000 .rsrc
       2000 .text
```

И снова отметим, что разработчикам приложений .NET не придется беспокоиться о тонких деталях информации заголовка CLR. Просто знайте, что каждая сборка .NET содержит данные такого рода, которые исполняющая среда .NET использует "за кулисами" при загрузке образа в память. Теперь переключим внимание на информацию, которая является намного более полезной при решении повседневных задач программирования.

Код CIL, метаданные типов и манифест сборки

В своей основе сборка содержит код CIL, который, как вы помните, представляет собой промежуточный язык, не зависящий от платформы и процессора. Во время выполнения внутренний код CIL на лету посредством JIT-компилятора компилируется в инструкции, специфичные для конкретной платформы и процессора. Благодаря такому проектному решению сборки .NET действительно могут выполняться под управлением разнообразных архитектур, устройств и операционных систем. (Хотя вы можете благополучно жить и продуктивно работать, не разбираясь в деталях языка программирования CIL, в главе 18 предлагается введение в синтаксис и семантику CIL.)

Сборка также содержит метаданные, полностью описывающие формат внутренних типов, а также формат внешних типов, на которые сборка ссылается. Исполняющая среда .NET применяет эти метаданные для выяснения местоположения типов (и их членов) внутри двоичного файла, для размещения типов в памяти и для упрощения удаленного вызова методов. Более подробно детали формата метаданных .NET будут раскрыты в главе 15 во время изучения служб рефлексии.

Сборка должна также содержать связанный с ней *манифест* (по-другому называемый *метаданными сборки*). Манифест документирует каждый *модуль* внутри сборки, устанавливает версию сборки и указывает любые *внешние* сборки, на которые ссыла-

ется текущая сборка. Как вы увидите далее в главе, среда CLR интенсивно использует манифест сборки в процессе нахождения ссылок на внешние сборки.

Необязательные ресурсы сборки

Наконец, сборка .NET может содержать любое количество встроенных ресурсов, таких как значки приложения, файлы изображений, звуковые клипы или таблицы строк. На самом деле платформа .NET поддерживает *подчиненные сборки*, которые содержат только локализованные ресурсы и ничего другого. Это может быть удобно, когда необходимо отделять ресурсы на основе культуры (русской, английской, немецкой и т.д.) при построении интернационального программного обеспечения. Тема создания подчиненных сборок выходит за рамки настоящей книги; если вам интересно, обращайтесь за дополнительными сведениями о подчиненных сборках в документацию .NET 4.6 Framework.

Построение и потребление специальной библиотеки классов

Перед началом исследований мира библиотек классов .NET давайте создадим сборку `*.dll` (по имени `CarLibrary`), которая содержит небольшой набор открытых типов. Для построения библиотеки классов в Visual Studio создадим новый проект Class Library (Библиотека классов), выбрав пункт меню File⇨New Project (Файл⇨Новый проект), как показано на рис. 14.2.

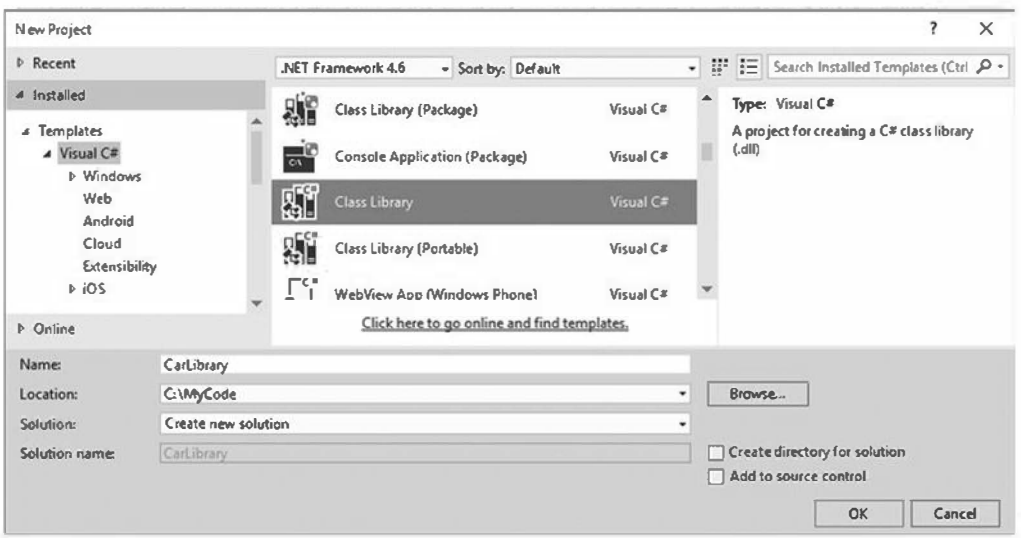


Рис. 14.2. Создание библиотеки классов C#

Проектное решение библиотеки для работы с автомобилями начинается с абстрактного базового класса по имени `Car`, который определяет разнообразные данные состояния посредством синтаксиса автоматических свойств. Этот класс также имеет единственный абстрактный метод `TurboBoost()`, в котором применяется специальное перечисление (`EngineState`), представляющее текущее состояние двигателя автомобиля:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CarLibrary
{
  // Представляет состояние двигателя.
  public enum EngineState
  { engineAlive, engineDead }

  // Абстрактный базовый класс в иерархии.
  public abstract class Car
  {
    public string PetName {get; set;}
    public int CurrentSpeed {get; set;}
    public int MaxSpeed {get; set;}
    protected EngineState egnState = EngineState.engineAlive;
    public EngineState EngineState
    {
      get { return egnState; }
    }
    public abstract void TurboBoost();
    public Car(){}
    public Car(string name, int maxSp, int currSp)
    {
      PetName = name; MaxSpeed = maxSp; CurrentSpeed = currSp;
    }
  }
}
```

Теперь предположим, что есть два непосредственных потомка класса `Car` **с именами** `MiniVan` **(минивэн) и** `SportsCar` **(спортивный автомобиль). В каждом из них абстрактный метод** `TurboBoost()` **переопределяется для отображения подходящего сообщения в окне сообщений Windows Forms. Вставим в проект новый файл классов C# под названием** `DerivedCars.cs` **со следующим кодом:**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

// Продолжайте чтение! Этот код не скомпилируется до тех пор,
// пока не будет добавлена ссылка на библиотеку .NET.
using System.Windows.Forms;

namespace CarLibrary
{
  public class SportsCar : Car
  {
    public SportsCar(){ }
    public SportsCar(string name, int maxSp, int currSp)
      : base (name, maxSp, currSp){ }
    public override void TurboBoost()
    {
      MessageBox.Show("Ramming speed!", "Faster is better...");
    }
  }
```

```
public class MiniVan : Car
{
  public MiniVan(){ }
  public MiniVan(string name, int maxSp, int currSp)
    : base (name, maxSp, currSp){ }

  public override void TurboBoost()
  {
    // Минивэны имеют плохие возможности ускорения!
    egnState = EngineState.engineDead;
    MessageBox.Show("Eek!", "Your engine block exploded!");
  }
 }
}
```

Обратите внимание, что каждый подкласс реализует метод `TurboBoost()` с использованием класса `MessageBox` из Windows Forms, который определен в сборке `System.Windows.Forms.dll`. Для применения в своей сборке типов, определенных внутри указанной внешней сборки, в проект `CarLibrary` потребуется добавить ссылку на эту сборку посредством диалогового окна Add Reference (Добавление ссылки), которое представлено на рис. 14.3 (оно доступно через пункт меню Project⇨Add Reference (Проект⇨Добавить ссылку)).

Очень *важно* понимать, что в области Framework диалогового окна Add Reference присутствуют далеко не все сборки, имеющиеся на машине. Диалоговое окно Add Reference *не* будет отображать специальные библиотеки, равно как и *не* будет отображать все библиотеки, расположенные в GAC (причины вы узнаете далее в главе). Взамен это диалоговое окно просто представляет список общих сборок, которые среда Visual Studio была изначально запрограммирована отображать. При построении приложений, которые требуют использования сборки, отсутствующей в диалоговом окне Add Reference, понадобится щелкнуть на узле Browse (Обзор) и вручную перейти к интересующему файлу `*.dll` или `*.exe`.

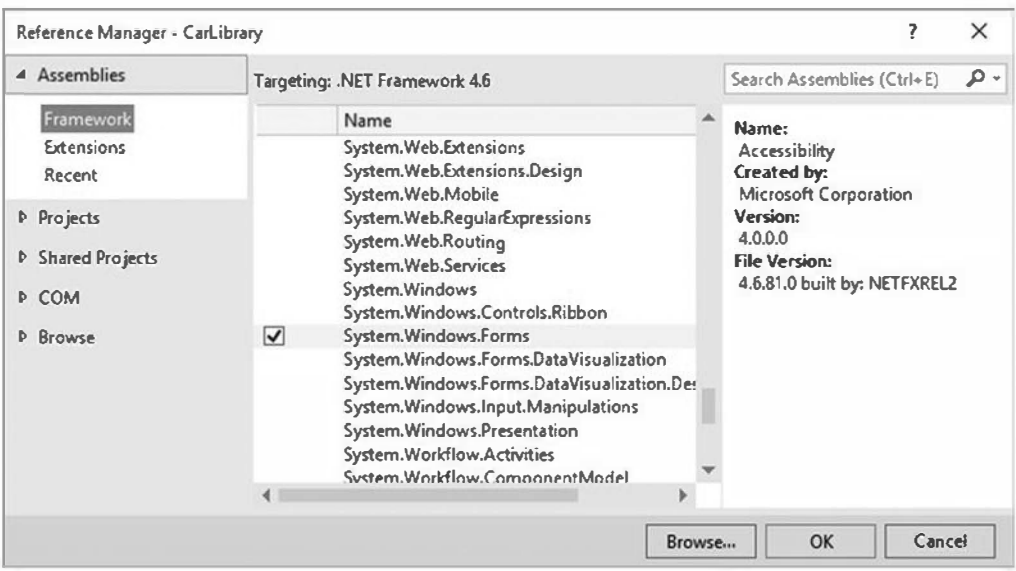


Рис. 14.3. Добавление ссылок на внешние сборки .NET с помощью диалогового окна Add Reference

На заметку! Имейте в виду, что в области Recent (Недавние ссылки) диалогового окна Add Reference поддерживается актуальный список сборок, на которые производилась ссылка ранее. Он может быть удобным, т.к. во многих проектах .NET приходится применять один и тот же основной набор внешних библиотек.

Исследование манифеста

Перед использованием `CarLibrary.dll` в клиентском приложении давайте посмотрим, как библиотека кода устроена внутри. Предполагая, что проект был скомпилирован, загрузим `CarLibrary.dll` в утилиту `ildasm.exe`, выбрав пункт меню File⇨Open (Файл⇨Открыть) и в открывшемся диалоговом окне перейдя в подкаталог `\bin\Debug` каталога проекта `CarLibrary`. Вы должны увидеть свою библиотеку, отображаемую в инструменте дизассемблера IL (рис. 14.4).

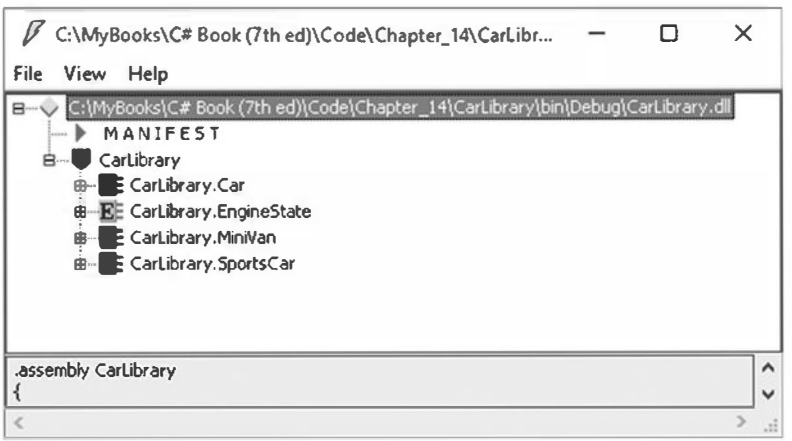


Рис. 14.4. Библиотека `CarLibrary.dll`, загруженная в `ildasm.exe`

Теперь откроем манифест сборки `CarLibrary.dll`, дважды щелкнув на значке MANIFEST (Манифест). В первом блоке кода манифеста указываются все внешние сборки, требуемые текущей сборкой для нормального функционирования. Вспомните, что в `CarLibrary.dll` применяются типы из внешних сборок `mscorlib.dll` и `System.Windows.Forms.dll`, которые перечислены в манифесте с использованием дескриптора `.assembly extern`:

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
.assembly extern System.Windows.Forms
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
```

Здесь каждый блок `.assembly extern` уточняется директивами `.publickeytoken` и `.ver`. Инструкция `.publickeytoken` присутствует только в случае, если сборка была сконфигурирована со *строгим именем* (более подробно о строгих именах речь пойдет в разделе "Понятие строгих имен" далее в главе). Дескриптор `.ver` определяет числовой идентификатор версии ссылаемой сборки.

После внешних ссылок вы обнаружите набор дескрипторов `.custom`, которые идентифицируют атрибуты уровня сборки (информацию об авторском праве, название компании, версию сборки и т.д.). Ниже приведена (весьма) небольшая часть этой порции данных манифеста:

```
.assembly CarLibrary
{
  .custom instance void ...AssemblyDescriptionAttribute...
  .custom instance void ...AssemblyConfigurationAttribute...
  .custom instance void ...RuntimeCompatibilityAttribute...
  .custom instance void ...TargetFrameworkAttribute...
  .custom instance void ...AssemblyTitleAttribute...
  .custom instance void ...AssemblyTrademarkAttribute...
  .custom instance void ...AssemblyCompanyAttribute...
  .custom instance void ...AssemblyProductAttribute...
  .custom instance void ...AssemblyCopyrightAttribute...
  ...
  .ver 1:0:0:0
}
.module CarLibrary.dll
```

Обычно эти настройки устанавливаются визуально с применением редактора свойств текущего проекта. Вернитесь в среду Visual Studio, щелкните на значке Properties внутри окна Solution Explorer и затем щелкните на кнопке Assembly Information (Информация о сборке) на (автоматически выбранной) вкладке Application. Откроется диалоговое окно Assembly Information (Информация о сборке), показанное на рис. 14.5.

После внесения и сохранения изменений будет обновлен файл `AssemblyInfo.cs` проекта, который поддерживается средой Visual Studio и может просматриваться путем раскрытия в окне Solution Explorer узла Properties (рис. 14.6).

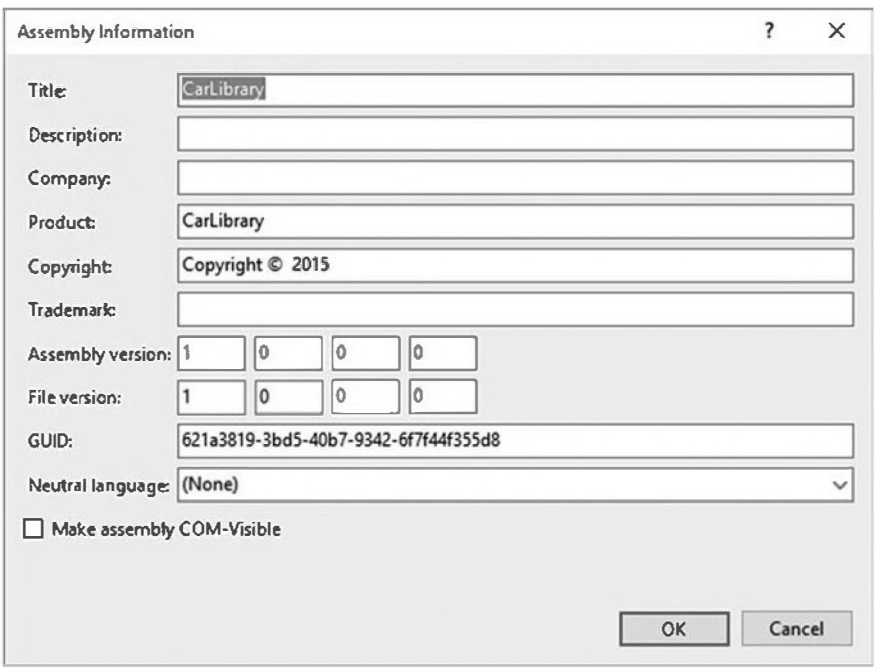


Рис. 14.5. Редактирование информации о сборке в диалоговом окне Assembly Information

Просматривая содержимое файла `AssemblyInfo.cs`, можно найти набор заключенных в квадратные скобки *атрибутов* .NET. Вот пример:

```
[assembly: AssemblyTitle("CarLibrary")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("CarLibrary")]
[assembly: AssemblyCopyright(
                    "Copyright © 2015")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

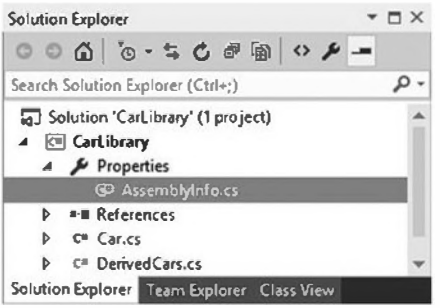


Рис. 14.6. После использования редактора свойств обновляется файл `AssemblyInfo.cs`

Роль атрибутов подробно обсуждается в главе 15, так что на данном этапе не беспокойтесь о деталях. Пока просто знайте, что большинство атрибутов в файле `AssemblyInfo.cs` будет применяться для обновления значений в блоках `.custom` внутри манифеста сборки.

Исследование кода CIL

Вспомните, что сборка не содержит инструкций, специфичных для платформы; вместо этого в ней хранятся инструкции на независимом от платформы общем промежуточном языке (Common Intermediate Language — CIL). Когда исполняющая среда .NET загружает сборку в память, лежащий в ее основе код CIL компилируется (с использованием JIT-компилятора) в инструкции, воспринимаемые целевой платформой. Например, если в утилите `ildasm.exe` дважды щелкнуть на методе `TurboBoost()` класса `SportsCar`, то откроется новое окно, в котором будут отображаться инструкции CIL, реализующие данный метод:

```
.method public hidebysig virtual instance void
  TurboBoost() cil managed
{
  // Code size       18 (0x12)
  .maxstack 8
  IL_0000: nop
  IL_0001: ldstr  "Ramming speed!"
  IL_0006: ldstr  "Faster is better..."
  IL_000b: call  valuetype [System.Windows.Forms]System.Windows.Forms.DialogResult
   [System.Windows.Forms]System.Windows.Forms.MessageBox::Show(string, string)
  IL_0010: pop
  IL_0011: ret
} // end of method SportsCar::TurboBoost
```

Несмотря на то что большинство разработчиков приложений .NET не нуждается в глубоких знаниях деталей кода CIL при повседневной работе, в главе 18 будут приведены более подробные сведения о синтаксисе и семантике языка CIL. Верите или нет, но понимание грамматики языка CIL может быть полезным, когда строятся более сложные приложения, которые требуют расширенных служб, таких как конструирование сборок во время выполнения (см. главу 18).

Исследование метаданных типов

Перед тем, как создавать приложения, в которых задействована ваша специальная библиотека .NET, находясь в окне утилиты `ildasm.exe`, нажмите комбинацию клавиш <Ctrl+M>; вы увидите метаданные для каждого типа внутри сборки `CarLibrary.dll` (рис. 14.7).

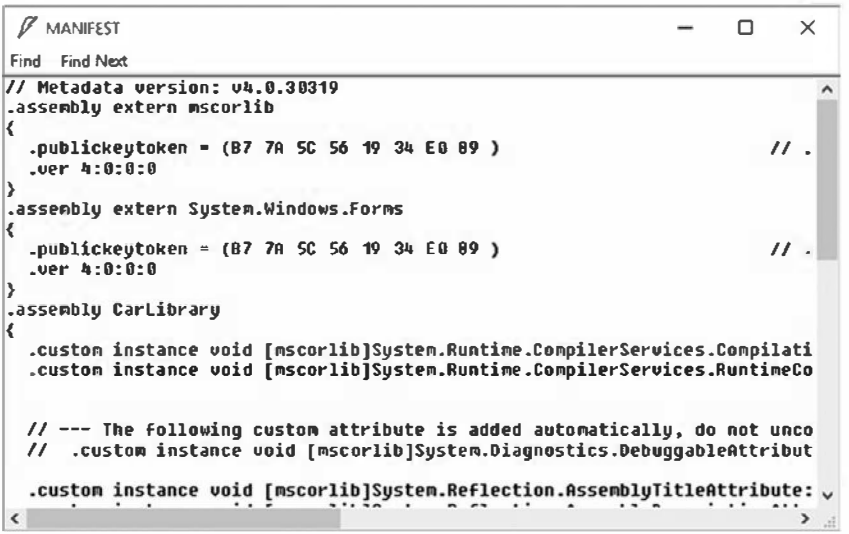


Рис. 14.7. Метаданные для типов в сборке `CarLibrary.dll`

Как объясняется в следующей главе, метаданные сборки являются важным элементом платформы .NET и служат основой для многочисленных технологий (сериализация объектов, позднее связывание, расширяемые приложения и т.д.). В любом случае теперь, когда мы заглянули внутрь сборки `CarLibrary.dll`, можно приступать к построению клиентских приложений, в которых будут применяться типы из сборки.

Исходный код. Проект `CarLibrary` доступен в подкаталоге `Chapter_14`.

Построение клиентского приложения C#

Поскольку все типы в `CarLibrary` были объявлены с ключевым словом `public`, другие приложения .NET имеют возможность их использовать. Вспомните, что типы могут также определяться с применением ключевого слова `internal` языка C# (в действительности это стандартный режим доступа в C#). Внутренние типы могут использоваться только в сборке, внутри которой они определены. Внешние клиенты не могут ни видеть, ни создавать экземпляры типов, помеченных ключевым словом `internal`.

Чтобы задействовать функциональность построенной библиотеки, создадим новый проект консольного приложения C# по имени `CSharpCarClient`. После этого установим ссылку на `CarLibrary.dll` с применением узла Browse диалогового окна Add Reference (если сборка `CarLibrary.dll` компилировалась в Visual Studio, то она будет находиться в подкаталоге `\Bin\Debug` внутри каталога проекта `CarLibrary`). Теперь можно строить клиентское приложение для использования внешних типов. Модифицируем начальный файл кода C#, как показано ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

// Не забудьте импортировать пространство имен CarLibrary!
using CarLibrary;
```

```
namespace CSharpCarClient
{
  public class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** C# CarLibrary Client App *****");

      // Создать объект SportsCar.
      SportsCar viper = new SportsCar("Viper", 240, 40);
      viper.TurboBoost();

      // Создать объект MiniVan.
      MiniVan mv = new MiniVan();
      mv.TurboBoost();

      Console.WriteLine("Done. Press any key to terminate");
      Console.ReadLine();
    }
  }
}
```

Этот код выглядит очень похожим на код в других приложениях, которые ранее разрабатывались в книге. Единственный интересный аспект связан с тем, что в клиентском приложении C# теперь применяются типы, определенные внутри отдельной специальной библиотеки. Запустив приложение, можно наблюдать отображение разнообразных окон с сообщениями.

Вас может интересовать, что в точности происходит при добавлении ссылки на `CarLibrary.dll` в диалоговом окне Add Reference. Щелкнув на кнопке Show All Files (Показать все файлы) в окне Solution Explorer, вы увидите, что среда Visual Studio добавила копию исходной библиотеки `CarLibrary.dll` в подкаталог `\bin\Debug` каталога проекта `CSharpCarClient` (рис. 14.8).

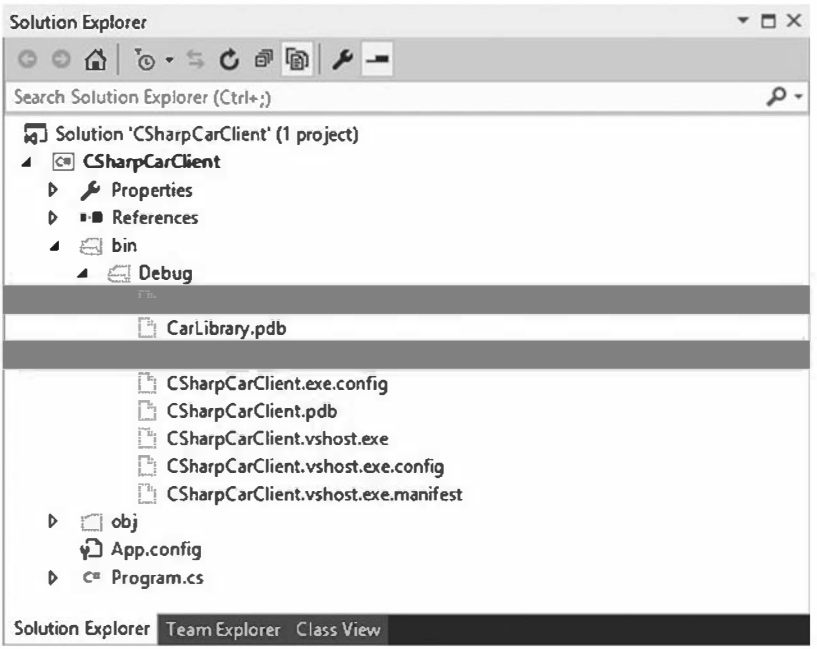


Рис. 14.8. Среда Visual Studio копирует закрытые сборки в каталог клиентского приложения

Как вскоре будет объяснено, `CarLibrary.dll` была сконфигурирована как закрытая сборка (это автоматическое поведение для всех проектов библиотек классов Visual Studio). Когда производится ссылка на закрытые сборки в новых приложениях (вроде `CSharpCarClient.exe`), IDE-среда реагирует помещением копии библиотеки в выходной каталог клиентского приложения.

Исходный код. Проект `CSharpCarClient` доступен в подкаталоге `Chapter_14`.

Построение клиентского приложения Visual Basic

Вспомните, что платформа .NET позволяет разработчикам разделять скомпилированный код между языками программирования. Чтобы проиллюстрировать языковую независимость платформы .NET, давайте создадим еще один проект консольного приложения (по имени `VisualBasicCarClient`) с использованием на этот раз языка Visual Basic (рис. 14.9). После создания проекта установим ссылку на `CarLibrary.dll` с применением диалогового окна Add Reference, которое открывается выбором пункта меню Project⇨Add Reference.

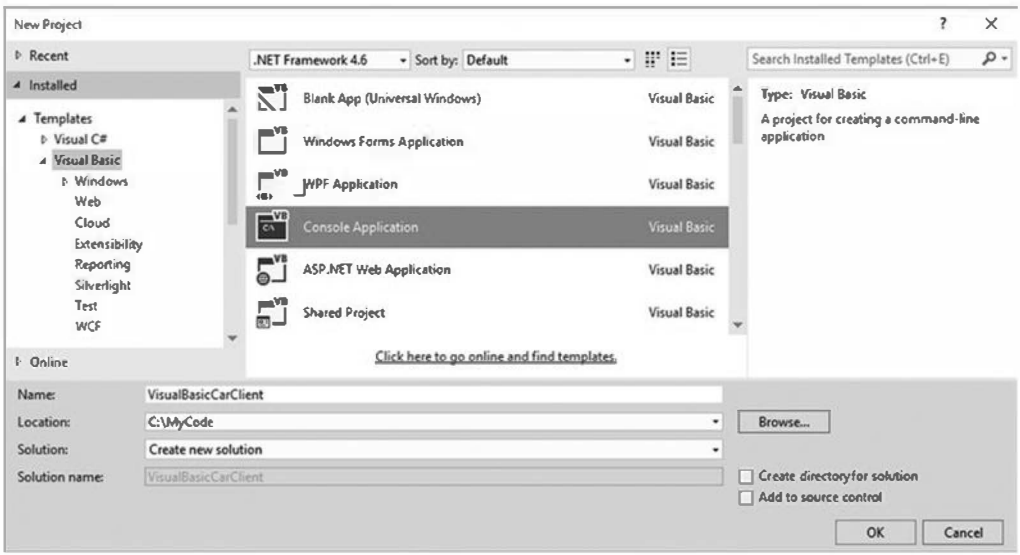


Рис. 14.9. Создание проекта консольного приложения Visual Basic

Подобно C# язык Visual Basic позволяет перечислять все пространства имен, используемые внутри текущего файла. Тем не менее, вместо ключевого слова `using`, применяемого в C#, для такой цели в Visual Basic служит ключевое слово `Imports`, поэтому добавим в файл кода `Module1.vb` следующий оператор `Imports`:

```
Imports CarLibrary

Module Module1
  Sub Main()
  End Sub
End Module
```

Обратите внимание, что метод `Main()` определен внутри типа модуля Visual Basic. Выражаясь кратко, модули — это система обозначений Visual Basic для определения класса, который может содержать только статические методы (очень похоже на стати-

ческий класс C#). Итак, чтобы испробовать типы MiniVan и SportsCar, используя синтаксис Visual Basic, модифицируем метод Main(), как показано ниже:

```
Sub Main()
  Console.WriteLine("***** VB CarLibrary Client App *****")

  ' Локальные переменные объявляются с применением ключевого слова Dim.
  Dim myMiniVan As New MiniVan()
  myMiniVan.TurboBoost()

  Dim mySportsCar As New SportsCar()
  mySportsCar.TurboBoost()
  Console.ReadLine()
End Sub
```

После компиляции и запуска приложения снова отобразится последовательность окон с сообщениями. Кроме того, это новое клиентское приложение имеет собственную локальную копию CarLibrary.dll в своем подкаталоге bin\Debug.

Межъязыковое наследование в действии

Привлекательным аспектом разработки в .NET является понятие *межъязыкового наследования*. В целях иллюстрации давайте создадим новый класс Visual Basic, производный от типа SportsCar (который был написан на C#). Для начала добавим в текущее приложение Visual Basic новый файл класса по имени PerformanceCar.vb (выбрав пункт меню Project⇨Add Class (Проект⇨Добавить класс)). Модифицируем начальное определение класса, унаследовав его от типа SportsCar с применением ключевого слова Inherits. Затем переопределим абстрактный метод TurboBoost(), используя ключевое слово Overrides, следующим образом:

```
Imports CarLibrary

' Этот класс VB унаследован от класса SportsCar, написанного на C#.
Public Class PerformanceCar
  Inherits SportsCar

  Public Overrides Sub TurboBoost()
    Console.WriteLine("Zero to 60 in a cool 4.8 seconds...")
  End Sub
End Class
```

Чтобы протестировать этот новый тип класса, модифицируем код метода Main() в модуле:

```
Sub Main()
  ...
  Dim dreamCar As New PerformanceCar()

  ' Использовать унаследованное свойство.
  dreamCar.PetName = "Hank"
  dreamCar.TurboBoost()
  Console.ReadLine()
End Sub
```

Обратите внимание, что объект dreamCar способен обращаться к любому открытому члену (такому как свойство PetName), находящемуся выше в цепочке наследования, невзирая на тот факт, что базовый класс был определен на совершенно другом языке и полностью в другой сборке! Возможность расширения классов за пределы границ сборок в независимой от языка манере — естественный аспект цикла разработки в .NET. Он упрощает применение скомпилированного кода, написанного программистами, которые предпочли не создавать свой разделяемый код на языке C#.

Исходный код. Проект `VisualBasicCarClient` доступен в подкаталоге `Chapter_14`.

Понятие закрытых сборок

Говоря формально, библиотеки классов, которые были созданы до сих пор в главе, развертывались как *закрытые сборки*. Закрытые сборки должны размещаться в том же самом каталоге, что и клиентское приложение, которое их использует (в *каталоге приложения*), или в одном из его подкаталогов. Вспомните, что при добавлении ссылки на `CarLibrary.dll` во время построения приложений `CSharpCarClient.exe` и `VbNetCarClient.exe` среда Visual Studio реагировала помещением копии `CarLibrary.dll` в каталоги упомянутых клиентских приложений (по крайней мере, после первой компиляции).

Когда в клиентской программе применяются типы, определенные в этой внешней сборке, среда CLR просто загружает локальную копию `CarLibrary.dll`. Поскольку при поиске внешних сборок исполняющая среда .NET не должна просматривать реестр Windows, сборки `CSharpCarClient.exe` (или `VisualBasicCarClient.exe`) и `CarLibrary.dll` можно перемещать в новое местоположение на машине и успешно запускать приложение (что часто называется *развертыванием с помощью Xcopy*).

Удаление (или копирование) приложения, в котором используются исключительно закрытые сборки, не требует особого труда: нужно просто удалить (или скопировать) каталог приложения. Важнее всего то, что не приходится переживать по поводу возможного нарушения работы других приложений на машине по причине удаления закрытых сборок.

Удостоверение закрытой сборки

Полное удостоверение закрытой сборки состоит из дружественного имени и числовой версии, которые записываются в манифест сборки. *Дружественное имя* — это просто название модуля, содержащего манифест сборки, без файлового расширения. Например, если просмотреть манифест сборки `CarLibrary.dll`, в нем можно обнаружить следующее:

```
.assembly CarLibrary
{
  ...
  .ver 1:0:0:0
}
```

Учитывая изолированную природу закрытой сборки, должно быть понятно, что среда CLR не применяет номер версии при выяснении места ее размещения. Предположение заключается в том, что закрытые сборки не нуждаются в выполнении сложной проверки версий, поскольку клиентское приложение является единственной сущностью, которой известно об их существовании. По этой причине на одной машине вполне возможно наличие множества копий одной и той же закрытой сборки в разных каталогах приложений.

Понятие процесса зондирования

Исполняющая среда .NET определяет местонахождение закрытой сборки, используя прием под названием *зондирование*, который в действительности не такой навязчивый, как может показаться. Зондирование представляет собой процесс отображения запроса внешней сборки на местоположение требуемого двоичного файла. Строго говоря, запрос на загрузку внешней сборки может быть либо *явным*, либо *неявным*. Неявный

запрос загрузки происходит, когда среда CLR исследует манифест для выяснения, где находится сборка, по дескрипторам `.assembly extern`. Вот пример:

```
// Неявный запрос загрузки.
.assembly extern CarLibrary
{ ... }
```

Явный запрос загрузки производится программно с применением метода `Load()` или `LoadFrom()` класса `System.Reflection.Assembly` обычно в целях позднего связывания или динамического вызова членов типа. Более подробно эти темы рассматриваются в главе 15, а пока ниже приведен пример явного запроса загрузки:

```
// Явный запрос загрузки, основанный на дружественном имени.
Assembly asm = Assembly.Load("CarLibrary");
```

Среда CLR извлекает дружественное имя сборки и начинает зондирование каталога клиентского приложения в поисках файла по имени `CarLibrary.dll`. Если указанный файл обнаружить не удалось, предпринимается попытка найти исполняемую сборку с таким же дружественным именем (например, `CarLibrary.exe`). Если ни один из файлов в каталоге приложения не найден, исполняющая среда генерирует исключение `FileNotFoundException` во время выполнения.

На заметку! Выражаясь формально, если копию запрашиваемой сборки не удалось найти в каталоге клиентского приложения, среда CLR будет также искать клиентский подкаталог с тем же именем, что и дружественное имя сборки (например, `C:\MyClient\CarLibrary`). Если запрашиваемая сборка размещена в этом подкаталоге, то среда CLR загрузит ее в память.

Конфигурирование закрытых сборок

В то время как допускается развертывать приложение .NET простым копированием всех требуемых сборок в единственный каталог на жестком диске пользователя, скорее всего, вы захотите определить несколько подкаталогов для группирования связанного содержимого. Например, предположим, что имеется каталог приложения по имени `C:\MyApp`, содержащий файл `CSharpCarClient.exe`. В этом каталоге может быть создан подкаталог `MyLibraries`, в котором находится сборка `CarLibrary.dll`.

Несмотря на подразумеваемую связь между этими двумя каталогами, среда CLR *не будет* зондировать подкаталог `MyLibraries`, если только не предоставить ей конфигурационный файл. Конфигурационные файлы содержат разнообразные XML-элементы, которые позволяют влиять на процесс зондирования. Конфигурационные файлы должны иметь такое же имя, как у запускаемого приложения, обладать расширением `*.config` и быть развернутыми в каталоге клиентского приложения. Таким образом, если вы хотите создать конфигурационный файл для приложения `CSharpCarClient.exe`, то он должен получить имя `CSharpCarClient.exe.config` и располагаться (в этом примере) в каталоге `C:\MyApp`.

Чтобы проиллюстрировать процесс, создадим на диске C: новый каталог по имени `MyApp`, используя проводник Windows. Далее скопируем в него файлы `CSharpCarClient.exe` и `CarLibrary.dll` и запустим программу, дважды щелкнув на исполняемом файле. На этом этапе программа должна запуститься успешно.

Теперь создадим в `C:\MyApp` новый подкаталог по имени `MyLibraries` (рис. 14.10) и переместим в него сборку `CarLibrary.dll`.

Попробуем запустить программу снова, дважды щелкнув на исполняемом файле. Поскольку среде CLR не удается обнаружить сборку по имени `CarLibrary` непосредственно в каталоге приложения, генерируется необработанное исключение `FileNotFoundException`.

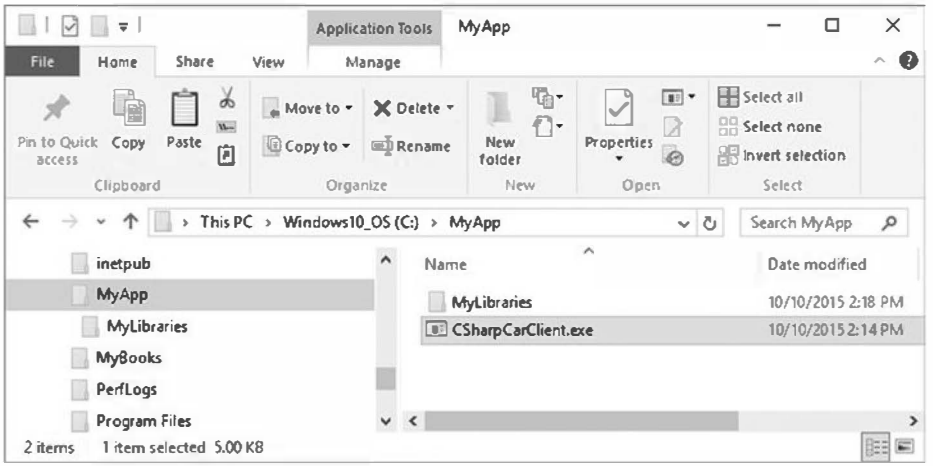


Рис. 14.10. Теперь файл `CarLibrary.dll` находится в подкаталоге `MyLibraries`

Для того чтобы указать среде CLR на необходимость зондирования подкаталога `MyLibraries`, создадим с помощью любого текстового редактора конфигурационный файл `CSharpCarClient.exe.config` и сохраним его в каталоге, содержащем приложение `CSharpCarClient.exe` (`C:\MyApp` в рассматриваемом примере). Откроем этот файл и поместим в него следующее содержимое (не забывайте, что язык XML чувствителен к регистру символов):

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="MyLibraries"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

Файлы `*.config` в .NET всегда начинаются с корневого элемента `<configuration>`. Вложенный в него элемент `<runtime>` может содержать элемент `<assemblyBinding>`, а тот, в свою очередь — вложенный элемент `<probing>`. В данном примере главный интерес представляет атрибут `privatePath`, т.к. именно он служит для указания подкаталогов внутри каталога приложения, которые среда CLR должна зондировать.

Завершив создание конфигурационного файла `CSharpCarClient.exe.config`, снова запустим клиентское приложение. На этот раз выполнение `CSharpCarClient.exe` должно пройти без проблем (если это не так, необходимо проверить конфигурационный файл на предмет опечаток).

Обязательно обратите внимание на то, что в элементе `<probing>` не указывается, *какая* сборка находится в заданном подкаталоге. Другими словами, невозможно указать, что сборка `CarLibrary` находится в подкаталоге `MyLibraries`, а сборка `MathLibrary` — в подкаталоге `OtherStuff`. Элемент `<probing>` просто инструктирует среду CLR о том, что она должна просмотреть все перечисленные подкаталоги в поисках запрашиваемой сборки до тех пор, пока не встретится первое совпадение.

На заметку! Очень важно запомнить, что атрибут `privatePath` не может применяться для указания абсолютного (`C:\Каталог\Подкаталог`) или относительного (`..\Каталог\ДругойПодкаталог`) пути! Если нужно указать каталог, находящийся за пределами каталога клиентского приложения, то придется применять совершенно другой XML-элемент под названием `<codeBase>` (он более подробно рассматривается позже в главе).

Атрибут `privatePath` допускает указание множества подкаталогов в виде списка значений, разделенных точками с запятой. В настоящий момент в этом нет необходимости, но ниже приведен пример, в котором среда CLR инструктируется на предмет просмотра клиентских подкаталогов `MyLibraries` и `MyLibraries\Tests`:

```
<probing privatePath="MyLibraries;MyLibraries\Tests"/>
```

В целях тестирования изменим (произвольным образом) имя конфигурационного файла и попробуем запустить приложение еще раз. Теперь клиентское приложение должно потерпеть неудачу. Вспомните, что файл `*.config` должен иметь то же имя, что и у связанного клиентского приложения. В качестве последнего теста откроем конфигурационный файл для редактирования и изменим регистр символов любого XML-элемента. После сохранения файла запуск клиентского приложения должен снова оказаться безуспешным (поскольку язык XML чувствителен к регистру символов).

На заметку! Имейте в виду, что среда CLR будет загружать ту сборку, которая во время процесса зондирования обнаруживается первой. Например, если в каталоге `C:\MyApp` имеется копия `CarLibrary.dll`, то именно она загрузится в память, а копия, содержащаяся в подкаталоге `MyLibraries`, будет проигнорирована.

Роль файла `App.Config`

Хотя всегда есть возможность создавать конфигурационные XML-файлы вручную в любом текстовом редакторе, среда Visual Studio позволяет создавать конфигурационный файл во время разработки клиентской программы. По умолчанию новый проект Visual Studio содержит конфигурационный файл, предназначенный для редактирования. Если когда-нибудь понадобится добавить его вручную, можно выбрать пункт меню Project⇨Add New Item. На рис. 14.11 обратите внимание, что для этого файла оставлено предложенное стандартное имя `App.config`.

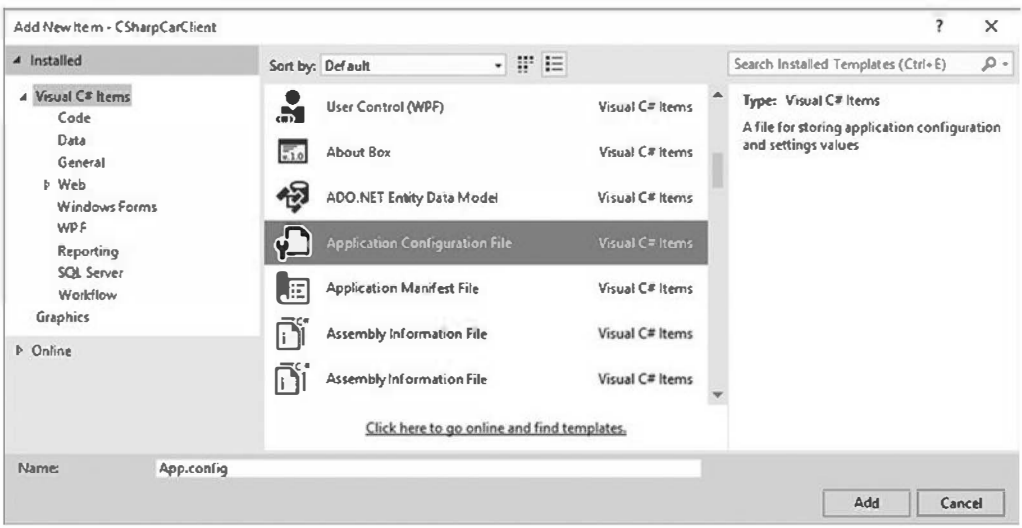


Рис. 14.11. Вставка нового конфигурационного XML-файла

Открыв этот файл для просмотра, можно увидеть минимальный набор инструкций, к которым вы будете добавлять дополнительные элементы:

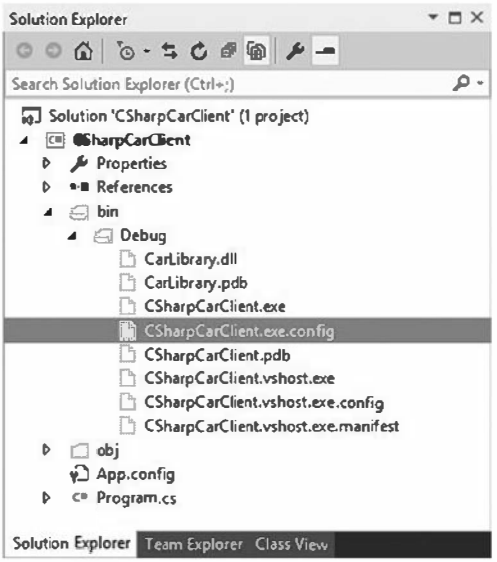


Рис. 14.12. Содержимое `App.config` будет скопировано в корректно именованный файл `*.config` внутри выходного каталога проекта

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0"
sku=".NETFramework,Version=v4.6" />
  </startup>
</configuration>
```

Необходимо отметить один интересный момент. Каждый раз, когда вы компилируете проект, среда Visual Studio будет автоматически копировать данные из `App.config` в новый файл, расположенный в каталоге `\bin\Debug`, используя надлежащее соглашение об именовании (вроде `CSharpCarClient.exe.config`). Однако это будет происходить, только если конфигурационный файл действительно имеет имя `App.config` (рис. 14.12).

При таком подходе от вас требуется лишь поддерживать файл `App.config`, а среда Visual Studio позаботится о том, чтобы в каталоге приложения содержались актуальные конфигурационные данные (даже если вы переименуете проект).

Понятие разделяемых сборок

Теперь, когда вы понимаете, каким образом развертывать и конфигурировать закрытые сборки, можно приступить к исследованию роли *разделяемых сборок*. Подобно закрытой сборке разделяемая сборка представляет собой коллекцию типов, предназначенных для многократного использования во множестве проектов. Самое очевидное отличие между разделяемыми и закрытыми сборками заключается в том, что единственная копия разделяемой сборки может быть задействована несколькими приложениями на той же самой машине.

Вспомните о том, что все приложения, создаваемые в книге, требуют доступа к сборке `mscorlib.dll`. Заглянув в каталог любого из этих клиентских приложений, вы *не* обнаружите там локальной копии упомянутой сборки .NET. Причина в том, что `mscorlib.dll` развернута как разделяемая сборка. Ясно, что если необходимо создать библиотеку классов для применения в масштабах всей машины, то именно так и следует поступать.

На заметку! Решение о том, каким образом должна развертываться библиотека кода — как закрытая или как разделяемая — является еще одним вопросом, который должен быть обдуман на этапе проектирования, и зависит от многих специфических деталей проекта. Существует эмпирическое правило: при построении библиотек, которые необходимо использовать в разнообразных приложениях, разделяемые сборки могут оказаться более удобными в том, что их легко обновлять до новых версий (как будет показано позже).

Глобальный кеш сборок

Из указанного выше следует, что разделяемая сборка не развертывается внутри того же самого каталога, где находится приложение, в котором она применяется. Вместо этого разделяемые сборки устанавливаются в глобальный кеш сборок (Global Assembly Cache — GAC). Тем не менее, точное местоположение GAC будет зависеть от версии платформы .NET, установленной на целевом компьютере.

На машинах с версиями, предшествующими .NET 4.0, глобальный кеш сборок размещен в подкаталоге `assembly` внутри каталога Windows (например, `C:\Windows\assembly`). В наши дни этот подкаталог можно считать "историческим GAC", т.к. он содержит только библиотеки .NET, скомпилированные для версий 1.0, 2.0, 3.0 или 3.5 (рис. 14.13).

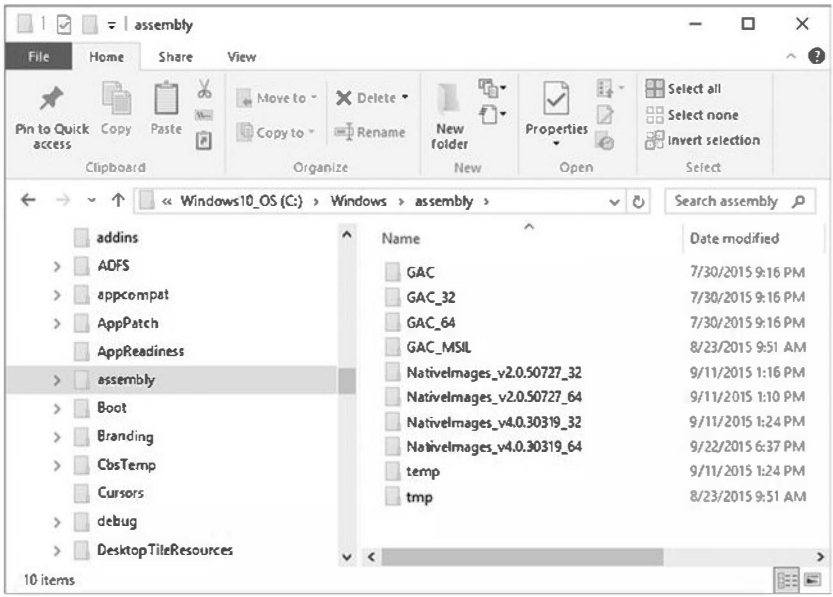


Рис. 14.13. "Исторический" глобальный кеш сборок

На заметку! Устанавливать в GAC исполняемые сборки (`*.exe`) не разрешено. В качестве разделяемых можно развертывать только сборки с расширением `*.dll`.

С выходом версии .NET 4.0 в Microsoft решили изолировать библиотеки для .NET 4.0 и последующих версий в отдельном месте, находящемся в `C:\Windows\Microsoft.NET\assembly\GAC_MSIL` (рис. 14.14).

В этом новом каталоге вы обнаружите набор подкаталогов, каждый из которых назван идентично дружественному имени отдельной библиотеки кода (например, `\System.Windows.Forms`, `\System.Core` и т.д.). Внутри заданного подкаталога с дружественным именем есть еще один подкаталог, который всегда именуется в соответствие со следующим соглашением:

```
v4.0_старшийНомер.младшийНомер.номерСборки.номерРедакции_значениеМаркераОткрытогоКлюча
```

Префикс `v4.0` обозначает, что библиотека скомпилирована в .NET 4.0 или последующей версии. За этим префиксом следует одиночный символ подчеркивания (_) и версия рассматриваемой библиотеки (скажем, `1.0.0.0`).

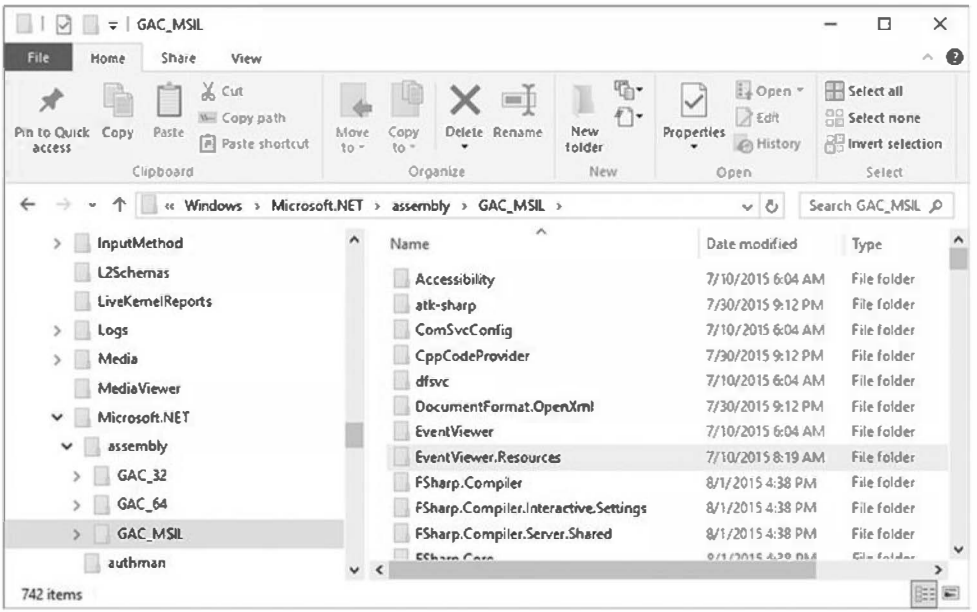


Рис. 14.14. Глобальный кеш сборок в .NET 4.0 и последующих версий

После пары символов подчеркивания находится число, называемое *значением маркера открытого ключа*. Как вы увидите в следующем разделе, значение открытого ключа является частью "строгого имени" сборки. Наконец, внутри этого подкаталога находится копия интересующей сборки `*.dll`.

В настоящей книге предполагается построение приложений с использованием версии .NET 4.6; таким образом, если вы устанавливаете библиотеку в GAC, то она попадет в каталог `C:\Windows\Microsoft.NET\assembly\GAC_MSIL`. Однако не забывайте, что если проект библиотеки классов сконфигурирован на компиляцию для платформы .NET 3.5 или более ранней версии, то разделяемые библиотеки следует искать в каталоге `C:\Windows\assembly`.

Понятие строгих имен

Перед развертыванием сборки в GAC ей должно быть назначено *строгое имя*, которое применяется для уникальной идентификации издателя заданного двоичного файла .NET. Имейте в виду, что в роли "издателя" может выступать отдельный программист, подразделение внутри компании или компания в целом.

В некоторых отношениях строгое имя является современным .NET-эквивалентом глобально уникальных идентификаторов (globally unique identifier — GUID) из технологии COM. Если вы ранее имели дело с COM, то можете вспомнить, что идентификаторы приложений (AppID) — это идентификаторы GUID, указывающие на конкретные COM-приложения. В отличие от значений GUID в COM (которые представляют собой всего лишь 128-битные числа) строгие имена основаны (отчасти) на двух криптографически связанных ключах (*открытых* и *секретных*), которые характеризуются гораздо более высокой уникальностью и устойчивостью к подделке, чем простые GUID.

Формально строгое имя образовано из набора связанных данных, большинство которых указывается с использованием перечисленных ниже атрибутов уровня сборки.

- Дружественное имя сборки (представляющее собой, как вы помните, имя сборки без файлового расширения).

- Номер версии сборки (назначенный посредством атрибута `[AssemblyVersion]`).
- Значение открытого ключа (назначенное с помощью атрибута `[AssemblyKeyFile]`).
- Необязательное значение, идентифицирующее культуру, для целей локализации (назначенное с применением атрибута `[AssemblyCulture]`).
- Встроенная цифровая подпись, созданная с использованием хеш-кода содержимого сборки и значения секретного ключа.

Для создания строгого имени сборки сначала генерируются данные открытого и секретного ключей посредством утилиты `sn.exe` из .NET Framework. Утилита `sn.exe` генерирует файл (обычно оканчивающийся расширением `*.snk` (Strong Name Key — ключ строгого имени)), который содержит данные для двух разных, но математически связанных ключей — открытого и секретного. После того как компилятору C# указано местоположение этого файла `*.snk`, он запишет полное значение открытого ключа в манифест сборки с применением дескриптора `.publickey`.

Компилятор C# также сгенерирует хеш-код на основе всего содержимого сборки (кода CIL, метаданных и т.д.). Как упоминалось в главе 6, *хеш-код* — это числовое значение, которое является статистически уникальным для фиксированных входных данных. Следовательно, в случае изменения любого аспекта сборки .NET (даже одного символа в каком-нибудь строковом литерале) компилятор выдаст другой хеш-код. Затем хеш-код комбинируется с данными секретного ключа, содержащимися внутри файла `*.snk`, для получения цифровой подписи, встраиваемой в данные заголовка CLR сборки. Процесс создания строгого имени показан на рис. 14.15.

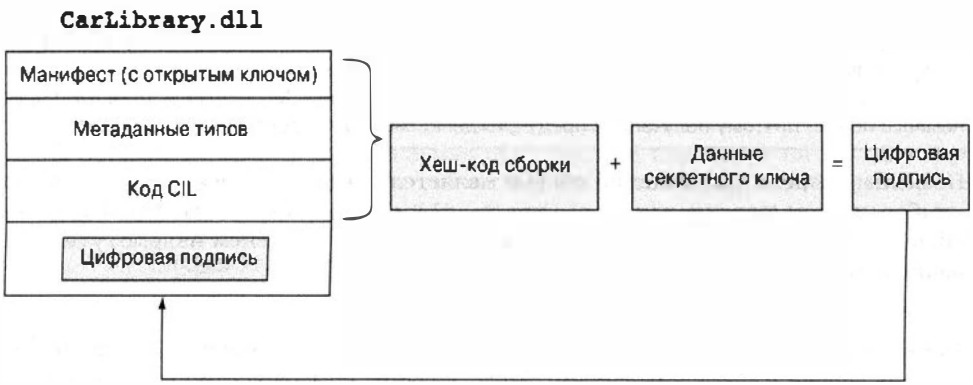


Рис. 14.15. На этапе компиляции генерируется цифровая подпись, основанная на данных открытого и секретного ключей, которая затем вставляется в сборку

Важно понимать, что действительные данные секретного ключа нигде в манифесте не встречаются, а используются только для цифрового подписания содержимого сборки (в сочетании со сгенерированным хеш-кодом). Общая идея применения открытого и секретного ключей — гарантия того, что никакие две компании, два подразделения или два программиста не будут иметь одно и то же удостоверение в мире .NET. По завершении процесса назначения строгого имени сборка может быть установлена в GAC.

На заметку! Строгие имена также обеспечивают уровень защиты от потенциальной подделки злоумышленниками содержимого сборок. С учетом этого установившейся практикой .NET считается назначение строгих имен всем сборкам (включая сборки `.exe`) независимо от того, развертываются они в GAC или нет.

Генерация строгих имен в командной строке

Давайте исследуем процесс назначения строгого имени сборке CarLibrary, созданной ранее в главе. В настоящее время необходимый файл *.snk, скорее всего, будет генерироваться с использованием Visual Studio. Тем не менее, в прошлом (примерно до 2003 года) назначать сборке строгое имя можно было только в командной строке. Посмотрим, как это делается. В первую очередь необходимо сгенерировать требуемые данные ключей с применением утилиты sn.exe. Хотя этот инструмент поддерживает многочисленные параметры командной строки, нас интересует только флаг -k, который приводит к генерированию нового файла с информацией открытого и секретного ключей.

Создадим на диске C: новый каталог по имени MyTestKeyPair, перейдем в него в окне командной строки Windows и введем следующую команду для генерации файла MyTestKeyPair.snk:

```
sn -k MyTestKeyPair.snk
```

Теперь, имея данные ключей, необходимо проинформировать компилятор C# о том, где находится файл MyTestKeyPair.snk. Как уже упоминалось ранее в главе, когда создается любой новый проект C# в Visual Studio, один из первоначальных файлов проекта (отображаемых в узле Properties окна Solution Explorer) имеет имя AssemblyInfo.cs. Этот файл содержит несколько атрибутов, которые описывают саму сборку. Чтобы сообщить компилятору местоположение действительного файла *.snk, в файл AssemblyInfo.cs можно добавить атрибут [AssemblyKeyFile] уровня сборки. Путь просто указывается в виде строкового параметра, например:

```
[assembly: AssemblyKeyFile(@"C:\MyTestKeyPair\MyTestKeyPair.snk")]
```

На заметку! Когда значение атрибута [AssemblyKeyFile] устанавливается вручную, среда Visual Studio выдаст предупреждение о том, что нужно либо указать параметр /keyfile для csc.exe, либо установить файл ключей в окне свойств проекта. Мы сделаем это в IDE-среде немного позже, поэтому полученное предупреждение можно попросту проигнорировать.

Поскольку версия разделяемой сборки является одним из аспектов строгого имени, выбор номера версии для CarLibrary.dll становится необходимой деталью. В файле AssemblyInfo.cs вы найдете еще один атрибут с именем AssemblyVersion. Первоначально его значение установлено в 1.0.0.0:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Номер версии в .NET состоит из четырех частей (старшего номера, младшего номера, номера сборки и номера редакции). Хотя указание номера версии возлагается полностью на вас, можно заставить Visual Studio автоматически инкрементировать номера сборки и редакции во время каждой компиляции, используя групповой символ. В рассматриваемом примере в этом нет нужды, но взгляните на следующий атрибут:

```
// Формат: <старший номер>.<младший номер>.<номер сборки>.<номер редакции>
// Допустимые значения для каждой части номера версии лежат
// в диапазоне от 0 до 65535.
 [assembly: AssemblyVersion("1.0.*")]
```

Теперь у компилятора C# есть вся информация, необходимая для генерации строгого имени (т.к. значение культуры в атрибуте [AssemblyCulture] не указано, "наследуется" культура, установленная на текущей машине).

Скомпилируем библиотеку кода CarLibrary, откроем ее в ildasm.exe и заглянем в манифест. Теперь можно видеть, что новый дескриптор .publickey содержит полную информацию об открытом ключе, а дескриптор .ver хранит номер версии, указанный в атрибуте [AssemblyVersion] (рис. 14.16).

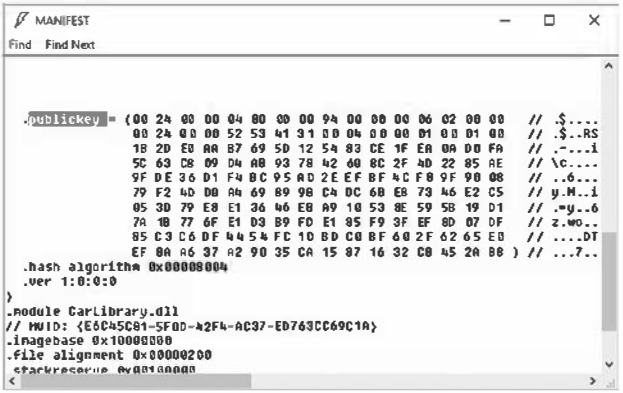


Рис. 14.16. В манифесте строго именованной сборки записана информация об открытом ключе

В этот момент можно было бы развернуть разделяемую сборку `CarLibrary.dll` в GAC. Однако вспомните, что в настоящее время для создания сборок со строгими именами разработчики приложений .NET могут применять Visual Studio вместо утилиты командной строки `sn.exe`. Прежде чем взглянуть, как это делается, потребуется удалить (или закомментировать) следующую строку кода в файле `AssemblyInfo.cs` (предполагая, что она была добавлена вручную):

```
// [assembly: AssemblyKeyFile(@"C:\MyTestKeyPair\MyTestKeyPair.snk")]
```

Генерация строгих имен в Visual Studio

Среда Visual Studio позволяет указывать местоположение существующего файла `*.snk` в окне свойств проекта, а также генерировать новый файл `*.snk`. Чтобы создать новый файл `*.snk` для проекта `CarLibrary`, дважды щелкните на значке Properties в окне Solution Explorer, в открывшемся окне свойств перейдите на вкладку Signing (Подпись), отметьте флажок Sign the assembly (Подписать сборку) и выберите в раскрывающемся списке вариант <New...> (Новый), как показано на рис. 14.17.

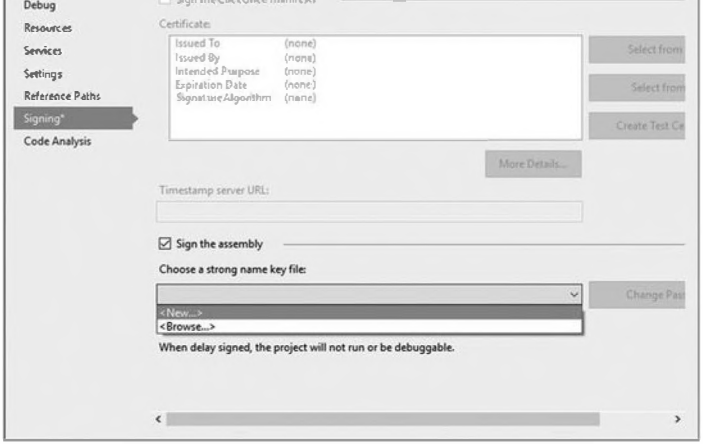


Рис. 14.17. Создание нового файла `*.snk` в Visual Studio

Откроется окно с приглашением указать имя для нового файла `*.snk` (например, `myKeyFile.snk`) и флажком Protect my key file with a password (Защитить файл ключей с помощью пароля), отмечать который в рассматриваемом примере не требуется (рис. 14.18). После этого новый файл `*.snk` появится в окне Solution Explorer (рис. 14.19).

Рис. 14.18. Именование нового файла `*.snk` в Visual Studio

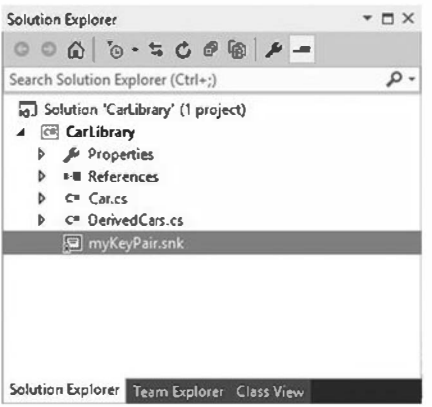


Рис. 14.19. Теперь при каждой компиляции среда Visual Studio будет назначать сборке строгое имя

Каждый раз, когда приложение компилируется, эти данные будут использоваться для назначения сборке надлежащего строгого имени.

На заметку! Вспомните, что на вкладке Application окна свойств проекта имеется кнопка Assembly Information. Щелчок на ней приводит к открытию диалогового окна, которое позволяет устанавливать многочисленные атрибуты уровня сборки, включая номер версии, информацию об авторских правах и т.д.

Установка строго именованных сборок в GAC

Последний шаг заключается в установке (теперь строго именованной) сборки `CarLibrary.dll` в GAC. Хотя предпочитаемый способ для развертывания сборок в GAC внутри производственной среды предусматривает создание установочного пакета (с применением коммерческой программы установки, такой как InstallShield), в составе .NET Framework SDK поставляется инструмент командной строки `gacutil.exe`, который удобен для проведения быстрых тестов.

На заметку! Для взаимодействия с GAC на своей машине необходимо иметь права администратора. Удостоверьтесь, что открываете окно командной строки от имени администратора.

В табл. 14.1 перечислены некоторые наиболее важные параметры `gacutil.exe` (для вывода полного списка параметров служит флаг `/?`).

Таблица 14.1. Параметры, которые принимает утилита `gacutil.exe`

Параметр	Описание
`/i`	Устанавливает сборку со строгим именем в GAC
`/u`	Удаляет сборку из GAC
`/l`	Отображает список сборок (или конкретную сборку) в GAC

Чтобы установить строго именованную сборку в GAC с помощью `gacutil.exe`, понадобится открыть окно командной строки и перейти в каталог, содержащий файл `CarLibrary.dll`. Вот пример (путь к каталогу у вас может быть другим):

```
cd C:\MyCode\CarLibrary\bin\Debug
```

Затем можно установить библиотеку, используя параметр `/i`:

```
gacutil /i CarLibrary.dll
```

После этого можно проверить, действительно ли библиотека была развернута, выполнив следующую команду с параметром `/l` (обратите внимание, что расширение файла в случае применения `/l` не указывается):

```
gacutil /l CarLibrary
```

Если все в порядке, то в окне консоли должен появиться показанный ниже вывод (как и ожидалось, вы увидите уникальное значение `PublicKeyToken`):

```
The Global Assembly Cache contains the following assemblies:
CarLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9,
processorArchitecture=MSIL
Number of items = 1

Глобальный кеш сборок содержит следующие сборки:
CarLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9,
processorArchitecture=MSIL
Количество элементов = 1
```

Более того, если вы перейдете в каталог `C:\Windows\Microsoft.NET\assembly\GAC_MSIL`, то обнаружите в нем новый каталог `CarLibrary` с корректной структурой подкаталогов (рис. 14.20).

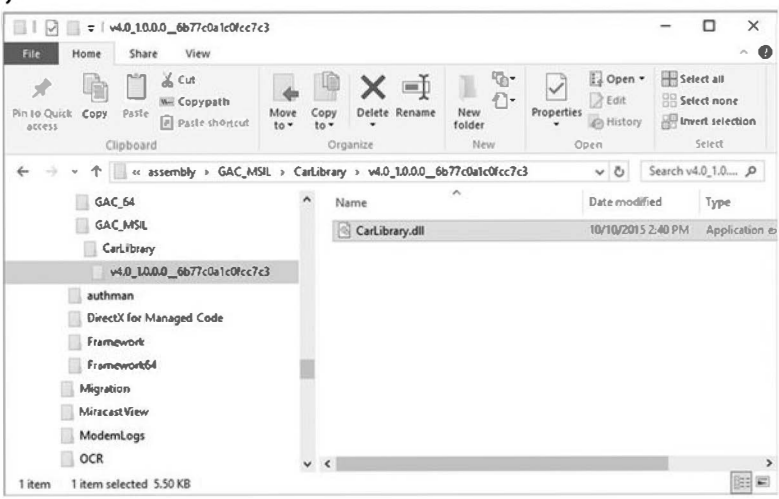


Рис. 14.20. Разделяемая сборка `CarLibrary` в GAC

Потребление разделяемой сборки

При построении приложений, которые используют разделяемую сборку, единственное отличие от случая закрытой сборки связано со способом ссылки на библиотеку в Visual Studio. На самом деле для этого применяется тот же самый инструмент — диалоговое окно Add Reference.

Когда необходимо сослаться на разделяемую сборку, можно было бы использовать кнопку Browse (Обзор) для перехода в соответствующий подкаталог GAC. Тем не менее, можно также просто перейти к месту хранения строго именованной сборки (такому как каталог \bin\debug проекта библиотеки классов) и сослаться на копию. Когда среда Visual Studio находит строго именованную сборку, она не будет копировать библиотеку в выходной каталог клиентского приложения. На рис. 14.21 показано добавление ссылки на библиотеку.

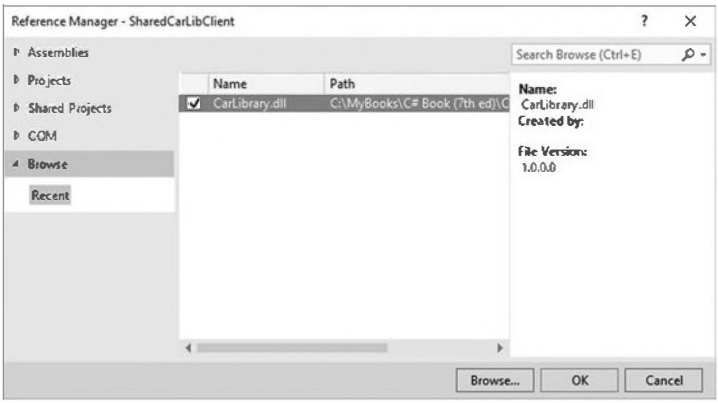


Рис. 14.21. Добавление ссылки на разделяемую сборку `CarLibrary` (версии 1.0.0.0) в Visual Studio

В целях иллюстрации создадим новый проект консольного приложения по имени `SharedCarLibClient` и добавим в него ссылку на сборку `CarLibrary`, как только что было описано. После этого вполне ожидаемо на вкладке References (Ссылки) окна Solution Explorer появится новый значок. Выбрав этот значок и открыв окно свойств (через меню View (Вид)), можно увидеть, что свойство Copy Local (Локальная копия) теперь установлено в `False`. Добавим в клиентское приложение следующий тестовый код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using CarLibrary;
namespace SharedCarLibClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Shared Assembly Client *****");
      SportsCar c = new SportsCar();
      c.TurboBoost();
      Console.ReadLine();
    }
  }
}
```

После компиляции этого клиентского приложения посредством проводника Windows перейдите в каталог, где хранится файл `SharedCarLibClient.exe`, и обратите внимание, что среда Visual Studio *не* скопировала `CarLibrary.dll` в каталог клиентского

приложения. При добавлении ссылки на сборку, манифест которой содержит значение .publickey, среда Visual Studio считает, что данная строго именованная сборка будет развернута в GAC, и потому не заботится о копировании ее двоичного файла.

Изучение манифеста SharedCarLibClient

Вспомните, что при генерации строгого имени для сборки в ее манифест записывается полный открытый ключ. В качестве связанного замечания: когда клиент ссылается на строго именованную сборку, в ее манифест помещается сжатое хеш-значение полного открытого ключа, обозначаемое дескриптором .publickeytoken. Если вы откроете манифест SharedCarLibClient.exe в ildasm.exe, то увидите следующий код (разумеется, значение маркера открытого ключа у вас будет отличаться, т.к. оно вычисляется на основе полного открытого ключа):

```
.assembly extern CarLibrary
{
  .publickeytoken = (33 A2 BC 29 43 31 E8 B9 )
  .ver 1:0:0:0
}
```

Сравнив значение маркера открытого ключа, записанного в манифесте клиента, и значение маркера открытого ключа, отображаемого в GAC, вы обнаружите, что они полностью совпадают. Как упоминалось ранее, открытый ключ представляет один из аспектов удостоверения строго именованной сборки. С учетом этого среда CLR будет загружать только версию 1.0.0.0 сборки по имени CarLibrary, имеющую открытый ключ, хеширование которого дает значение 33A2BC294331E8B9. Если среде CLR не удастся найти сборку, удовлетворяющую этому описанию, в GAC (и закрытую сборку по имени CarLibrary в каталоге клиента), то она сгенерирует исключение FileNotFoundException.

Исходный код. Проект SharedCarLibClient доступен в подкаталоге Chapter_14.

Конфигурирование разделяемых сборок

Подобно закрытым сборкам разделяемые сборки можно конфигурировать с применением клиентского файла *.config. Естественно, поскольку разделяемые сборки развертываются в хорошо известном месте (GAC), элемент <privatePath> для них не используется, как это делалось для закрытых сборок (хотя если клиент работает и с разделяемыми, и с закрытыми сборками, то элемент <privatePath> может присутствовать в файле *.config).

Конфигурационные файлы приложения в сочетании с разделяемыми сборками могут применяться, когда необходимо заставить среду CLR привязаться к *другой* версии заданной сборки, пропуская значение, которое записано в манифесте клиента. Так поступать удобно по нескольким причинам. Например, представьте, что вы поставили версию 1.0.0.0 сборки, но со временем в ней был выявлен крупный дефект. Одним из корректирующих действий могла бы быть перекомпиляция клиентского приложения для ссылки на версию сборки с устраненным дефектом (скажем, 1.1.0.0) и распространение обновленного приложения и новой сборки на все целевые машины.

Другой вариант предусматривает поставку новой библиотеки кода и файла *.config, который автоматически инструктирует исполняющую среду о необходимости привязки к новой (свободной от дефектов) версии. При условии, что новая версия установлена в GAC, первоначальное клиентское приложение будет функционировать без перекомпиляции или повторного распространения.

Рассмотрим еще один пример. Предположим, что была поставлена первая версия свободной от дефектов сборки (1.0.0.0), а через пару месяцев вы добавили в сборку новую функциональность, получив версию 2.0.0.0. Очевидно, что существующие клиентские приложения, которые компилировались со сборкой версии 1.0.0.0, не имеют никакого понятия о появившихся новых типах, т.к. их кодовая база не содержит ссылки на них.

Однако новым клиентским приложениям требуется ссылка на новую функциональность в версии 2.0.0.0. В .NET вы можете поставить на целевые машины сборку версии 2.0.0.0 и обеспечить ее работу бок о бок со сборкой версии 1.0.0.0. При необходимости существующие клиенты могут динамически перенаправляться для загрузки версии 2.0.0.0 (чтобы получить доступ к улучшениям реализации) с использованием конфигурационного файла, не требуя повторной компиляции и развертывания клиентского приложения.

Замораживание текущей версии разделяемой сборки

В целях иллюстрации динамической привязки к конкретной версии разделяемой сборки откроем окно проводника Windows и скопируем текущую версию скомпилированной сборки `CarLibrary.dll` (1.0.0.0) в другой подкаталог (например, `CarLibrary Version 1.0.0.0`), чтобы символизировать замораживание этой версии (рис. 14.22).

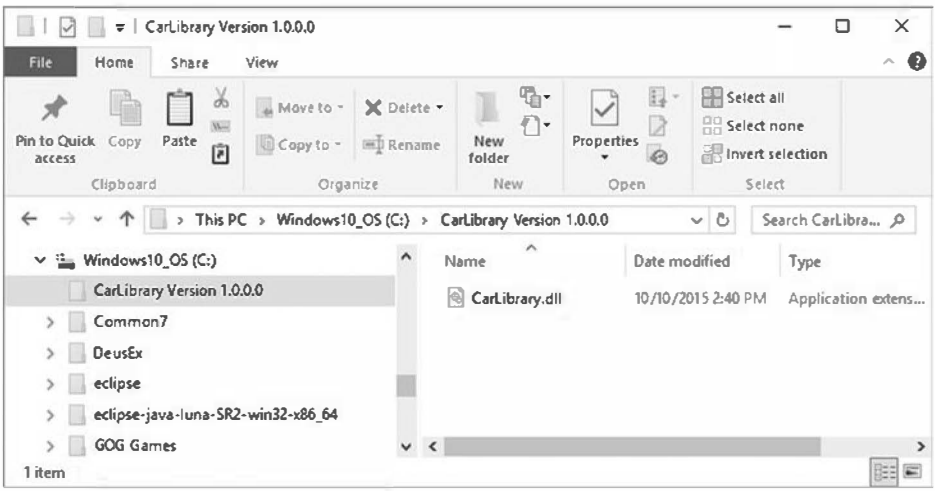


Рис. 14.22. Замораживание текущей версии сборки `CarLibrary.dll`

Построение разделяемой сборки версии 2.0.0.0

Теперь откроем существующий проект `CarLibrary` и модифицируем кодовую базу, добавив новый тип `enum` по имени `MusicMedia`, который определяет четыре возможных музыкальных устройства:

```
// Тип музыкального устройства, установленного в автомобиле.
public enum MusicMedia
{
  musicCd,
  musicTape,
  musicRadio,
  musicMp3
}
```

Кроме того, добавим в тип `Car` новый открытый метод, который позволяет вызывающему коду включать один из заданных музыкальных проигрывателей (при необходимости импортировав пространство имен `System.Windows.Forms`):

```
public abstract class Car
{
  ...
  public void TurnOnRadio(bool musicOn, MusicMedia mm)
  {
    if(musicOn)
      MessageBox.Show(string.Format("Jamming {0}", mm));
    else
      MessageBox.Show("Quiet time...");
  }
}
```

Изменим код конструкторов класса `Car` так, чтобы они отображали окно `MessageBox` с сообщением, подтверждающим применение версии 2.0.0.0 сборки `CarLibrary`:

```
public abstract class Car
{
  ...
  public Car()
  {
    MessageBox.Show("CarLibrary Version 2.0!");
  }
  public Car(string name, int maxSp, int currSp)
  {
    MessageBox.Show("CarLibrary Version 2.0!");
    PetName = name; MaxSpeed = maxSp; CurrentSpeed = currSp;
  }
  ...
}
```

И, наконец, перед перекомпиляцией новой библиотеки изменим номер версии с 1.0.0.0 на 2.0.0.0. Вспомните, что это можно делать визуально, дважды щелкнув на значке Properties в окне Solution Explorer и затем щелкнув на кнопке Assembly Information на вкладке Application. В открывшемся диалоговом окне необходимо просто изменить значения в полях Assembly Version (Версия сборки), как показано на рис. 14.23.

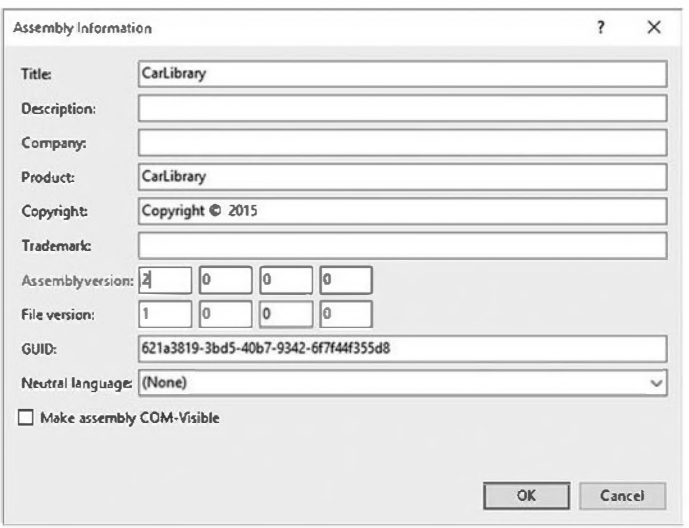


Рис. 14.23. Установка номера версии сборки `CarLibrary.dll` в 2.0.0.0

Заглянув в каталог \bin\Debug проекта, вы увидите, что в нем появилась сборка новой версии (2.0.0.0), в то время как сборка версии 1.0.0.0 хранится в подкаталоге CarLibrary Version 1.0.0.0. Давайте установим сборку новой версии в GAC для .NET 4.0 с помощью утилиты gacutil.exe, как было описано ранее в главе. Обратите внимание, что теперь на машине имеются две версии той же самой сборки (рис. 14.24).

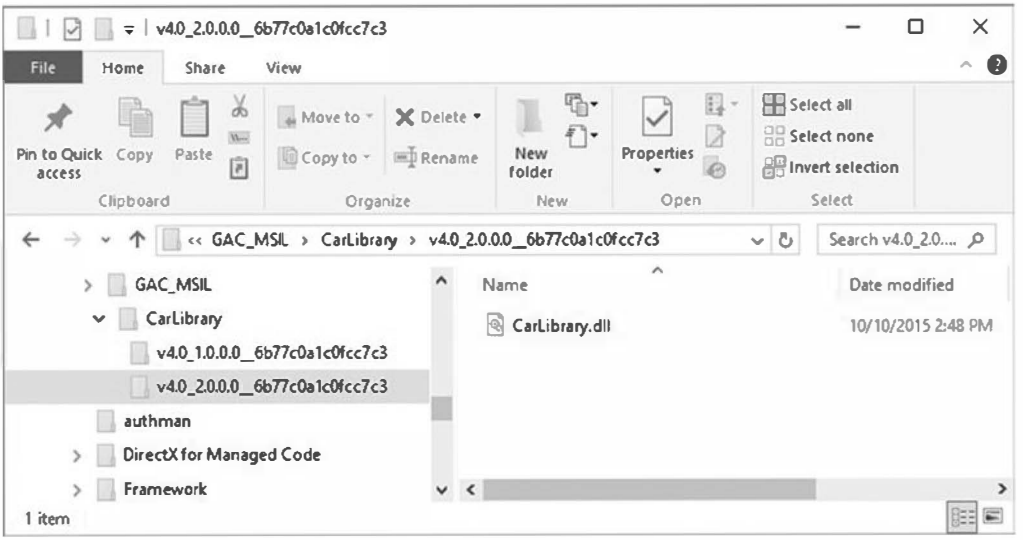


Рис. 14.24. Несколько версий разделяемой сборки

После запуска приложения SharedCarLibClient.exe окно с сообщением "CarLibrary Version 2.0!" *не* отображается, потому что в манифесте явным образом запрашивается версия 1.0.0.0. Как же тогда указать среде CLR о необходимости привязки к версии 2.0.0.0? Ответ на этот вопрос ищите ниже.

На заметку! При компиляции приложений среда Visual Studio будет автоматически сбрасывать ссылки! Следовательно, если вы запускаете приложение SharedCarLibClient.exe внутри Visual Studio, она захватит сборку CarLibrary.dll версии 2.0.0.0. Если вы случайно запустили приложение подобным образом, то просто удалите текущую ссылку на CarLibrary.dll и выберите сборку версии 1.0.0.0 (которая была помещена в подкаталог CarLibrary Version 1.0.0.0).

Динамическое перенаправление на специфичные версии разделяемой сборки

Когда среде CLR нужно сообщить о загрузке разделяемой сборки с версией, отличающейся от указанной в манифесте, можно создать файл *.config, который содержит элемент <dependentAssembly>. Потребуется создать подэлемент <assemblyIdentity> с дружественным именем сборки, которое указано в манифесте клиента (CarLibrary в этом примере), и необязательным атрибутом культуры (которому можно присвоить пустую строку или вообще опустить, если должна использоваться стандартная культура машины). Кроме того, элемент <dependentAssembly> будет содержать подэлемент <bindingRedirect> для определения версии, *в текущий момент* заданной в манифесте (атрибут oldVersion), и версии, которая должна загружаться вместо нее из GAC (атрибут newVersion).

Модифицируем конфигурационный файл `SharedCarLibClient.exe.config`, **находящийся в каталоге приложения** `SharedCarLibClient`, **как показано ниже.**

На заметку! Значение вашего маркера открытого ключа будет отличаться от того, который вы видите в приведенной далее разметке. Чтобы найти маркер открытого ключа, откройте клиентскую сборку в `ildasm.exe`, дважды щелкните на значке MANIFEST и скопируйте нужное значение в буфер (не забудьте удалить пробелы).

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!-- Информация о привязке во время выполнения -->
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
                          publicKeyToken="64ee9364749d8328"
                          culture="neutral"/>
        <bindingRedirect oldVersion= "1.0.0.0"
                         newVersion= "2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Теперь запустим программу `SharedCarLibClient.exe`. **На этот раз должно появиться окно с сообщением о загрузке сборки версии 2.0.0.0.**

В конфигурационном файле клиента могут присутствовать многочисленные элементы `<dependentAssembly>`. **Хотя в текущем примере в этом нет никакой необходимости, предположим, что манифест** `SharedCarLibClient.exe` **также ссылается на сборку** `MathLibrary` **версии 2.5.0.0. Если нужно перенаправить на сборку** `MathLibrary` **версии 3.0.0.0 (в дополнение к перенаправлению на сборку** `CarLibrary` **версии 2.0.0.0), то содержимое конфигурационного файла** `SharedCarLibClient.exe.config` **могло бы выглядеть следующим образом:**

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <!-- Управляет привязкой к CarLibrary -->
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
                          publicKeyToken="64ee9364749d8328"
                          culture=""/>
        <bindingRedirect oldVersion= "1.0.0.0" newVersion= "2.0.0.0"/>
      </dependentAssembly>

      <!-- Управляет привязкой к MathLibrary -->
      <dependentAssembly>
        <assemblyIdentity name="MathLibrary"
                          publicKeyToken="64ee9364749d8328"
                          culture=""/>
        <bindingRedirect oldVersion= "2.5.0.0" newVersion= "3.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

На заметку! В атрибуте `oldVersion` допускается указывать диапазон номеров старых версий; например, `<bindingRedirect oldVersion="1.0.0.0-1.2.0.0" newVersion="2.0.0.0"/>` информирует среду CLR о том, что вместо любой старой версии из диапазона от 1.0.0.0 до 1.2.0.0 должна применяться версия 2.0.0.0.

Понятие сборок политик издателя

Рассмотрим еще один аспект, касающийся конфигурации — *сборки политик издателя*. Как было только что показано, с помощью файлов `*.config` можно привязываться к специфической версии разделяемой сборки, обходя тем самым версию, которая записана в манифесте клиента. Но предположим, что вам, как администратору, требуется переконфигурировать *все* клиентские приложения на заданной машине с целью привязки к версии 2.0.0.0 сборки `CarLibrary.dll`. Учитывая строгое соглашение об именовании конфигурационных файлов, вам придется дублировать одно и то же XML-содержимое во множестве мест (при условии, что вы действительно знаете местоположение исполняемых файлов, работающих с `CarLibrary`). Понятно, что это может превратиться в настоящий кошмар сопровождения.

Политики издателя позволяют издателю конкретной сборки (в роли которого может выступать программист, подразделение или целая компания) поставлять двоичную версию файла `*.config`, которая устанавливается в GAC вместе с более новой версией связанной с ним сборки. Преимущество такого подхода состоит в том, что каталоги клиентских приложений *не* нуждаются в наличии специфических файлов `*.config`. Взамен среда CLR будет считывать текущий манифест и пытаться найти сборку запрашиваемой версии в GAC. Тем не менее, если среда CLR обнаружит сборку политик издателя, она прочитает содержащиеся в ней XML-данные и выполнит запрашиваемое перенаправление *на уровне GAC*.

Сборки политик издателя создаются в командной строке с использованием утилиты .NET под названием `al.exe` (редактор связей сборки). Данный инструмент поддерживает множество параметров, но построение сборки политик издателя требует передачи только нескольких из них:

- местоположение файла `*.config` или `*.xml`, содержащего инструкции перенаправления;
- имя результирующей сборки политик издателя;
- местоположение файла `*.snk`, применяемого для подписания сборки политик издателя;
- номера версии для назначения создаваемой сборке политик издателя.

Чтобы построить сборку политик издателя, которая управляет библиотекой `CarLibrary.dll`, выполните следующую команду (должна вводиться в одной строке):

```
al /link:CarLibraryPolicy.xml /out:policy.1.0.CarLibrary.dll
/keyf:C:\MyKey\myKey.snk /v:1.0.0.0
```

Здесь указано, что необходимое XML-содержимое находится в файле по имени `CarLibraryPolicy.xml`. Имя выходного файла (которое должно быть представлено в формате `policy.<старший номер>.<младший номер>.имяКонфигурируемойСборки`) задано с помощью флага `/out`. Кроме того, обратите внимание, что посредством флага `/keyf` также должно быть указано имя файла, содержащего пару открытого и секретного ключей. Не забывайте, что файлы политик издателя являются разделяемыми и потому должны иметь строгие имена!

В результате запуска `al.exe` создается новая сборка, которая может быть помещена в GAC, чтобы заставить всех клиентов привязываться к версии 2.0.0.0 сборки `CarLibrary.dll` без использования специального конфигурационного файла для каждого клиентского приложения. При таком подходе можно спроектировать перенаправление в масштабах машины для всех приложений, работающих с конкретной версией (или диапазоном версий) существующей сборки.

Отключение политики издателя

А теперь предположим, что вы (будучи системным администратором) развернули сборку политик издателя (и последнюю версию связанной сборки) в GAC на клиентской машине. Как обычно бывает, девять из десяти задействованных приложений привязались к версии 2.0.0.0 без ошибок, но одно (по ряду причин) при получении доступа к `CarLibrary.dll` версии 2.0.0.0 терпит неудачу. (Как все мы знаем, практически невозможно создать программное обеспечение с обратной совместимостью, которое функционировало бы корректно в любых ситуациях.)

В таком случае для проблемного клиента можно создать конфигурационный файл, который укажет среде CLR о необходимости *игнорировать* любые файлы политик издателя, установленные в GAC. Остальные клиентские приложения, успешно работающие с новой версией сборки .NET, будут просто перенаправляться посредством установленной сборки политик издателя. Чтобы отключить политику издателя на уровне отдельного клиента, потребуется создать файл `*.config` (с подходящим именем), добавить в него элемент `<publisherPolicy>` и установить атрибут `apply` в `no`. После этого среда CLR будет загружать сборку той версии, которая была изначально указана в манифесте клиента:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <publisherPolicy apply="no" />
    </assemblyBinding>
  </runtime>
</configuration>
```

Элемент <codeBase>

В конфигурационных файлах приложений можно также указывать *кодовые базы*. Элемент `<codeBase>` позволяет инструктировать среду CLR о необходимости зондирования зависимых сборок, находящихся в произвольных местоположениях (таких как сетевые конечные точки или любые пути на машине за пределами каталога клиентского приложения).

Если в `<codeBase>` указано местоположение на удаленной машине, то сборка будет загружаться по требованию в специальный каталог внутри GAC, который называется *кешем загрузки*. С учетом того, что вы уже знаете о развертывании сборок в GAC, должно быть ясно, что сборки, загружаемые из указанного в `<codeBase>` места, нуждаются в строгих именах (иначе среда CLR не смогла бы их установить в GAC). Просмотреть содержимое кеша загрузки на своей машине можно, запустив утилиту `gacutil.exe` с параметром `/ldl`:

```
gacuti1 /ld1
```

На заметку! Говоря формально, элемент `<codeBase>` может применяться для зондирования сборок, не обладающих строгими именами. Однако в таком случае местоположение сборки должно быть относительным к каталогу клиентского приложения (и тогда элемент `<codeBase>` становится просто альтернативой `<privatePath>`).

Чтобы посмотреть на элемент <codeBase> в действии, создадим новый проект консольного приложения по имени CodeBaseClient, установим ссылку на сборку CarLibrary.dll версии 2.0.0.0 и модифицируем начальный файл кода следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using CarLibrary;

namespace CodeBaseClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Fun with CodeBases *****");
      SportsCar c = new SportsCar();
      Console.WriteLine("Sports car has been allocated.");
      Console.ReadLine();
    }
  }
}
```

Поскольку сборка CarLibrary.dll была развернута в GAC, программу можно запускать в том виде, как есть. Тем не менее, для иллюстрации использования элемента <codeBase> создадим на диске C: новый каталог (скажем, C:\MyAsms) и скопируем в него сборку CarLibrary.dll версии 2.0.0.0.

Теперь добавим в проект CodeBaseClient файл App.config (или отредактируем существующий), как объяснялось ранее в главе, и поместим в него такое XML-содержимое (не забывайте, что ваше значение .publickeytoken будет отличаться; при необходимости просмотрите его в GAC):

```
<configuration>
  ...
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
       <assemblyIdentity name="CarLibrary" publicKeyToken="33A2BC294331E8B9"
                         culture="neutral"/>
       <codeBase version="2.0.0.0" href="file:///C:/MyAsms/CarLibrary.dll" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Как видите, элемент <codeBase> вложен внутрь элемента <assemblyIdentity>, в атрибутах name и publicKeyToken которого указано дружественное имя сборки и ассоциированное значение маркера открытого ключа. В самом элементе <codeBase> задается версия и местоположение (в свойстве href) сборки, которая должна загружаться. Если удалить версию 2.0.0.0 сборки CarLibrary.dll из GAC, то клиентское приложение по-прежнему будет успешно функционировать, т.к. среда CLR способна найти необходимую внешнюю сборку в каталоге C:\MyAsms.

На заметку! Размещая сборки в произвольных местах на машине разработки, вы по существу воссоздаете реестр Windows (и связанный с ним "ад DLL"), если учесть, что перемещение или переименование каталога, содержащего двоичные файлы, приведет к нарушению текущей привязки. Поэтому применяйте элемент <codeBase> осмотрительно.

Элемент <codeBase> может также быть удобным при ссылке на сборки, находящиеся на удаленной машине в сети. Предположим, что у вас есть права доступа к каталогу, расположенному на http://www.MySite.com. Для помещения удаленного файла *.dll в кеш загрузки GAC на локальной машине можно использовать следующий элемент <codeBase>:

```
<codeBase version="2.0.0.0"
  href="http://www.MySite.com/Assemblies/CarLibrary.dll" />
```

Исходный код. Проект CodeBaseClient доступен в подкаталоге Chapter_14.

Пространство имен System.Configuration

До сих пор во всех показанных в главе файлах *.config применялись хорошо известные XML-элементы, которые среда CLR считывала для выяснения местоположений внешних сборок. В дополнение к этим распознаваемым элементам конфигурационный файл клиента может содержать специфичные для приложения данные, которые не имеют ничего общего с механизмом привязки. Таким образом, не должен вызывать удивления тот факт, что платформа .NET Framework предоставляет пространство имен, которое позволяет программно читать данные из конфигурационного файла клиента.

Пространство имен System.Configuration предлагает небольшой набор типов, которые можно использовать для чтения специальных данных из файла *.config клиента. Эти специальные настройки должны быть помещены внутрь элемента <appSettings>. Элемент <appSettings> может содержать любое количество элементов <add>, которые определяют пары "ключ-значение", предназначенные для получения программным образом.

В качестве примера предположим, что имеется файл *.config, относящийся к проекту консольного приложения AppConfigReaderApp, в котором определены два значения, специфичные для приложения:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
  </startup>
  <!-- Специальные настройки приложения -->
  <appSettings>
    <add key="TextColor" value="Green" />
    <add key="RepeatCount" value="8" />
  </appSettings>
</configuration>
```

Чтение этих значений для работы с ними в клиентском приложении сводится просто к вызову метода GetValue() уровня экземпляра типа System.Configuration.AppSettingsReader. В следующем коде видно, что первым параметром GetValue() является имя ключа в файле *.config, а вторым — тип этого ключа (получаемый посредством операции typeof):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using System.Configuration;

namespace AppConfigReaderApp
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Reading <appSettings> Data *****\n");

      // Извлечь специальные данные из файла *.config.
      AppSettingsReader ar = new AppSettingsReader();
      int numbOfTimes = (int)ar.GetValue("RepeatCount", typeof(int));
      string textColor = (string)ar.GetValue("TextColor", typeof(string));

      Console.ForegroundColor =
        (ConsoleColor)Enum.Parse(typeof(ConsoleColor), textColor);

      // Вывести сообщение нужное количество раз.
      for (int i = 0; i < numbOfTimes; i++)
        Console.WriteLine("Howdy!");
      Console.ReadLine();
    }
  }
}
```

Исходный код. Проект `AppConfigReaderApp` доступен в подкаталоге `Chapter_14`.

Документация по схеме конфигурационного файла

В этой главе вы узнали о роли конфигурационных XML-файлов. Основное внимание здесь было сосредоточено на нескольких настройках, которые можно добавлять к элементу `<runtime>` для управления тем, как среда CLR будет искать требуемые внешние библиотеки. По мере дальнейшего чтения книги (и после перехода к построению крупномасштабного программного обеспечения) вы быстро заметите, что XML-файлы конфигурации применяются повсеместно.

Действительно, в рамках платформы .NET файлы `*.config` используются в многочисленных API-интерфейсах. Например, в главе 25 вы увидите, что в инфраструктуре Windows Communication Foundation (WCF) конфигурационные файлы применяются для установки сложных настроек сети. Позже в книге, когда будет демонстрироваться разработка веб-приложений с помощью ASP.NET, вы узнаете, что файл `web.config` содержит инструкции того же типа, что и файл `App.config` для настольных приложений.

Поскольку конфигурационный файл .NET может содержать большое количество инструкций, вы должны знать, что полная схема этого XML-файла документирована в справочной системе .NET. В частности, если поискать в справочной системе тему "Configuration File Schema for the .NET Framework" ("Схема конфигурационного файла для .NET Framework"), то вы получите детальные объяснения каждого элемента (рис. 14.25).

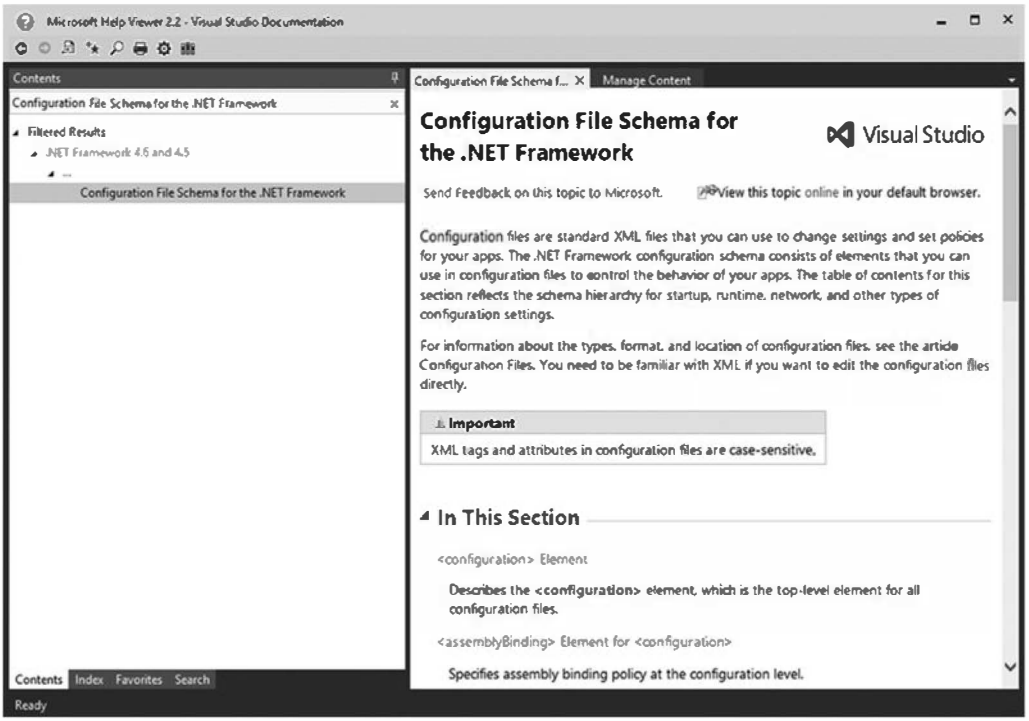


Рис. 14.25. Конфигурационные XML-файлы полностью документированы в справочной системе .NET

Резюме

В настоящей главе исследовалась роль библиотек классов .NET (файлов `*.dll` для .NET). Вы видели, что библиотеки классов представляют собой двоичные файлы .NET, содержащие логику, которая предназначена для многократного использования в разнообразных проектах. Вспомните, что библиотеки могут быть развернуты двумя основными путями — как закрытые или как разделяемые. Закрытые сборки развертываются в каталоге клиента или в его подкаталоге при условии, что имеется подходящий конфигурационный XML-файл. Разделяемые сборки являются библиотеками, которые могут потребляться любым приложением на машине, и на них также можно оказывать влияние посредством настроек в конфигурационном файле клиентской стороны.

Вы узнали, как назначать разделяемым сборками строгие имена, с помощью которых устанавливается уникальное удостоверение библиотек с точки зрения среды CLR. Кроме того, вы ознакомились с различными инструментами командной строки (`sn.exe` и `gacutil.exe`), которые применяются во время разработки и развертывания разделяемых библиотек.

Глава была завершена рассмотрением роли политик издателя и процесса сохранения и извлечения специальных настроек с использованием пространства имен `System.Configuration`.

ГЛАВА 15

Рефлексия типов, позднее связывание и программирование на основе атрибутов

Как было показано в главе 14, сборки являются базовой единицей развертывания в мире .NET. Используя интегрированные браузеры объектов Visual Studio (и многих других IDE-сред), можно просматривать типы внутри набора сборок, на которые ссылается проект. Кроме того, внешние инструменты, такие как утилита `ildasm.exe`, позволяют заглядывать внутрь лежащего в основе кода CIL, метаданных типов и манифеста сборки для заданного двоичного файла .NET. В дополнение к подобному исследованию сборок .NET на этапе проектирования ту же самую информацию можно получить *программно* с применением пространства имен `System.Reflection`. Таким образом, первой задачей этой главы является определение роли рефлексии и потребности в метаданных .NET.

Остаток главы посвящен нескольким тесно связанным темам, которые вращаются вокруг служб рефлексии. Например, вы узнаете, как клиент .NET может задействовать динамическую загрузку и позднее связывание для активизации типов, о которых на этапе компиляции ничего не известно. Вы также научитесь вставлять специальные метаданные в сборки .NET за счет использования системных и специальных атрибутов. Для практической демонстрации всех этих аспектов в завершение главы приводится пример построения нескольких "объектов-оснасток", которые можно подключать к расширяемому настольному приложению с графическим пользовательским интерфейсом.

Потребность в метаданных типов

Возможность полного описания типов (классов, интерфейсов, структур, перечислений и делегатов) с помощью метаданных является ключевым элементом платформы .NET. Многочисленным технологиям .NET, таким как Windows Communication Foundation (WCF) и сериализация объектов, требуется способность выяснения формата типов во время выполнения. Более того, межъязыковое взаимодействие, многие службы компилятора и средства IntelliSense в IDE-среде опираются на конкретное описание *типа*.

Вспомните из главы 1, что утилита `ildasm.exe` позволяет просматривать метаданные типов сборки по нажатию комбинации клавиш <Ctrl+M>. Таким образом, если вы откроете в `ildasm.exe` любую из сборок `*.dll` или `*.exe`, которые создавались ранее

в книге (например, CarLibrary.dll из главы 14), и нажмете <Ctrl+M>, то увидите метаданные типов (рис. 15.1).

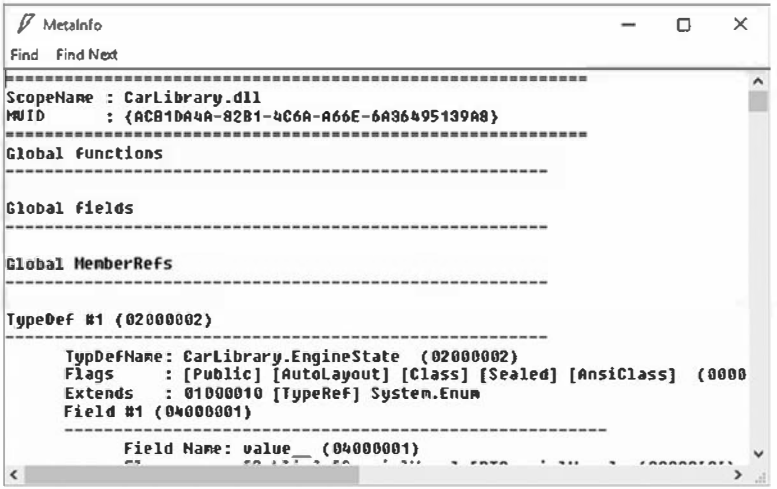


Рис. 15.1. Просмотр метаданных сборки с помощью утилиты ildasm.exe

Как видите, утилита ildasm.exe отображает метаданные типов .NET очень подробно (действительный двоичный формат гораздо компактнее). В действительности описание всех метаданных сборки CarLibrary.dll заняло бы несколько страниц. Однако для понимания вполне достаточно кратко взглянуть на некоторые ключевые описания метаданных сборки CarLibrary.dll.

На заметку! Не стоит слишком глубоко вникать в синтаксис каждого фрагмента метаданных .NET, приводимого в нескольких последующих разделах. Важно усвоить, что метаданные .NET являются исключительно дескриптивными и учитывают каждый внутренне определенный (и внешне ссылаемый) тип, который найден в заданной кодовой базе.

Просмотр (частичных) метаданных для перечисления EngineState

Каждый тип, определенный внутри текущей сборки, документируется с применением маркера TypeDef #n (где TypeDef — сокращение от *type definition* (определение типа)). Если описываемый тип использует какой-то тип, определенный в отдельной сборке .NET, то ссылаемый тип документируется с помощью маркера TypeRef #n (где TypeRef — сокращение от *type reference* (ссылка на тип)). Если хотите, то можете считать, что маркер TypeRef является указателем на полное определение метаданных ссылаемого типа во внешней сборке. Коротко говоря, метаданные .NET — это набор таблиц, явно помечающих все определения типов (TypeDef) и ссылаемые типы (TypeRef), которые могут быть просмотрены в окне метаданных утилиты ildasm.exe. В случае сборки CarLibrary.dll один из маркеров TypeDef представляет описание метаданных перечисления CarLibrary.EngineState (номер TypeDef у вас может отличаться; нумерация TypeDef основана на порядке, в котором компилятор C# обрабатывает файл):

```
TypeDef #2 (02000003)
-------------------------------------------------------
  TypDefName: CarLibrary.EngineState (02000003)
  Flags     : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass] (00000101)
  Extends   : 01000001 [TypeRef] System.Enum
```

```
  Field #1 (04000006)
-------------------------------------------------------
    Field Name: value__ (04000006)
    Flags     : [Public] [SpecialName] [RTSpecialName] (00000606)
    CallCnvntn: [FIELD]
    Field type: I4
  Field #2 (04000007)
-------------------------------------------------------
  Field Name: engineAlive (04000007)
  Flags     : [Public] [Static] [Literal] [HasDefault] (00008056)
  DefltValue: (I4) 0
  CallCnvntn: [FIELD]
  Field type: ValueClass CarLibrary.EngineState
...
```

Маркер `TypDefName` здесь служит для установления имени заданного типа, которым в рассматриваемом случае является специальное перечисление `CarLibrary.EngineState`. Маркер метаданных `Extends` применяется при документировании базового типа для заданного типа .NET (ссылаемого типа `System.Enum` в этом случае). Каждое поле перечисления помечается с использованием маркера `Field #n`. Ради краткости выше были приведены только метаданные для поля `CarLibrary.EngineState.engineAlive`.

Просмотр (частичных) метаданных для типа `Car`

Ниже показана часть метаданных класса `Car`, которая иллюстрирует следующие аспекты:

- как поля определены в терминах метаданных .NET;
- как методы документированы посредством метаданных .NET;
- как автоматическое свойство представлено в метаданных .NET.

```
TypeDef #3 (02000004)
-------------------------------------------------------
  TypDefName: CarLibrary.Car (02000004)
  Flags     : [Public] [AutoLayout] [Class] [Abstract]
              [AnsiClass] [BeforeFieldInit] (00100081)
  Extends   : 01000002 [TypeRef] System.Object
...
  Field #2 (0400000a)
-------------------------------------------------------
  Field Name: <PetName>k__BackingField (0400000A)
  Flags     : [Private] (00000001)
  CallCnvntn: [FIELD]
  Field type: String
...
  Method #1 (06000001)
-------------------------------------------------------
    MethodName: get_PetName (06000001)
    Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName] (00000886)
    RVA       : 0x000020d0
    ImplFlags : [IL] [Managed] (00000000)
    CallCnvntn: [DEFAULT]
    hasThis
    ReturnType: String
    No arguments.
...
```

```
  Method #2 (06000002)
-------------------------------------------------------
  MethodName: set_PetName (06000002)
  Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName] (00000886)
  RVA       : 0x000020e7
  ImplFlags : [IL] [Managed] (00000000)
  CallCnvntn: [DEFAULT]
  hasThis
  ReturnType: Void
  1 Arguments
    Argument #1: String
  1 Parameters
    (1) ParamToken : (08000001) Name : value flags: [none] (00000000)
...
  Property #1 (17000001)
-------------------------------------------------------
    Prop.Name : PetName (17000001)
    Flags     : [none] (00000000)
    CallCnvntn: [PROPERTY]
    hasThis
    ReturnType: String
    No arguments.
    DefltValue:
    Setter    : (06000002) set_PetName
    Getter    : (06000001) get_PetName
    0 Others
...
```

Прежде всего, метаданные класса `Car` указывают базовый класс этого типа (`System.Object`) и включают разнообразные флаги, которые описывают то, как тип был сконструирован (например, `[Public]`, `[Abstract]` и т.п.). Описания методов (вроде конструктора `Car`) содержат имя, возвращаемое значение и параметры.

Обратите внимание, что автоматическое свойство дает в результате сгенерированное компилятором закрытое поддерживающее поле (по имени `<PetName>k__BackingField`) и два сгенерированных компилятором метода (в случае свойства для чтения и записи) с именами `get_PetName()` и `set_PetName()`. Наконец, само свойство отображается на внутренние методы получения/установки с применением маркеров `Setter` и `Getter` метаданных .NET.

Исследование блока `TypeRef`

Вспомните, что метаданные сборки будут описывать не только набор внутренних типов (`Car`, `EngineState` и т.д.), но также любые внешние типы, на которые ссылаются внутренние типы. Например, с учетом того, что в сборке `CarLibrary.dll` определены два перечисления, метаданные типа `System.Enum` будут содержать следующий блок `TypeRef`:

```
TypeRef #1 (01000001)
-------------------------------------------------------
Token:             0x01000001
ResolutionScope:   0x23000001
TypeRefName:       System.Enum
```

Документирование определяемой сборки

Окно метаданных `ildasm.exe` также позволяет просматривать метаданные .NET, которые описывают саму сборку с использованием маркера `Assembly`. Как показано в приведенном далее (неполном) листинге, информация, документируемая внутри таблицы `Assembly`, совпадает с той, которую можно просматривать по щелчку на значке MANIFEST (Манифест). Ниже представлена часть метаданных манифеста сборки `CarLibrary.dll` (версии 2.0.0.0):

```
Assembly
-------------------------------------------------------
  Token: 0x20000001
  Name : CarLibrary
  Public Key     : 00 24 00 00 04 80 00 00 // Etc...
  Hash Algorithm : 0x00008004
  Major Version  : 0x00000002
  Minor Version  : 0x00000000
  Build Number   : 0x00000000
  Revision Number: 0x00000000
  Locale: <null>
  Flags : [PublicKey] ...
```

Документирование ссылаемых сборок

В дополнение к маркеру `Assembly` и набору блоков `TypeDef` и `TypeRef` в метаданных .NET также применяются маркеры `AssemblyRef #n` для документирования каждой внешней сборки. Поскольку в сборке `CarLibrary.dll` используется класс `System.Windows.Forms.MessageBox`, вы обнаружите следующий блок `AssemblyRef` для сборки `System.Windows.Forms`:

```
AssemblyRef #2 (23000002)
-------------------------------------------------------
  Token: 0x23000002
  Public Key or Token: b7 7a 5c 56 19 34 e0 89
  Name: System.Windows.Forms
  Version: 4.0.0.0
  Major Version: 0x00000004
  Minor Version: 0x00000000
  Build Number: 0x00000000
  Revision Number: 0x00000000
  Locale: <null>
  HashValue Blob:
  Flags: [none] (00000000)
```

Документирование строковых литералов

Последний полезный аспект, относящийся к метаданным .NET, связан с тем, что все строковые литералы в кодовой базе документируются внутри маркера `User Strings`:

```
User Strings
-------------------------------------------------------
70000001 : (11) L"Jamming {0}"
70000019 : (13) L"Quiet time..."
70000035 : (23) L"CarLibrary Version 2.0!"
70000065 : (14) L"Ramming speed!"
70000083 : (19) L"Faster is better..."
700000ab : ( 4) L"Eek!"
700000cd : (27) L"Your engine block exploded!"
```

На заметку! Как демонстрирует представленный выше листинг метаданных, всегда важно иметь в виду, что все строки прозрачно документируются в метаданных сборки. Это может привести к серьезным последствиям в плане безопасности, если вы применяете строковые литералы для хранения паролей, номеров кредитных карт или другой конфиденциальной информации.

У вас может возникнуть вопрос о том, каким образом задействовать эту информацию в разрабатываемых приложениях (в лучшем сценарии) или зачем вообще заботиться о метаданных (в худшем сценарии). Чтобы получить ответ, необходимо ознакомиться со службами рефлексии .NET. Следует отметить, что полезность рассматриваемых далее тем может стать ясной только ближе к концу главы, поэтому наберитесь терпения.

На заметку! В окне метаданных `ildasm.exe` вы также найдете несколько маркеров `CustomAttribute`, которые документируют атрибуты, применяемые внутри кодовой базы. О роли атрибутов .NET речь пойдет позже в главе.

Понятие рефлексии

В мире .NET *рефлексией* называется процесс обнаружения типов во время выполнения. Службы рефлексии дают возможность получать программно ту же самую информацию о метаданных, которую отображает утилита `ildasm.exe`, используя дружественную объектную модель. Например, посредством рефлексии можно извлечь список всех типов, содержащихся внутри заданной сборки `*.dll` или `*.exe`, в том числе методы, поля, свойства и события, которые определены конкретным типом. Можно также динамически получать набор интерфейсов, поддерживаемых заданным типом, параметры метода и другие относящиеся к ним детали (базовые классы, информацию о пространствах имен, данные манифеста и т.п.). Как любое пространство имен, `System.Reflection` (которое определено в сборке `mscorlib.dll`) содержит набор связанных типов. В табл. 15.1 описаны основные его члены, о которых вы должны знать.

Таблица 15.1. Избранные члены пространства имен `System.Reflection`

Тип	Описание
`Assembly`	Этот абстрактный класс содержит несколько членов, которые позволяют загружать, исследовать и манипулировать сборкой
`AssemblyName`	Этот класс позволяет выяснить многочисленные детали, связанные с идентичностью сборки (номер версии, информация о культуре и т.д.)
`EventInfo`	Этот абстрактный класс хранит информацию о заданном событии
`FieldInfo`	Этот абстрактный класс хранит информацию о заданном поле
`MemberInfo`	Этот абстрактный базовый класс определяет общее поведение для типов `EventInfo`, `FieldInfo`, `MethodInfo` и `PropertyInfo`
`MethodInfo`	Этот абстрактный класс содержит информацию о заданном методе
`Module`	Этот абстрактный класс позволяет получить доступ к заданному модулю внутри многофайловой сборки
`ParameterInfo`	Этот класс хранит информацию о заданном параметре
`PropertyInfo`	Этот абстрактный класс хранит информацию о заданном свойстве

Чтобы понять, каким образом задействовать пространство имен `System.Reflection` для программного чтения метаданных .NET, необходимо сначала ознакомиться с классом `System.Type`.

Класс System.Type

Класс System.Type определяет набор членов, которые могут применяться для исследования метаданных типа, большое количество которых возвращают типы из пространства имен System.Reflection. Например, метод Type.GetMethods() возвращает массив объектов MethodInfo, метод Type.GetFields() — массив объектов FieldInfo и т.д. Полный перечень членов, доступных в System.Type, довольно велик, но в табл. 15.2 приведен список избранных членов, поддерживаемых System.Type (за исчерпывающими сведениями обращайтесь в документацию .NET Framework 4.6 SDK).

Таблица 15.2. Избранные члены System.Type

Член	Описание
IsAbstract IsArray IsClass IsCOMObject IsEnum IsGenericTypeDefinition IsGenericParameter IsInterface IsPrimitive IsNestedPrivate IsNestedPublic IsSealed IsValueType	Эти свойства позволяют выяснять базовые характерные черты типа, на который осуществляется ссылка (например, является ли он абстрактной сущностью, массивом, вложенным классом и т.п.)
GetConstructors() GetEvents() GetFields() GetInterfaces() GetMembers() GetMethods() GetNestedTypes() GetProperties()	Эти методы позволяют получать массив интересующих элементов (интерфейсов, методов, свойств и т.д.). Каждый метод возвращает связанный массив (например, GetFields() возвращает массив FieldInfo, метод GetMethods() — массив MethodInfo и т.д.). Следует отметить, что для каждого метода предусмотрена версия с именем в единственном числе (например, GetMethod(), GetProperty() и т.п.), которая позволяет извлекать специфический элемент по имени, а не массив всех связанных элементов
FindMembers()	Этот метод возвращает массив объектов MemberInfo на основе указанного критерия поиска
GetType()	Этот статический метод возвращает экземпляр Type с заданным строковым именем
InvokeMember()	Этот метод позволяет выполнять "позднее связывание" для указанного элемента. Вы узнаете о позднем связывании далее в главе

Получение информации о типе с помощью System.Object.GetType()

Экземпляр класса Type можно получать разнообразными способами. Тем не менее, есть одна вещь, которую делать невозможно — создавать объект Type напрямую, используя ключевое слово new, т.к. Type является абстрактным классом. Что касается первого способа, вспомните, что в классе System.Object определен метод GetType(), который возвращает экземпляр класса Type, представляющий метаданные текущего объекта:

```
// Получить информацию о типе с применением экземпляра SportsCar.
SportsCar sc = new SportsCar();
Type t = sc.GetType();
```

Очевидно, такой подход будет работать, только если подвергаемый рефлексии тип (SportsCar в данном случае) известен на этапе компиляции и в памяти присутствует его экземпляр. С учетом этого ограничения должно быть понятно, почему инструменты вроде ildasm.exe не получают информацию о типе, непосредственно вызывая метод System.Object.GetType() для каждого типа; ведь утилита ildasm.exe не компилировалась вместе с вашими специальными сборками.

Получение информации о типе с помощью typeof()

Следующий способ получения информации о типе предполагает применение операции typeof:

```
// Получить информацию о типе с использованием операции typeof.
Type t = typeof(SportsCar);
```

В отличие от метода System.Object.GetType() операция typeof удобна тем, что она не требует предварительного создания экземпляра объекта перед получением информации о типе. Однако кодовой базе по-прежнему должно быть известно об исследуемом типе на этапе компиляции, поскольку typeof ожидает получения строго типизированного имени типа.

Получение информации о типе с помощью System.Type.GetType()

Для получения информации о типе в более гибкой манере можно вызывать статический метод GetType() класса System.Type и указывать полностью заданное строковое имя типа, который планируется изучить. При таком подходе знать тип, из которого будут извлекаться метаданные, на этапе компиляции *не* нужно, т.к. метод Type.GetType() принимает в качестве параметра экземпляр вездесущего класса System.String.

На заметку! Когда речь идет о том, что при вызове метода Type.GetType() знание типа на этапе компиляции не требуется, имеется в виду тот факт, что этот метод может принимать любое строковое значение (а не строго типизированную переменную). Разумеется, знать имя типа в строковом формате по-прежнему необходимо!

Метод Type.GetType() перегружен, позволяя указывать два булевских параметра, из которых один управляет тем, должно ли генерироваться исключение, если тип не удается найти, а второй отвечает за то, должен ли учитываться регистр символов в строке. В целях иллюстрации рассмотрим следующий код:

```
// Получить информацию о типе с использованием статического метода Type.GetType()
// (не генерировать исключение, если тип SportsCar не удается найти,
// и игнорировать регистр символов).
Type t = Type.GetType("CarLibrary.SportsCar", false, true);
```

В приведенном примере обратите внимание на то, что в строке, передаваемой методу GetType(), никак не упоминается сборка, внутри которой содержится интересующий тип. В этом случае делается предположение о том, что тип определен внутри сборки, выполняющейся в текущий момент. Тем не менее, когда необходимо получить метаданные для типа из внешней закрытой сборки, строковый параметр форматируется с использованием полностью заданного имени типа, за которым следует запятая и дружественное имя сборки, содержащей этот тип:

```
// Получить информацию о типе из внешней сборки.
Type t = Type.GetType("CarLibrary.SportsCar, CarLibrary");
```

Кроме того, в передаваемой методу GetType() строке может быть указан символ "плюс" (+) для обозначения *вложенного типа*. Пусть необходимо получить информацию о типе перечисления (SpyOptions), вложенного в класс по имени JamesBondCar. Для этого можно написать следующий код:

```
// Получить информацию о типе перечисления, вложенного в текущую сборку.
Type t = Type.GetType("CarLibrary.JamesBondCar+SpyOptions");
```

Построение специального средства для просмотра метаданных

Чтобы продемонстрировать базовый процесс рефлексии (и полезность класса System.Type), создадим новый проект консольного приложения по имени MyTypeViewer. Приложение будет отображать детали методов, свойств, полей и поддерживаемых интерфейсов (в дополнение к другим интересным данным) для любого типа внутри mscorlib.dll (вспомните, что все приложения .NET автоматически получают доступ к этой основной библиотеке классов платформы) или типа внутри самого приложения MyTypeViewer. После создания приложения не забудьте импортировать пространство имен System.Reflection:

```
// Это пространство имен должно импортироваться для выполнения любой рефлексии!
using System.Reflection;
```

Рефлексия методов

Класс Program потребуется модифицировать для определения в нем нескольких статических методов, каждый из которых принимает единственный параметр System.Type и возвращает void. Сначала определим метод ListMethods(), который выводит имена методов, определенных во входном типе. Обратите внимание, что Type.GetMethods() возвращает массив объектов System.Reflection.MethodInfo, по которому можно осуществлять проход с помощью цикла foreach:

```
// Отобразить имена методов в типе.
static void ListMethods(Type t)
{
  Console.WriteLine("***** Methods *****");
  MethodInfo[] mi = t.GetMethods();
  foreach(MethodInfo m in mi)
    Console.WriteLine("->{0}", m.Name);
  Console.WriteLine();
}
```

Здесь просто выводится имя метода с применением свойства MethodInfo.Name. Как не трудно догадаться, класс MethodInfo имеет множество дополнительных членов, которые позволяют выяснить, является метод статическим, виртуальным, обобщенным или абстрактным. Вдобавок тип MethodInfo дает возможность получить информацию о возвращаемом значении и наборе параметров метода. Чуть позже реализация ListMethods() будет немного улучшена.

При желании для перечисления имен методов можно было бы также построить подходящий запрос LINQ. Вспомните из главы 12, что технология LINQ to Object позволяет создавать строго типизированные запросы и применять их к коллекциям объектов в памяти. В качестве эмпирического правила запомните, что при обнаружении блоков с программной логикой циклов или принятия решений можно использовать соответствующий запрос LINQ. К примеру, предыдущий метод можно было бы переписать так:

```
static void ListMethods(Type t)
{
  Console.WriteLine("***** Methods *****");
  var methodNames = from n in t.GetMethods() select n.Name;
  foreach (var name in methodNames)
    Console.WriteLine("->{0}", name);
  Console.WriteLine();
}
```

Рефлексия полей и свойств

Реализация метода `ListFields()` похожа. Единственным заметным отличием является вызов `Type.GetFields()` и результирующий массив элементов `FieldInfo`. И снова для простоты выводятся только имена каждого поля с применением запроса LINQ:

```
// Отобразить имена полей в типе.
static void ListFields(Type t)
{
  Console.WriteLine("***** Fields *****");
  var fieldNames = from f in t.GetFields() select f.Name;
  foreach (var name in fieldNames)
    Console.WriteLine("->{0}", name);
  Console.WriteLine();
}
```

Логика для отображения имен свойств типа аналогична:

```
// Отобразить имена свойств в типе.
static void ListProps(Type t)
{
  Console.WriteLine("***** Properties *****");
  var propNames = from p in t.GetProperties() select p.Name;
  foreach (var name in propNames)
    Console.WriteLine("->{0}", name);
  Console.WriteLine();
}
```

Рефлексия реализованных интерфейсов

Следующим создается метод по имени `ListInterfaces()`, который будет выводить имена любых интерфейсов, поддерживаемых входным типом. Один интересный момент здесь в том, что вызов `GetInterfaces()` возвращает массив объектов `System.Type`! Это вполне логично, поскольку интерфейсы на самом деле являются типами:

```
// Отобразить имена интерфейсов, которые реализует тип.
static void ListInterfaces(Type t)
{
  Console.WriteLine("***** Interfaces *****");
  var ifaces = from i in t.GetInterfaces() select i;
  foreach(Type i in ifaces)
    Console.WriteLine("->{0}", i.Name);
}
```

На заметку! Имейте в виду, что большинство методов "получения" в `System.Type` (`GetMethods()`, `GetInterfaces()` и т.д.) перегружены, чтобы позволить указывать значения из перечисления `BindingFlags`. В итоге появляется высокий уровень контроля над тем, что в точности необходимо искать (например, только статические члены, только открытые члены, включать закрытые члены и т.д.). За более подробной информацией обращайтесь в документацию .NET Framework 4.6 SDK.

Отображение разнообразных дополнительных деталей

В качестве последнего, но не менее важного действия, осталось реализовать финальный вспомогательный метод, который будет отображать различные статистические данные о входном типе (является ли он обобщенным, какой его базовый класс, запечатан ли он и т.п.):

```
// Просто ради полноты картины.
static void ListVariousStats(Type t)
{
  Console.WriteLine("***** Various Statistics *****");
  Console.WriteLine("Base class is: {0}", t.BaseType);
                    // Базовый класс
  Console.WriteLine("Is type abstract? {0}", t.IsAbstract);
                    // Абстрактный?
  Console.WriteLine("Is type sealed? {0}", t.IsSealed);
                    // Запечатанный?
  Console.WriteLine("Is type generic? {0}", t.IsGenericTypeDefinition);
                    // Обобщенный?
  Console.WriteLine("Is type a class type? {0}", t.IsClass);
                    // Класс?
  Console.WriteLine();
}
```

Реализация метода `Main()`

Метод `Main()` класса `Program` запрашивает у пользователя полностью заданное имя типа. После получения этих строковых данных они передаются методу `Type.GetType()`, а результирующий объект `System.Type` отправляется каждому вспомогательному методу. Процесс повторяется до тех пор, пока пользователь не введет `Q` для прекращения работы приложения.

```
static void Main(string[] args)
{
  Console.WriteLine("***** Welcome to MyTypeViewer *****");
  string typeName = "";

  do
  {
    Console.WriteLine("\nEnter a type name to evaluate");
                      // Предложить ввести имя типа
    Console.Write("or enter Q to quit: "); // или Q для завершения.

    // Получить имя типа.
    typeName = Console.ReadLine();

    // Пользователь желает завершить программу?
    if (typeName.ToUpper() == "Q")
    {
      break;
    }

    // Попробовать отобразить информацию о типе.
    try
    {
      Type t = Type.GetType(typeName);
      Console.WriteLine("");
      ListVariousStats(t);
```

```
      ListFields(t);
      ListProps(t);
      ListMethods(t);
      ListInterfaces(t);
    }
    catch
    {
      Console.WriteLine("Sorry, can't find type");  // Не удается найти тип.
    }
  } while (true);
}
```

На этой стадии приложение `MyTypeViewer.exe` готово к тестовому запуску. Давайте запустим его и введем следующие полностью заданные имена (не забывая, что выбранный способ вызова `Type.GetType()` требует ввода строковых имен с учетом регистра):

- `System.Int32`
- `System.Collections.ArrayList`
- `System.Threading.Thread`
- `System.Void`
- `System.IO.BinaryWriter`
- `System.Math`
- `System.Console`
- `MyTypeViewer.Program`

Ниже показан частичный вывод при указании строкового имени `System.Math`:

```
***** Welcome to MyTypeViewer *****

Enter a type name to evaluate
or enter Q to quit: System.Math

***** Various Statistics *****
Base class is: System.Object
Is type abstract? True
Is type sealed? True
Is type generic? False
Is type a class type? True

***** Fields *****
->PI
->E

***** Properties *****

***** Methods *****
->Acos
->Asin
->Atan
->Atan2
->Ceiling
->Ceiling
->Cos
...
```

Рефлексия обобщенных типов

При вызове `Type.GetType()` для получения описаний метаданных обобщенных типов должен использоваться специальный синтаксис, включающий символ обратной одинарной кавычки (`), за которым следует числовое значение, представляющее количество поддерживаемых параметров типа. Например, чтобы вывести описание метаданных `System.Collections.Generic.List<T>`, приложению потребуется передать следующую строку:

```
System.Collections.Generic.List`1
```

Здесь указано числовое значение 1, т.к. `List<T>` имеет только один параметр типа. Однако для применения рефлексии к типу `Dictionary<TKey, TValue>` понадобится предоставить значение 2:

```
System.Collections.Generic.Dictionary`2
```

Рефлексия параметров и возвращаемых значений методов

Давайте проведем небольшое усовершенствование приложения. В частности, мы модифицируем вспомогательный метод `ListMethods()` так, чтобы он выводил не только имя заданного метода, но также возвращаемый тип и типы входных параметров. Для решения этих задач тип `MethodInfo` предоставляет свойство `ReturnType` и метод `GetParameters()`. В следующем измененном коде обратите внимание на то, что строка с информацией о типе и имени каждого параметра строится с использованием вложенного цикла `foreach` (LINQ не применяется):

```
static void ListMethods(Type t)
{
  Console.WriteLine("***** Methods *****");
  MethodInfo[] mi = t.GetMethods();
  foreach (MethodInfo m in mi)
  {
    // Получить информацию о возвращаемом типе.
    string retVal = m.ReturnType.FullName;
    string paramInfo = "( ";
    // Получить информацию о параметрах.
    foreach (ParameterInfo pi in m.GetParameters())
    {
      paramInfo += string.Format("{0} {1} ", pi.ParameterType, pi.Name);
    }
    paramInfo += " )";
    // Отобразить базовую сигнатуру метода.
    Console.WriteLine("->{0} {1} {2}", retVal, m.Name, paramInfo);
  }
  Console.WriteLine();
}
```

Если теперь запустить обновленное приложение, то обнаружится, что методы заданного типа будут описаны более подробно. Например, в случае ввода типа `System.Object` отобразятся следующие описания методов:

```
***** Methods *****
->System.String ToString ( )
->System.Boolean Equals ( System.Object obj )
->System.Boolean Equals ( System.Object objA System.Object objB )
->System.Boolean ReferenceEquals ( System.Object objA System.Object objB )
->System.Int32 GetHashCode ( )
->System.Type GetType ( )
```

Текущая реализация `ListMethods()` удобна тем, что позволяет исследовать непосредственно каждый параметр и возвращаемый тип методов с использованием объектной модели `System.Reflection`. В качестве исключительного сокращения имейте в виду, что все типы `XXXInfo` (`MethodInfo`, `PropertyInfo`, `EventInfo` и т.д.) переопределяют метод `ToString()` с целью отображения сигнатуры запрашиваемого элемента. Таким образом, метод `ListMethods()` можно было бы реализовать в следующем виде (здесь снова применяется запрос LINQ, выбирающий все объекты `MethodInfo`, а не только значения `Name`):

```
static void ListMethods(Type t)
{
  Console.WriteLine("***** Methods *****");
  var methodNames = from n in t.GetMethods() select n;
  foreach (var name in methodNames)
    Console.WriteLine("->{0}", name);
  Console.WriteLine();
}
```

Довольно интересно, не так ли? Очевидно, что пространство имен `System.Reflection` и класс `System.Type` позволяют выполнять рефлексию многих других аспектов типа помимо тех, которые в настоящий момент отображает приложение `MyTypeViewer`. Как и можно было ожидать, допускается также получать информацию о событиях, поддерживаемых типом, любых обобщенных параметрах для заданных членов и десятки других деталей.

Итак, к этому моменту создан (довольно мощный) браузер объектов. Главное ограничение состоит в том, что он не позволяет подвергать рефлексии ничего кроме текущей сборки (`MyTypeViewer`) и всегда доступной сборки `mscorlib.dll`. Возникает вопрос: как построить приложение, способное загружать (и проводить рефлексию) сборки, на которые отсутствуют ссылки на этапе компиляции? Об этом пойдет речь в следующем разделе.

Исходный код. Проект `MyTypeViewer` доступен в подкаталоге `Chapter_15`.

Динамически загружаемые сборки

В главе 14 вы узнали все о том, как среда CLR заглядывает в манифест сборки во время зондирования внешних сборок, на которые ссылается текущая сборка. Тем не менее, во многих случаях сборки необходимо загружать на лету программным образом, даже если в манифесте отсутствуют записи о них. Формально процесс загрузки внешних сборок по требованию называется *динамической загрузкой*.

В пространстве имен `System.Reflection` определен класс `Assembly`, с использованием которого можно динамически загружать сборку, а также исследовать связанные с ней свойства. С помощью `Assembly` можно динамически загружать закрытые или разделяемые сборки, равно как и сборки, расположенные в произвольных местах. По существу класс `Assembly` предлагает методы (в частности, `Load()` и `LoadFrom()`), которые позволяют программно предоставлять информацию того же рода, что и в клиентских файлах `*.config`.

Чтобы проиллюстрировать динамическую загрузку, создадим новый проект консольного приложения по имени `ExternalAssemblyReflector`. Наша задача связана с написанием метода `Main()`, который будет запрашивать у пользователя дружественное имя сборки с целью ее динамической загрузки. Ссылка на `Assembly` будет передаваться вспомогательному методу под названием `DisplayTypes()`, который просто выведет

имена всех содержащихся в сборке классов, интерфейсов, структур, перечислений и делегатов. Вот необходимый код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;
using System.IO;  // Для определения FileNotFoundException.

namespace ExternalAssemblyReflector
{
  class Program
  {
    static void DisplayTypesInAsm(Assembly asm)
    {
      Console.WriteLine("\n***** Types in Assembly *****");
      Console.WriteLine("->{0}", asm.FullName);
      Type[] types = asm.GetTypes();
      foreach (Type t in types)
        Console.WriteLine("Type: {0}", t);
      Console.WriteLine("");
    }

    static void Main(string[] args)
    {
      Console.WriteLine("***** External Assembly Viewer *****");

      string asmName = "";
      Assembly asm = null;
      do
      {
        Console.WriteLine("\nEnter an assembly to evaluate");
                          // Предложить ввести имя сборки
        Console.Write("or enter Q to quit: ");  // или Q для завершения.

        // Получить имя сборки.
        asmName = Console.ReadLine();

        // Пользователь желает завершить программу?
        if (asmName.ToUpper() == "Q")
        {
          break;
        }

        // Попробовать загрузить сборку.
        try
        {
          asm = Assembly.Load(asmName);
          DisplayTypesInAsm(asm);
        }
        catch
        {
          Console.WriteLine("Sorry, can't find assembly.");
                          // Сборка не найдена.
        }
      } while (true);
    }
  }
}
```

Обратите внимание, что статическому методу `Assembly.Load()` передается только дружественное имя сборки, которую требуется загрузить в память. Таким образом, чтобы подвергнуть рефлексии сборку `CarLibrary.dll`, понадобится скопировать двоичный файл `CarLibrary.dll` в подкаталог `bin\Debug` внутри каталога приложения `ExternalAssemblyReflector`. После этого можно будет получить примерно такой вывод:

```
***** External Assembly Viewer *****

Enter an assembly to evaluate
or enter Q to quit: CarLibrary

***** Types in Assembly *****
->CarLibrary, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=33a2bc294331e8b9
Type: CarLibrary.MusicMedia
Type: CarLibrary.EngineState
Type: CarLibrary.Car
Type: CarLibrary.SportsCar
Type: CarLibrary.MiniVan
```

Если вы хотите сделать приложение `ExternalAssemblyReflector` более гибким, то модифицируйте код так, чтобы загрузка внешней сборки производилась с применением метода `Assembly.LoadFrom()` вместо `Assembly.Load()`:

```
try
{
  asm = Assembly.LoadFrom(asmName);
  DisplayTypesInAsm(asm);
}
```

Это даст возможность вводить абсолютный путь к интересующей сборке (скажем, `C:\MyApp\MyAsm.dll`). По существу метод `Assembly.LoadFrom()` позволяет программно предоставлять значение <codeBase>. Теперь консольному приложению можно передавать полный путь. Если сборка `CarLibrary.dll` находится в каталоге `C:\MyCode`, то будет получен следующий вывод:

```
***** External Assembly Viewer *****

Enter an assembly to evaluate
or enter Q to quit: C:\MyCode\CarLibrary.dll

***** Types in Assembly *****
->CarLibrary, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=33a2bc294331e8b9
Type: CarLibrary.EngineState
Type: CarLibrary.Car
Type: CarLibrary.SportsCar
Type: CarLibrary.MiniVan
```

Исходный код. Проект `ExternalAssemblyReflector` доступен в подкаталоге `Chapter_15`.

Рефлексия разделяемых сборок

Метод `Assembly.Load()` имеет несколько перегруженных версий. Одна из них разрешает указывать значение культуры (для локализованных сборок), а также номер версии и значение маркера открытого ключа (для разделяемых сборок). Коллективно многочисленные элементы, идентифицирующие сборку, называются *отображаемым именем*.

Форматом отображаемого имени является строка пар "имя-значение", разделенных запятыми, которая начинается с дружественного имени сборки, а за ним следуют необязательные квалификаторы (в любом порядке). Вот как выглядит шаблон (необязательные элементы указаны в круглых скобках):

```
Имя (,Version = <старший номер>.<младший номер>.<номер сборки>.<номер редакции>)
(,Culture = <маркер культуры>)
(,PublicKeyToken = <маркер открытого ключа>)
```

При создании отображаемого имени соглашение `PublicKeyToken=null` отражает тот факт, что требуется связывание и сопоставление со сборкой, не имеющей строгого имени. Вдобавок `Culture=""` указывает, что сопоставление должно осуществляться со стандартной культурой целевой машины, например:

```
// Загрузить версию 1.0.0.0 сборки CarLibrary, используя стандартную культуру.
Assembly a =
  Assembly.Load(@"CarLibrary, Version=1.0.0.0, PublicKeyToken=null, Culture="""");
```

Кроме того, следует иметь в виду, что пространство имен `System.Reflection` предлагает тип `AssemblyName`, который позволяет представлять показанную выше строковую информацию в удобной объектной переменной. Обычно этот класс применяется вместе с классом `System.Version`, который представляет собой объектно-ориентированную оболочку для номера версии сборки. После создания отображаемого имени его затем можно передавать перегруженной версии метода `Assembly.Load()`:

```
// Применение типа AssemblyName для определения отображаемого имени.
AssemblyName asmName;
asmName = new AssemblyName();
asmName.Name = "CarLibrary";
Version v = new Version("1.0.0.0");
asmName.Version = v;
Assembly a = Assembly.Load(asmName);
```

Для загрузки разделяемой сборки из GAC в параметре метода `Assembly.Load()` должно быть указано значение `PublicKeyToken`. Например, пусть создан новый проект консольного приложения по имени `SharedAsmReflector`, и нужно загрузить версию 4.0.0.0 сборки `System.Windows.Forms.dll`, предоставляемую библиотеками базовых классов .NET. Поскольку количество типов в данной сборке довольно велико, приложение будет выводить только имена открытых перечислений, используя простой запрос LINQ:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;
using System.IO;

namespace SharedAsmReflector
{
  public class SharedAsmReflector
  {
    private static void DisplayInfo(Assembly a)
    {
      Console.WriteLine("***** Info about Assembly *****");
      Console.WriteLine("Loaded from GAC? {0}", a.GlobalAssemblyCache);
                        // Загружена из GAC?
```

```
      Console.WriteLine("Asm Name: {0}", a.GetName().Name);
                        // Имя сборки
      Console.WriteLine("Asm Version: {0}", a.GetName().Version);
                        // Версия сборки
      Console.WriteLine("Asm Culture: {0}",
                        // Культура сборки
        a.GetName().CultureInfo.DisplayName);
      Console.WriteLine("\nHere are the public enums:");
                         // Список открытых перечислений
      // Использовать запрос LINQ для нахождения открытых перечислений.
      Type[] types = a.GetTypes();
      var publicEnums = from pe in types where pe.IsEnum &&
                        pe.IsPublic select pe;
      foreach (var pe in publicEnums)
      {
        Console.WriteLine(pe);
      }
    }
    static void Main(string[] args)
    {
      Console.WriteLine("***** The Shared Asm Reflector App *****\n");
      // Загрузить System.Windows.Forms.dll из GAC.
      string displayName = null;
      displayName = "System.Windows.Forms," +
        "Version=4.0.0.0," +
        "PublicKeyToken=b77a5c561934e089," +
        @"Culture=""";
      Assembly asm = Assembly.Load(displayName);
      DisplayInfo(asm);
      Console.WriteLine("Done!");
      Console.ReadLine();
    }
  }
}
```

Исходный код. Проект `SharedAsmReflector` доступен в подкаталоге `Chapter_15`.

К настоящему моменту вы должны уметь работать с некоторыми основными членами пространства имен `System.Reflection` для получения метаданных во время выполнения. Конечно, необходимость в самостоятельном построении специальных браузеров объектов в повседневной практике вряд ли будет возникать часто. Однако не забывайте, что службы рефлексии являются основой для нескольких распространенных действий программирования, включая *позднее связывание*.

Позднее связывание

Позднее связывание — это прием, который позволяет создавать экземпляр заданного типа и обращаться к его членам во время выполнения без необходимости в жестком кодировании факта его существования на этапе компиляции. При построении приложения, в котором производится позднее связывание с типом из внешней сборки, нет причин устанавливать ссылку на эту сборку; следовательно, в манифесте вызывающего кода она прямо не указывается.

На первый взгляд значимость позднего связывания оценить нелегко. Действительно, если есть возможность выполнить "раннее связывание" с объектом (например, добавить ссылку на сборку и выделить память под экземпляр типа с помощью ключевого слова new), то именно так следует поступать. Причина в том, что ранее связывание позволяет выявлять ошибки на этапе компиляции, а не во время выполнения. Тем не менее, позднее связывание играет важную роль в любом расширяемом приложении, которое может строиться. Пример построения такого "расширяемого" приложения будет приведен в конце главы, в разделе "Построение расширяемого приложения", а пока давайте изучим роль класса Activator.

Класс System.Activator

Класс System.Activator (определенный в mscorlib.dll) играет ключевую роль в процессе позднего связывания .NET. В текущем примере нам интересен только метод Activator.CreateInstance(), который применяется для создания экземпляра типа в стиле позднего связывания. Этот метод имеет несколько перегруженных версий, обеспечивая достаточно высокую гибкость. Самая простая версия метода CreateInstance() принимает действительный объект Type, описывающий сущность, которую необходимо разместить в памяти на лету.

Создадим новый проект консольного приложения по имени LateBindingApp и с помощью ключевого слова using импортируем в него пространства имен System.IO и System.Reflection. Теперь модифицируем класс Program следующим образом:

```
// Это приложение будет загружать внешнюю сборку и
// создавать объект, используя позднее связывание.
public class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Late Binding *****");
    // Попробовать загрузить локальную копию CarLibrary.
    Assembly a = null;
    try
    {
      a = Assembly.Load("CarLibrary");
    }
    catch(FileNotFoundException ex)
    {
      Console.WriteLine(ex.Message);
      return;
    }
    if(a != null)
      CreateUsingLateBinding(a);
    Console.ReadLine();
  }
  static void CreateUsingLateBinding(Assembly asm)
  {
    try
    {
      // Получить метаданные для типа Minivan.
      Type miniVan = asm.GetType("CarLibrary.MiniVan");
      // Создать экземпляр Minivan на лету.
      object obj = Activator.CreateInstance(miniVan);
      Console.WriteLine("Created a {0} using late binding!", obj);
    }
```

```
    catch(Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

Перед запуском нового приложения понадобится вручную скопировать файл `CarLibrary.dll` в подкаталог `bin\Debug` внутри каталога этого приложения. Причина в том, что в коде вызывается метод `Assembly.Load()`, т.е. среда CLR будет зондировать только каталог клиента (при желании можно было бы вызвать метод `Assembly.LoadFrom()` и вводить полный путь к сборке, но в данном случае в этом нет нужды).

На заметку! В рассматриваемом примере не добавляйте ссылку на сборку `CarLibrary.dll` с использованием Visual Studio! Такое действие приведет к помещению в манифест клиента записи о данной библиотеке. Вся суть позднего связывания заключается в попытке создания объекта, который не известен на этапе компиляции.

Обратите внимание, что метод `Activator.CreateInstance()` возвращает экземпляр `System.Object`, а не строго типизированный объект `MiniVan`. Следовательно, если применить к переменной `obj` операцию точки, то члены класса `MiniVan` не будут видны. На первый взгляд может показаться, что проблему удастся решить с помощью явного приведения:

```
// Привести к типу MiniVan, чтобы получить доступ к его членам?
// Нет! Компилятор сообщит об ошибке!
object obj = (MiniVan)Activator.CreateInstance(minivan);
```

Однако из-за того, что в приложение не была добавлена ссылка на сборку `CarLibrary.dll`, использовать ключевое слово `using` для импортирования пространства имен `CarLibrary` нельзя, а значит и невозможно указывать `MiniVan` в операции приведения! Не забывайте, что смысл позднего связывания — создание экземпляров типов, о которых на этапе компиляции ничего не известно. Учитывая это, возникает вопрос: как вызывать методы объекта `MiniVan`, сохраненного в ссылке на `System.Object`? Ответ: конечно, с помощью рефлексии.

Вызов методов без параметров

Предположим, что требуется вызвать метод `TurboBoost()` объекта `MiniVan`. Как вы наверняка помните, этот метод переводит двигатель в нерабочее состояние и затем отображает окно с соответствующим сообщением. Первый шаг заключается в получении объекта `MethodInfo` для метода `TurboBoost()` посредством `Type.GetMethod()`. Имея результирующий объект `MethodInfo`, можно вызвать `MiniVan.TurboBoost()` с помощью метода `Invoke()`. Метод `MethodInfo.Invoke()` требует указания всех параметров, которые подлежат передаче методу, представленному `MethodInfo`. Параметры задаются в виде массива объектов `System.Object` (т.к. они могут быть самыми разнообразными сущностями).

Поскольку метод `TurboBoost()` не принимает параметров, можно просто передать `null` (т.е. сообщить, что вызываемый метод не имеет параметров). Модифицируем метод `CreateUsingLateBinding()` следующим образом:

```
static void CreateUsingLateBinding(Assembly asm)
{
```

```
    try
    {
      // Получить метаданные для типа Minivan.
      Type miniVan = asm.GetType("CarLibrary.MiniVan");
      // Создать объект MiniVan на лету.
      object obj = Activator.CreateInstance(miniVan);
      Console.WriteLine("Created a {0} using late binding!", obj);
      // Получить информацию о TurboBoost.
      MethodInfo mi = miniVan.GetMethod("TurboBoost");
      // Вызвать метод (null означает отсутствие параметров).
      mi.Invoke(obj, null);
    }
    catch(Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
```

Рис. 15.2. Вызов метода через позднее связывание

Теперь после запуска приложения в результате вызова метода `TurboBoost()` отображается окно с сообщением (рис. 15.2).

Вызов методов с параметрами

Когда позднее связывание нужно применять для вызова метода, ожидающего параметры, аргументы потребуется упаковать в слабо типизированный массив `object`. Вспомните, что в версии 2.0.0.0 библиотеки `CarLibrary.dll` был определен следующий метод класса `Car`:

```
public void TurnOnRadio(bool musicOn, MusicMedia mm)
{
  if (musicOn)
    MessageBox.Show(string.Format("Jamming {0}", mm));
  else
    MessageBox.Show("Quiet time...");
}
```

Метод `TurnOnRadio()` принимает два параметра: булевское значение, которое указывает, должна ли быть включена музыкальная система в автомобиле, и перечисление, представляющее тип музыкального проигрывателя. Как упоминалось ранее, это перечисление выглядит так:

```
public enum MusicMedia
{
  musicCd,      // 0
  musicTape,    // 1
  musicRadio,   // 2
  musicMp3      // 3
}
```

Ниже приведен код нового метода класса `Program`, в котором вызывается `TurnOnRadio()`. Обратите внимание на использование внутренних числовых значений перечисления `MusicMedia`:

```
static void InvokeMethodWithArgsUsingLateBinding(Assembly asm)
{
```

```
    try
    {
      // Получить описание метаданных для типа SportsCar.
      Type sport = asm.GetType("CarLibrary.SportsCar");

      // Создать объект типа SportsCar.
      object obj = Activator.CreateInstance(sport);

      // Вызвать метод TurnOnRadio() с аргументами.
      MethodInfo mi = sport.GetMethod("TurnOnRadio");
      mi.Invoke(obj, new object[] { true, 2 });
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
```

В идеале к настоящему времени вы уже видите отношения между рефлексией, динамической загрузкой и поздним связыванием. Естественно, кроме раскрытых здесь возможностей API-интерфейс рефлексии предлагает много дополнительных средств, но вы уже должны быть в хорошей форме, чтобы погрузиться в дальнейшие детали.

Вас все еще может интересовать вопрос: *когда* эти приемы должны применяться в разрабатываемых приложениях? Ответ прояснится ближе к концу главы, а пока мы займемся исследованием роли атрибутов .NET.

Исходный код. Проект `LateBindingApp` доступен в подкаталоге `Chapter_15`.

Роль атрибутов .NET

Как было показано в начале главы, одной из задач компилятора .NET является генерация описаний метаданных для всех определяемых типов и для типов, на которые имеются ссылки. Помимо стандартных метаданных, содержащихся в любой сборке, платформа .NET предоставляет программистам способ встраивания в сборку дополнительных метаданных с использованием *атрибутов*. Выражаясь кратко, атрибуты — это всего лишь аннотации кода, которые могут применяться к заданному типу (классу, интерфейсу, структуре и т.п.), члену (свойству, методу и т.д.), сборке или модулю.

Атрибуты .NET представляют собой типы классов, расширяющих абстрактный базовый класс `System.Attribute`. По мере изучения пространств имен .NET вы найдете много предопределенных атрибутов, которые можно использовать в своих приложениях. Более того, вы также можете строить собственные атрибуты для дополнительного уточнения поведения своих типов путем создания нового типа, производного от `Attribute`.

Библиотека базовых классов .NET предлагает множество атрибутов в различных пространствах имен. Некоторые (но, безусловно, далеко не все) предопределенные атрибуты описаны в табл. 15.3.

Важно понимать, что когда вы применяете атрибуты в своем коде, встроенные метаданные по существу бесполезны до тех пор, пока другая часть программного обеспечения явно не запросит эту информацию посредством рефлексии. В противном случае метаданные, встроенные в сборку, игнорируются и не причиняют никакого вреда.

Таблица 15.3. Небольшое число избранных предопределенных атрибутов

Атрибут	Описание
`[CLSCompliant]`	Заставляет аннотированный элемент соответствовать правилам CLS (Common Language Specification — общеязыковая спецификация). Вспомните, что совместимые с CLS типы могут гарантированно использоваться во всех языках программирования .NET
`[DllImport]`	Позволяет коду .NET обращаться к любой неуправляемой библиотеке кода C или C++, включая API-интерфейс операционной системы. Обратите внимание, что при взаимодействии с программным обеспечением, основанным на COM, этот атрибут не применяется
`[Obsolete]`	Помечает устаревший тип или член. Если другие программисты попытаются использовать такой элемент, то они получат соответствующее предупреждение от компилятора
`[Serializable]`	Помечает класс или структуру как сериализируемую, что означает возможность сохранения своего текущего состояния в потоке
`[NonSerialized]`	Указывает, что заданное поле в классе или структуре не должно сохраняться в процессе сериализации
`[ServiceContract]`	Помечает метод как контракт, реализованный службой WCF

Потребители атрибутов

Как нетрудно догадаться, в составе .NET Framework 4.6 SDK поставляется многочисленные утилиты, которые действительно ищут разнообразные атрибуты. Сам компилятор C# (`csc.exe`) запрограммирован на обнаружение различных атрибутов при проведении компиляции. Например, встретив атрибут `[CLSCompilant]`, компилятор автоматически проверяет помеченный им элемент и удостоверяется в том, что в нем открыт доступ только к конструкциям, совместимым с CLS. Еще один пример: если компилятор обнаруживает элемент с атрибутом `[Obsolete]`, то он отображает в окне Error List (Список ошибок) среды Visual Studio сообщение с предупреждением.

В дополнение к инструментам разработки многие методы в библиотеках базовых классов .NET изначально запрограммированы на распознавание определенных атрибутов посредством рефлексии. Например, если нужно сохранить информацию о состоянии объекта в файл, то все, что потребуется сделать — аннотировать класс или структуру атрибутом `[Serializable]`. Если метод `Serialize()` класса `BinaryFormatter` встречает этот атрибут, то объект автоматически сохраняется в файле в компактном двоичном формате.

Наконец, можно строить приложения, способные распознавать специальные атрибуты, а также любые атрибуты из библиотек базовых классов .NET. По сути, тем самым создается набор "ключевых слов", которые понимает специфичное множество сборок.

Применение атрибутов в C#

Чтобы продемонстрировать процесс применения атрибутов в C#, создадим новый проект консольного приложения по имени `ApplyingAttributes`. Предположим, что необходимо построить класс под названием `Motorcycle` (мотоцикл), который может сохраняться в двоичном формате. Для этого нужно просто применить к определению класса атрибут `[Serializable]`. Если какое-то поле не должно сохраняться, то к нему можно применить атрибут `[NonSerialized]`.

```
// Этот класс может быть сохранен на диске.
[Serializable]
public class Motorcycle
{
  // Однако это поле сохраняться не будет.
  [NonSerialized]
  float weightOfCurrentPassengers;

  // Эти поля остаются сериализируемыми.
  bool hasRadioSystem;
  bool hasHeadSet;
  bool hasSissyBar;
}
```

На заметку! Атрибут применяется только к элементу, следующему непосредственно после него. Например, единственным несериализируемым полем в классе `Motorcycle` является `weightOfCurrentPassengers`. Остальные поля будут сериализируемыми, т.к. сам класс аннотирован атрибутом `[Serializable]`.

В настоящий момент пусть вас не заботит действительный процесс сериализации объектов (он подробно рассматривается в главе 20). Просто знайте, что для применения атрибута его имя должно быть помещено в квадратные скобки.

После компиляции этого класса дополнительные метаданные можно просматривать с помощью утилиты `ildasm.exe`. Обратите внимание, что эти атрибуты записаны с использованием маркера `serializable` (взгляните на значок в форме треугольника внутри класса `Motorcycle`) и маркера `notserialized` (в поле `weightOfCurrentPassengers`), как показано на рис. 15.3.

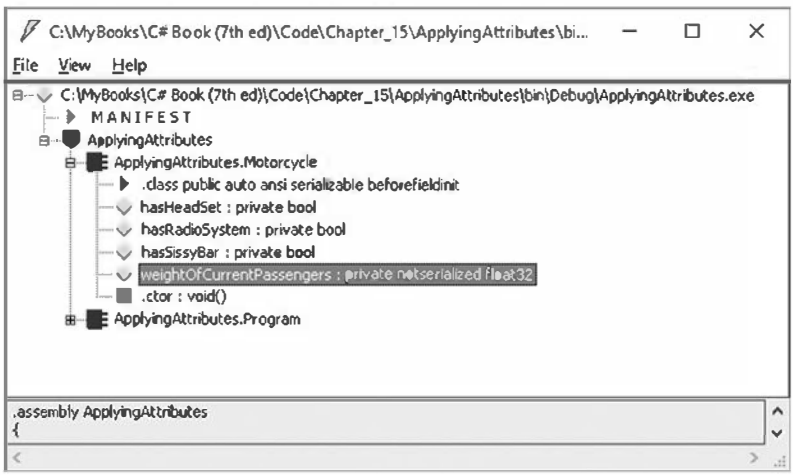


Рис. 15.3. Просмотр атрибутов в `ildasm.exe`

Как нетрудно догадаться, к одиночному элементу можно применять множество атрибутов. Пусть имеется унаследованный тип класса C# (`HorseAndBuggy`), который был помечен как сериализируемый, но для текущей разработки он считается устаревшим. Вместо того чтобы удалять определение этого класса из кодовой базы (с риском нарушения работы существующего программного обеспечения), его можно пометить атрибутом `[Obsolete]`. Для применения множества атрибутов к одному элементу просто используйте список с разделителями-запятыми:

```
[Serializable, Obsolete("Use another vehicle!")]
public class HorseAndBuggy
{
  // ...
}
```

В качестве альтернативы применить множество атрибутов к единственному элементу можно также, указывая их друг за другом (конечный результат будет идентичным):

```
[Serializable]
[Obsolete("Use another vehicle!")]
public class HorseAndBuggy
{
  // ...
}
```

Сокращенная система обозначения атрибутов C#

Заглянув в документацию .NET Framework 4.6 SDK, вы можете заметить, что действительным именем класса, представляющего атрибут `[Obsolete]`, является `ObsoleteAttribute`, а не просто `Obsolete`. По соглашению имена всех атрибутов .NET (включая специальные атрибуты, которые создаете вы сами) снабжаются суффиксом `Attribute`. Тем не менее, чтобы упростить процесс применения атрибутов, язык C# не требует обязательного ввода суффикса `Attribute`. Учитывая это, показанная ниже версия типа `HorseAndBuggy` идентична предыдущей версии (но влечет за собой более объемный клавиатурный набор):

```
[SerializableAttribute]
[ObsoleteAttribute("Use another vehicle!")]
public class HorseAndBuggy
{
  // ...
}
```

Имейте в виду, что такая сокращенная система обозначения для атрибутов предлагается только в C#. Не все языки .NET ее поддерживают.

Указание параметров конструктора для атрибутов

Обратите внимание, что атрибут `[Obsolete]` может принимать то, что выглядит как параметр конструктора. Если вы просмотрите формальное определение атрибута `[Obsolete]`, щелкнув на нем правой кнопкой мыши в окне кода и выбрав в контекстном меню пункт Go To Definition (Перейти к определению), то обнаружите, что этот класс действительно предоставляет конструктор, принимающий `System.String`:

```
public sealed class ObsoleteAttribute : Attribute
{
  public ObsoleteAttribute(string message, bool error);
  public ObsoleteAttribute(string message);
  public ObsoleteAttribute();
  public bool IsError { get; }
  public string Message { get; }
}
```

Важно понимать, что когда вы предоставляете атрибуту параметров конструктора, этот атрибут *не* размещается в памяти до тех пор, пока к параметрам не будет применена рефлексия со стороны другого типа или внешнего инструмента. Строковые данные, определенные на уровне атрибутов, просто сохраняются внутри сборки в виде блока метаданных.

Атрибут [Obsolete] в действии

Теперь, когда класс HorseAndBuggy помечен как устаревший, попытка выделения памяти под его экземпляр, как показано ниже:

```
static void Main(string[] args)
{
  HorseAndBuggy mule = new HorseAndBuggy();
}
```

приводит к тому, что указанные строковые данные извлекаются и отображаются в окне Error List среды Visual Studio, а также в проблемной строке кода при наведении курсора мыши на устаревший тип (рис. 15.4).

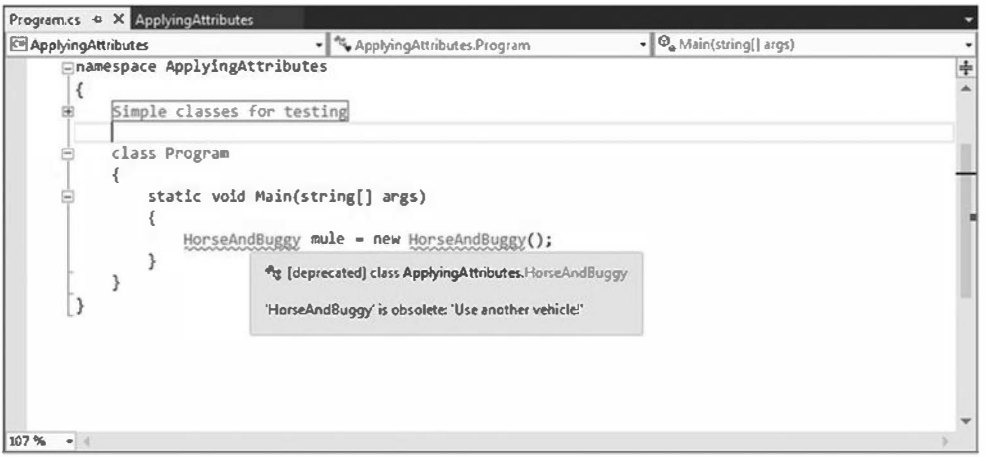


Рис. 15.4. Атрибуты в действии

В этом случае "другой частью программного обеспечения", которая производит рефлексию атрибута [Obsolete], является компилятор C#. По идее к этому моменту вы должны понимать следующие ключевые моменты, касающиеся атрибутов .NET:

- атрибуты представляют собой классы, производные от System.Attribute;
- атрибуты дают в результате встроенные метаданные;
- атрибуты в основном бесполезны до тех пор, пока другой агент не проведет в их отношении рефлексию;
- атрибуты в языке C# применяются с использованием квадратных скобок.

А теперь давайте посмотрим, как реализовывать собственные специальные атрибуты и создавать специальное программное обеспечение, которое выполняет рефлексию по встроенным метаданным.

Исходный код. Проект ApplyingAttributes доступен в подкаталоге Chapter_15.

Построение специальных атрибутов

Первый шаг при построении специального атрибута предусматривает создание нового класса, производного от System.Attribute. Не отклоняясь от автомобильной темы, повсеместно встречающейся в книге, создадим новый проект типа Class Library (Библиотека классов) на C# под названием AttributedCarLibrary.

В этой сборке будет определено несколько классов для представления транспортных средств, каждый из которых описан с использованием специального атрибута по имени VehicleDescriptionAttribute:

```
// Специальный атрибут.
public sealed class VehicleDescriptionAttribute : System.Attribute
{
  public string Description { get; set; }

  public VehicleDescriptionAttribute(string vehicalDescription)
  {
    Description = vehicalDescription;
  }
  public VehicleDescriptionAttribute(){ }
}
```

Как видите, класс VehicleDescriptionAttribute поддерживает фрагмент строковых данных, которым можно манипулировать с помощью автоматического свойства (Description). Помимо того факта, что этот класс является производным от System.Attribute, ничего примечательного в его определении нет.

На заметку! По причинам, связанным с безопасностью, установившейся практикой в .NET считается проектирование всех специальных атрибутов как запечатанных. На самом деле среда Visual Studio предлагает фрагмент кода под названием Attribute, который позволяет сгенерировать в окне редактора кода новый класс, производный от System.Attribute. О применении фрагментов кода подробно рассказывалось в главе 2; как вы, скорее всего, помните, для раскрытия любого фрагмента кода необходимо набрать его имя и два раза нажать клавишу <Tab>.

Применение специальных атрибутов

С учетом того, что класс VehicleDescriptionAttribute является производным от System.Attribute, теперь можно аннотировать транспортные средства. В целях тестирования добавим в новую библиотеку классов следующие определения:

```
// Назначить описание с помощью "именованного свойства".
[Serializable]
[VehicleDescription(Description = "My rocking Harley")]
public class Motorcycle
{
}

[Serializable]
[Obsolete ("Use another vehicle!")]
[VehicleDescription("The old gray mare, she ain't what she used to be...")]
public class HorseAndBuggy
{
}

[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
}
```

Синтаксис именованных свойств

Обратите внимание, что описание классу Motorcycle назначается с использованием нового фрагмента синтаксиса, связанного с атрибутами, который называется *именованным свойством*. В конструкторе первого атрибута [VehicleDescription] лежащие

в основе строковые данные устанавливаются с применением свойства `Description`. Когда внешний агент выполняет рефлексию для этого атрибута, свойству `Description` будет передано указанное значение (синтаксис именованных свойств разрешен, только если атрибут предоставляет поддерживающее запись свойство .NET).

В противоположность этому для типов `HorseAndBuggy` и `Winnebago` синтаксис именованных свойств не используется, а строковые данные просто передаются посредством специального конструктора. После компиляции сборки `AttributedCarLibrary` с помощью утилиты `ildasm.exe` можно просмотреть добавленные описания метаданных. Например, на рис. 15.5 показано встроенное описание класса `Winnebago`, в частности, данные внутри элемента `beforefieldinit`.

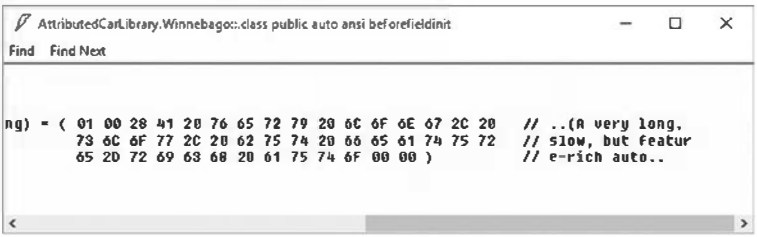


Рис. 15.5. Встроенные данные описания транспортного средства

Ограничение использования атрибутов

По умолчанию специальные атрибуты могут быть применены практически к любому аспекту кода (методам, классам, свойствам и т.д.). Таким образом, если бы это имело смысл, то `VehicleDescription` можно было бы использовать для уточнения методов, свойств или полей (помимо прочего):

```
[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
  [VehicleDescription("My rocking CD player")]
  public void PlayMusic(bool On)
  {
    ...
  }
}
```

В одних случаях такое поведение является именно тем, что необходимо, но в других может требоваться специальный атрибут, который допускается применять только к избранным элементам кода. Чтобы ограничить область действия специального атрибута, понадобится добавить к его определению атрибут `[AttributeUsage]`. Атрибут `[AttributeUsage]` позволяет предоставлять любую комбинацию значений (посредством операции "ИЛИ") из перечисления `AttributeTargets`:

```
// Это перечисление определяет возможные целевые элементы для атрибута.
public enum AttributeTargets
{
  All, Assembly, Class, Constructor,
  Delegate, Enum, Event, Field, GenericParameter,
  Interface, Method, Module, Parameter,
  Property, ReturnValue, Struct
}
```

Кроме того, атрибут `[AttributeUsage]` также дает возможность дополнительно устанавливать именованное свойство (`AllowMultiple`), которое указывает, может ли ат-

рибут применяться к тому же самому элементу более одного раза (стандартное значение этого свойства — false). Вдобавок [AttributeUsage] позволяет указывать, должен ли атрибут наследоваться производными классами, с использованием именованного свойства Inherited (со стандартным значением true).

Модифицируем определение VehicleDescriptionAttribute для указания на то, что атрибут [VehicleDescription] может применяться только к классу или структуре:

```
// На этот раз мы используем атрибут AttributeUsage
// для аннотирования специального атрибута.
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
                Inherited = false)]
public sealed class VehicleDescriptionAttribute : System.Attribute
{
  ...
}
```

Теперь если разработчик попытается применить атрибут [VehicleDescription] не к классу или структуре, то компилятор сообщит об ошибке.

Атрибуты уровня сборки

Атрибуты можно также применять ко всем типам внутри отдельной сборки, используя дескриптор [assembly:]. Например, предположим, что необходимо обеспечить совместимость с CLS для всех открытых членов во всех открытых типах, определенных внутри сборки.

На заметку! В главе 1 упоминалось о роли сборок, совместимых с CLS. Вспомните, что совместимая с CLS сборка может быть задействована во всех языках программирования .NET. Если в открытых типах создаются открытые члены, которые предоставляют доступ к несовместимым с CLS конструкциям программирования (таким как данные без знака или параметры типа указателей), то другие языки .NET могут оказаться не в состоянии работать с ними. Следовательно, при построении библиотек кода C#, которые должны применяться во множестве языков .NET, проверка на совместимость с CLS является обязательной.

Для этого нужно просто добавить в начало файла исходного кода C# показанный ниже атрибут уровня сборки. Имейте в виду, что все атрибуты уровня сборки или модуля должны быть указаны за пределами области действия любого пространства имен! При добавлении в проект атрибутов уровня сборки или модуля рекомендуется придерживаться следующей схемы для файла кода:

```
// Первыми перечислить операторы using.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

// Теперь перечислить атрибуты уровня сборки или модуля.
// Обеспечить совместимость с CLS для всех открытых типов в этой сборке.
[assembly: CLSCompliant(true)]

// Далее может следовать ваше пространство (пространства) имен и типы.
namespace AttributedCarLibrary
{
  // Типы...
}
```

Если теперь добавить фрагмент кода, выходящий за рамки спецификации CLS (вроде открытого элемента данных без знака):

```
// Тип ulong не соответствует спецификации CLS.
public class Winnebago
{
  public ulong notCompliant;
}
```

то компилятор выдаст предупреждение.

Файл `AssemblyInfo.cs`, генерируемый Visual Studio

По умолчанию проекты Visual Studio получают файл по имени `AssemblyInfo.cs`, который можно просмотреть, раскрыв в окне Solution Explorer узел Properties (Свойства), как показано на рис. 15.6.

В этот файл удобно помещать атрибуты, подлежащие применению на уровне сборки. Во время исследования сборок .NET в главе 14 рассказывалось о том, что в манифесте содержатся метаданные уровня сборки, большая часть которых берется из атрибутов уровня сборки, описанных в табл. 15.4.

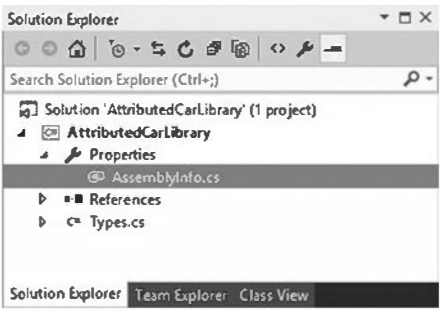


Рис. 15.6. Файл `AssemblyInfo.cs`

Таблица 15.4. Избранные атрибуты уровня сборки

Атрибут	Описание
`[AssemblyCompany]`	Хранит общую информацию о компании
`[AssemblyCopyright]`	Хранит любую информацию, касающуюся авторских прав на данный продукт или сборку
`[AssemblyCulture]`	Предоставляет информацию о том, какие культуры или языки поддерживает сборка
`[AssemblyDescription]`	Хранит дружественное описание продукта или модулей, из которых состоит сборка
`[AssemblyKeyFile]`	Указывает имя файла, в котором содержится пара ключей, используемых для подписания сборки (т.е. создания строгого имени)
`[AssemblyProduct]`	Предоставляет информацию о продукте
`[AssemblyTrademark]`	Предоставляет информацию о торговой марке
`[AssemblyVersion]`	Указывает информацию о версии сборки в формате `<старший_номер.младший_номер.номер_сборки.номер_редакции>`

Исходный код. Проект `AttributedCarLibrary` доступен в подкаталоге `Chapter_15`.

Рефлексия атрибутов с использованием раннего связывания

Вспомните, что атрибуты остаются бесполезными до тех пор, пока к их значениям не будет применена рефлексия в другой части программного обеспечения. После обнаружения атрибута эта другая часть кода может предпринять необходимый образ действий. Подобно любому приложению "другая часть программного обеспечения" может обнаруживать присутствие специального атрибута с использованием либо раннего, либо позднего связывания. Для применения раннего связывания определение интересующего атрибута (в данном случае `VehicleDescriptionAttribute`) должно находиться в клиентском приложении на этапе компиляции. Учитывая то, что специальный атрибут определен в сборке `AttributedCarLibrary` как открытый класс, раннее связывание будет наилучшим выбором.

Чтобы проиллюстрировать процесс рефлексии специальных атрибутов, создадим новый проект консольного приложения по имени `VehicleDescriptionAttributeReader`, добавим в него ссылку на сборку `AttributedCarLibrary` и поместим в первоначальный файл `*.cs` следующий код:

```
// Выполнение рефлексии атрибутов, используя раннее связывание.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using AttributedCarLibrary;

namespace VehicleDescriptionAttributeReader
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
      ReflectOnAttributesUsingEarlyBinding();
      Console.ReadLine();
    }

    private static void ReflectOnAttributesUsingEarlyBinding()
    {
      // Получить объект Type, представляющий тип Winnebago.
      Type t = typeof(Winnebago);

      // Получить все атрибуты Winnebago.
      object[] customAtts = t.GetCustomAttributes(false);

      // Вывести описание.
      foreach (VehicleDescriptionAttribute v in customAtts)
        Console.WriteLine("-> {0}\n", v.Description);
    }
  }
}
```

Метод `Type.GetCustomAttributes()` возвращает массив объектов со всеми атрибутами, примененными к члену, который представлен объектом `Type` (булевский параметр управляет тем, должен ли поиск распространяться вверх по цепочке наследования). После получения списка атрибутов осуществляется проход по всем элементам `VehicleDescriptionAttribute` с отображением значения свойства `Description`.

Исходный код. Проект VehicleDescriptionAttributeReader доступен в подкаталоге Chapter_15.

Рефлексия атрибутов с использованием позднего связывания

В предыдущем примере для вывода описания транспортного средства типа Winnebago применялось ранее связывание. Это было возможно благодаря тому, что тип класса VehicleDescriptionAttribute определен в сборке AttributedCarLibrary как открытый член. Для рефлексии атрибутов также допускается использовать динамическую загрузку и позднее связывание.

Создадим новый проект консольного приложения по имени VehicleDescriptionAttributeReaderLateBinding и скопируем сборку AttributedCarLibrary.dll в каталог bin\Debug проекта. Теперь модифицируем класс Program, как показано ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;

namespace VehicleDescriptionAttributeReaderLateBinding
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
      ReflectAttributesUsingLateBinding();
      Console.ReadLine();
    }

    private static void ReflectAttributesUsingLateBinding()
    {
      try
      {
        // Загрузить локальную копию сборки AttributedCarLibrary.
        Assembly asm = Assembly.Load("AttributedCarLibrary");

        // Получить информацию о типе VehicleDescriptionAttribute.
        Type vehicleDesc =
          asm.GetType("AttributedCarLibrary.VehicleDescriptionAttribute");

        // Получить информацию о типе свойства Description.
        PropertyInfo propDesc = vehicleDesc.GetProperty("Description");

        // Получить все типы в сборке.
        Type[] types = asm.GetTypes();

        // Пройти по всем типам и получить любые
        // атрибуты VehicleDescriptionAttribute.
        foreach (Type t in types)
        {
          object[] objs = t.GetCustomAttributes(vehicleDesc, false);

          // Пройти по каждому VehicleDescriptionAttribute и вывести
          // описание, используя позднее связывание.
```

```
            foreach (object o in objs)
            {
              Console.WriteLine("-> {0}: {1}\n",
              t.Name, propDesc.GetValue(o, null));
            }
          }
        }
        catch (Exception ex)
        {
          Console.WriteLine(ex.Message);
        }
      }
    }
  }
```

Если вы прорабатывали примеры, приведенные ранее в главе, то этот код должен быть более или менее понятен. Единственный интересный момент здесь связан с применением метода `PropertyInfo.GetValue()`, который служит для активизации средства доступа к свойству. Вот как выглядит вывод, полученный в результате выполнения текущего примера:

```
***** Value of VehicleDescriptionAttribute *****
-> Motorcycle: My rocking Harley
-> HorseAndBuggy: The old gray mare, she ain't what she used to be...
-> Winnebago: A very long, slow, but feature-rich auto
```

Исходный код. Проект `VehicleDescriptionAttributeReaderLateBinding` доступен в подкаталоге `Chapter_15`.

Практическое использование рефлексии, позднего связывания и специальных атрибутов

Хотя вы видели многочисленные примеры применения этих приемов, вас по-прежнему может интересовать, когда использовать рефлексию, динамическое связывание и специальные атрибуты в своих программах. Действительно, данные темы могут показаться как относящиеся в большей степени к академической стороне программирования (в зависимости от вашей точки зрения это может быть как отрицательным, так и положительным моментом). Для содействия в отображении этих тем на реальные ситуации необходим более серьезный пример. Предположим, что вы работаете в составе команды программистов, которая занимается построением приложения со следующим требованием:

- продукт должен быть расширяемым за счет использования дополнительных сторонних инструментов.

Что понимается под *расширяемостью*? Возьмем IDE-среду Visual Studio. Когда это приложение разрабатывалось, в его кодовую базу были вставлены многочисленные "зацепки", чтобы позволить другим производителям программного обеспечения "привязывать" (или подключать) специальные модули к IDE-среде. Очевидно, что у разработчиков Visual Studio отсутствовал какой-либо способ установки ссылок на внешние сборки .NET, которые пока еще не были созданы (таким образом, раннее связывание недоступно), тогда как они обеспечили наличие в приложении необходимых зацепок?

Ниже представлен один из возможных способов решения этой задачи.

- Во-первых, расширяемое приложение должно предоставлять некоторый механизм ввода, позволяющий пользователю указать модуль для подключения (наподобие диалогового окна или флага командной строки). Это требует динамической загрузки.
- Во-вторых, расширяемое приложение должно иметь возможность выяснять, поддерживает ли модуль корректную функциональность (такую как набор обязательных интерфейсов), необходимую для его подключения к среде. Это требует рефлексии.
- В-третьих, расширяемое приложение должно получать ссылку на требуемую инфраструктуру (вроде набора интерфейсных типов) и вызывать члены для запуска лежащей в основе функциональности. Это может требовать позднего связывания.

Попросту говоря, если расширяемое приложение изначально было запрограммировано для запрашивания специфических интерфейсов, то оно в состоянии выяснять во время выполнения, может ли активизироваться интересующий тип. После успешного прохождения такой проверки тип может поддерживать дополнительные интерфейсы, которые формируют полиморфную фабрику его функциональности. Именно этот подход был принят командой разработчиков Visual Studio, и вопреки тому, что вы могли подумать, в нем нет ничего сложного!

Построение расширяемого приложения

В последующих разделах будет рассмотрен полноценный пример создания расширяемого приложения Windows Forms, которое может быть дополнено функциональностью внешних сборок. Если вы не имеете опыта построения графических пользовательских интерфейсов с помощью Windows Forms, то можете просто загрузить предоставленный код решения и работать с ним.

На заметку! Инфраструктура Windows Forms была первоначальным API-интерфейсом для разработки настольных приложений в .NET. Однако после выхода .NET 3.0 предпочтительной инфраструктурой для построения графических пользовательских интерфейсов стал API-интерфейс Windows Presentation Foundation (WPF). Несмотря на это, Windows Forms будет применяться в нескольких примерах клиентов с графическими пользовательскими интерфейсами, т.к. связанный с ним код более понятен, чем соответствующий код WPF.

Если вы не знакомы с процессом построения приложений Windows Forms, то просто откройте загруженный код примера и следуйте дальнейшим указаниям в главе. Расширяемое приложение образовано из следующих сборок.

- `CommonSnappableTypes.dll`. Эта сборка содержит определения типов, которые будут использоваться каждым объектом оснастки, и на нее будет напрямую ссылаться приложение Windows Forms.
- `CSharpSnapIn.dll`. Оснастка, написанная на C#, в которой задействованы типы из сборки `CommonSnappableTypes.dll`.
- `VbSnapIn.dll`. Оснастка, написанная на Visual Basic, в которой применяются типы из сборки `CommonSnappableTypes.dll`.
- `MyExtendableApp.exe`. Исполняемое приложение Windows Forms, которое может быть расширено функциональностью каждой оснастки.

Стоит еще раз повторить, что в этом приложении будут использоваться динамическая загрузка, рефлексия и позднее связывание для динамического получения функциональности сборок, о которых заранее ничего не известно.

Построение сборки CommonSnappableTypes.dll

Первым делом необходимо построить сборку, содержащую типы, которые заданная оснастка должна задействовать, чтобы быть подключенной к расширяемому приложению Windows Forms. Создадим проект библиотеки классов по имени CommonSnappableTypes и определим в нем следующие два типа:

```
namespace CommonSnappableTypes
{
  public interface IAppFunctionality
  {
    void DoIt();
  }

  [AttributeUsage(AttributeTargets.Class)]
  public sealed class CompanyInfoAttribute : System.Attribute
  {
    public string CompanyName { get; set; }
    public string CompanyUrl { get; set; }
  }
}
```

Интерфейс IAppFunctionality предоставляет полиморфный интерфейс для всех оснасток, которые могут потребляться расширяемым приложением Windows Forms. Учитывая, что рассматриваемый пример является полностью демонстрационным, в интерфейсе определен единственный метод по имени DoIt(). Более реалистичный интерфейс (или набор интерфейсов) мог бы позволять объекту генерировать код сценария, визуализировать изображение в панели инструментов приложения либо интегрироваться в главное меню размещающего приложения.

Тип CompanyInfoAttribute — это специальный атрибут, который может применяться к любому классу, желающему подключиться к контейнеру. Как несложно заметить по определению класса, [CompanyInfo] позволяет разработчику оснастки указывать общие сведения о месте происхождения компонента.

Построение оснастки на C#

Теперь займемся созданием типа, реализующего интерфейс IAppFunctionality. Опять-таки, чтобы целиком сосредоточиться на общем проектном решении расширяемого приложения, этот тип будет простым. Пусть в новом проекте библиотеки классов C# по имени CSharpSnapIn определен класс CSharpModule. Учитывая, что этот класс должен использовать типы, определенные в сборке CommonSnappableTypes, не забудьте добавить ссылку на нее (а также на сборку System.Windows.Forms.dll для отображения сообщений). Ниже приведен необходимый код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using CommonSnappableTypes;
using System.Windows.Forms;

namespace CSharpSnapIn
{
```

```
  [CompanyInfo(CompanyName = "FooBar",
    CompanyUrl = "www.FooBar.com")]
  public class CSharpModule : IAppFunctionality
  {
    void IAppFunctionality.DoIt()
    {
      MessageBox.Show("You have just used the C# snap-in!");
    }
  }
}
```

Обратите внимание на явную реализацию интерфейса `IAppFunctionality` (см. главу 9). Это не является обязательным; идея в том, что единственной частью системы, которая нуждается в прямом взаимодействии с этим интерфейсным типом, является размещающее Windows-приложение. Благодаря явной реализации этого интерфейса метод `DoIt()` не будет доступен непосредственно из типа `CSharpModule`.

Построение оснастки на Visual Basic

Для эмуляции стороннего производителя, который предпочитает применять язык Visual Basic, а не C#, создадим новый проект библиотеки классов на Visual Basic (`VbSnapIn`), ссылающийся на те же самые внешние сборки, что и предыдущий проект `CSharpSnapIn`.

На заметку! По умолчанию вкладка References (Ссылки) для проекта Visual Basic в окне Solution Explorer отображаться не будет, поэтому для добавления ссылок в такой проект придется использовать пункт меню Add Reference⇨Project (Добавить ссылку⇨Проект) в Visual Studio.

Код Visual Basic также прост:

```
Imports System.Windows.Forms
Imports CommonSnappableTypes

<CompanyInfo(CompanyName:="Chucky's Software",
             CompanyUrl:="www.ChuckySoft.com")>
Public Class VbSnapIn
  Implements IAppFunctionality

  Public Sub DoIt() Implements CommonSnappableTypes.IAppFunctionality.DoIt
    MessageBox.Show("You have just used the VB snap in!")
  End Sub
End Class
```

Как видите, применение атрибутов в синтаксисе Visual Basic требует указания угловых скобок (<>), а не квадратных ([]). Кроме того, для реализации интерфейсных типов заданным классом или структурой используется ключевое слово `Implements`.

Построение расширяемого приложения Windows Forms

Финальным шагом является создание нового приложения Windows Forms (`MyExtendableApp`) на C#, которое позволяет пользователю выбирать оснастку с помощью стандартного диалогового окна открытия файлов Windows. Если ранее вам не приходилось создавать приложения Windows Forms, то можете просто открыть проект, поставляемый в рамках исходного кода примеров для этой книги. Тем не менее, если вы все же хотите строить графический пользовательский интерфейс самостоятельно, то начните с выбора в диалоговом окне New Project (Новый проект) среды Visual Studio шаблона проекта Windows Forms Application (Приложение Windows Forms), как показано на рис. 15.7.

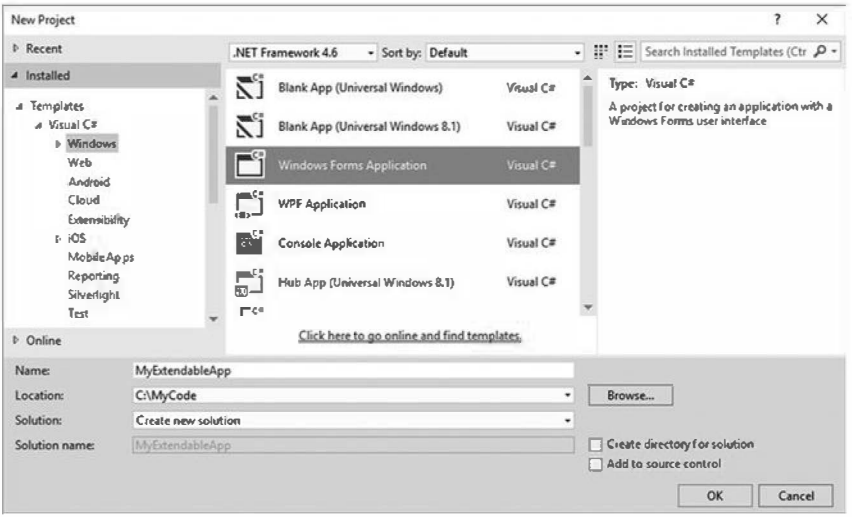


Рис. 15.7. Создание нового проекта Windows Forms в Visual Studio

---

На заметку! Обзор API-интерфейса Windows Forms приведен в приложении A, которое доступно для загрузки на веб-сайте издательства.

Теперь добавьте ссылку на сборку `CommonSnappableTypes.dll`, но *не* на библиотеки кода `CSharpSnapIn.dll` и `VbSnapIn.dll`. Также импортируйте пространства имен `System.Reflection` и `CommonSnappableTypes` в основном файле кода формы (который можно открыть, щелкнув правой кнопкой мыши на поверхности визуального конструктора форм и выбрав в контекстном меню пункт View Code (Просмотреть код)). Вспомните, что цель этого приложения заключается в применении позднего связывания и рефлексии для выяснения "подключаемости" независимых двоичных файлов, созданных другими производителями.

Мы не будем здесь исследовать все детали разработки приложений Windows Forms. Однако все же отметим, что графический пользовательский интерфейс состоит из компонента `MenuStrip`, помещенного на поверхность визуального конструктор форм; в этом компоненте определен единственный элемент верхнего меню по имени File (Файл), который предоставляет одно подменю Snap In Module (Подключить модуль). Кроме того, главное окно содержит элемент `ListBox` (с именем `lstLoadedSnapIns`), который будет отображать имена оснасток, загруженных пользователем. Результирующий графический пользовательский интерфейс показан на рис. 15.8.

Код обработки события `Click` для пункта меню File⇨Snap In Module (который можно создать двойным щелчком на этом пункте в визуальном конструкторе форм) отображает диалоговое окно File Open (Открытие файла) и извлекает путь к выбранному файлу. Предполагая, что пользователь не выбрал сборку `CommonSnappableTypes.dll` (т.к. это чистая инфраструктура), данный путь затем передается на обработку вспомогательному методу `LoadExternalModule()`, который реализуется следующим. Этот метод будет возвращать `false`, если не удается найти класс, реализующий интерфейс `IAppFunctionality`.

```
private void snapInModuleToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Позволить пользователю выбрать сборку для загрузки.
  OpenFileDialog dlg = new OpenFileDialog();
```

```
  if (dlg.ShowDialog() == DialogResult.OK)
  {
    if(dlg.FileName.Contains("CommonSnappableTypes"))
      MessageBox.Show("CommonSnappableTypes has no snap-ins!");
                    // CommonSnappableTypes не содержит оснасток.
    else if(!LoadExternalModule(dlg.FileName))
      MessageBox.Show("Nothing implements IAppFunctionality!");
             // Не удается обнаружить класс, реализующий IAppFunctionality.
  }
}
```

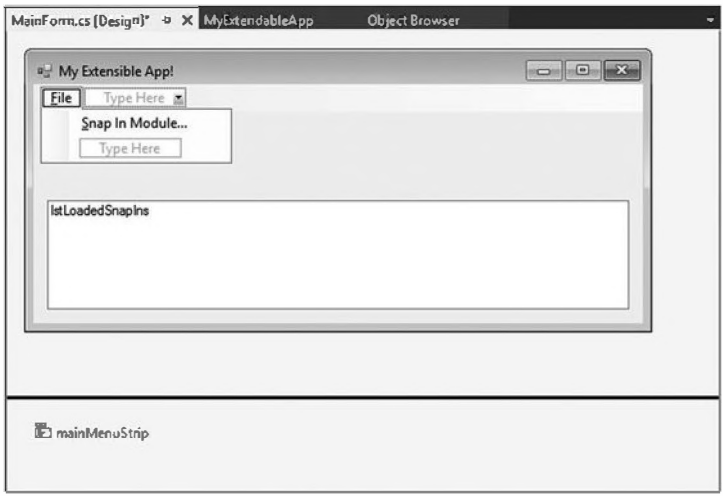


Рис. 15.8. Графический пользовательский интерфейс для приложения `MyExtendableApp`

Метод `LoadExternalModule()` выполняет следующие действия:

- **динамически загружает в память выбранную сборку;**
- **определяет, содержит ли сборка типы, реализующие интерфейс `IAppFunctionality`;**
- **создает экземпляр типа, используя позднее связывание.**

Если тип, реализующий `IAppFunctionality`, найден, то вызывается метод `DoIt()` и полностью заданное имя типа добавляется в `ListBox` (обратите внимание, что цикл `foreach` будет проходить по всем типам в сборке, чтобы учесть возможность наличия в одной сборке нескольких оснасток):

```
private bool LoadExternalModule(string path)
{
  bool foundSnapIn = false;
  Assembly theSnapInAsm = null;
  try
  {
    // Динамически загрузить выбранную сборку.
    theSnapInAsm = Assembly.LoadFrom(path);
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
    return foundSnapIn;
  }
```

```
  // Получить все совместимые с IAppFunctionality классы в сборке.
  var theClassTypes = from t in theSnapInAsm.GetTypes()
                      where t.IsClass &&
                        (t.GetInterface("IAppFunctionality") != null)
                      select t;
  // Создать объект и вызвать метод DoIt().
  foreach (Type t in theClassTypes)
  {
    foundSnapIn = true;
    // Использовать позднее связывание для создания экземпляра типа.
    IAppFunctionality itfApp =
      (IAppFunctionality)theSnapInAsm.CreateInstance(t.FullName, true);
    itfApp.DoIt();
    lstLoadedSnapIns.Items.Add(t.FullName);
  }
  return foundSnapIn;
}
```

На этом этапе приложение уже можно запускать. В случае выбора сборки `CSharpSnapIn.dll` **или** `VbSnapIn.dll` **должно отобразиться корректное сообщение. Последней задачей является отображение метаданных, предоставляемых атрибутом** `[CompanyInfo]`. **Для этого модифицируем метод** `LoadExternalModule()`, **добавив в конец тела цикла** `foreach` **вызов нового вспомогательного метода по имени** `DisplayCompanyData()`, **который принимает параметр** `System.Type`:

```
private bool LoadExternalModule(string path)
{
  ...
  foreach (Type t in theClassTypes)
  {
    ...
    // Отобразить информацию о компании.
    DisplayCompanyData(t);
  }
  return foundSnapIn;
}
```

Выполним рефлексию атрибута `[CompanyInfo]` **с применением этого входного типа:**

```
private void DisplayCompanyData(Type t)
{
  // Получить данные [CompanyInfo].
  var compInfo = from ci in t.GetCustomAttributes(false) where
              (ci.GetType() == typeof(CompanyInfoAttribute))
              select ci;
  // Отобразить данные.
  foreach (CompanyInfoAttribute c in compInfo)
  {
    MessageBox.Show(c.CompanyUrl,
      string.Format("More info about {0} can be found at", c.CompanyName));
  }
}
```

На рис. 15.9 показан результат тестового запуска приложения.

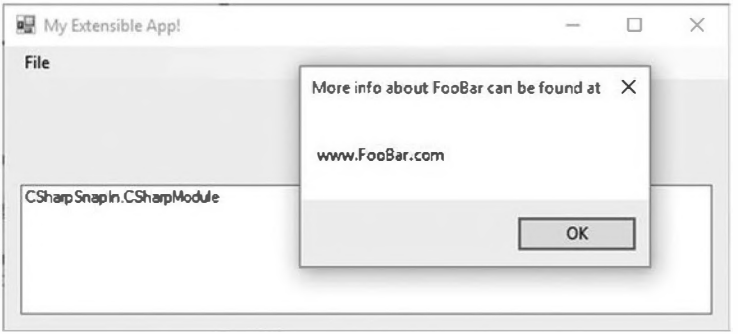


Рис. 15.9. Подключение внешних сборок

Итак, создание примера расширяемого приложения завершено. Вы смогли увидеть, что представленные в главе приемы могут оказаться весьма полезными в реальности, причем не только для разработчиков инструментов.

Исходный код. Проекты `CommonSnappableTypes`, `CSharpSnapIn`, `VbSnapIn` и `MyExtendableApp` доступны в подкаталоге `ExtendableApp` внутри каталога `Chapter_15`.

Резюме

Рефлексия является интересным аспектом надежной объектно-ориентированной среды. В мире .NET службы рефлексии вращаются вокруг класса `System.Type` и пространства имен `System.Reflection`. Вы видели, что рефлексия — это процесс помещения типа под "увеличительное стекло" во время выполнения с целью выяснения его характеристик и возможностей.

Позднее связывание представляет собой процесс создания экземпляра типа и обращения к его членам без предварительного знания имен членов типа. Позднее связывание часто является прямым результатом *динамической загрузки*, которая позволяет программно загружать сборку .NET в память. С помощью примера расширяемого приложения было продемонстрировано, что это мощный прием, используемый разработчиками инструментов, а также их потребителями.

Кроме того, в главе была исследована роль программирования на основе атрибутов. Снабжение типов атрибутами приводит к дополнению метаданных лежащей в основе сборки.

ГЛАВА 16

Динамические типы и среда DLR

В версии .NET 4.0 язык C# получил новое ключевое слово `dynamic`. Это ключевое слово позволяет внедрять в строго типизированный мир безопасности к типам, точек с запятой и фигурных скобок поведение, характерное для сценариев. Используя такую слабую типизацию, можно значительно упростить некоторые сложные задачи кодирования и также получить возможность взаимодействия с несколькими динамическими языками (вроде IronRuby и IronPython), которые поддерживают .NET.

В этой главе вы узнаете о ключевом слове `dynamic` и о том, как слабо типизированные вызовы отображаются на корректные объекты в памяти с применением *исполняющей среде динамического языка* (Dynamic Language Runtime — DLR). После освоения служб, предоставляемых средой DLR, вы увидите примеры использования динамических типов для облегчения вызова методов с поздним связыванием (через службы рефлексии) и простого взаимодействия с унаследованными библиотеками COM.

На заметку! Не путайте ключевое слово `dynamic` языка C# с концепцией *динамической сборки* (объясняемой в главе 18). Хотя ключевое слово `dynamic` может применяться при построении динамической сборки, все же это две совершенно независимые концепции.

Роль ключевого слова `dynamic` языка C#

В главе 3 вы ознакомились с ключевым словом `var`, которое позволяет объявлять локальные переменные таким способом, что их действительные типы данных определяются на основе начального присваивания во время компиляции (вспомните, что это называется *неявной типизацией*). После того как начальное присваивание выполнено, вы имеете строго типизированную переменную, и любая попытка присвоить ей несовместимое значение приведет к ошибке на этапе компиляции.

Чтобы приступить к исследованию ключевого слова `dynamic` языка C#, создадим новый проект консольного приложения по имени `DynamicKeyword`. Далее добавим в класс `Program` следующий метод и удостоверимся, что финальный оператор кода действительно генерирует ошибку во время компиляции, если убрать символы комментария:

```
static void ImplicitlyTypedVariable()
{
  // Переменная a имеет тип List<int>.
  var a = new List<int>();
  a.Add(90);
  // Этот оператор приведет к ошибке на этапе компиляции!
  // a = "Hello";
}
```

Использование неявной типизации только из-за того, что она возможна, некоторые считают плохим стилем (если известно, что необходима переменная типа `List<int>`, то так и следует ее объявлять). Однако, как было показано в главе 12, неявная типизация удобна в сочетании с LINQ, поскольку многие запросы LINQ возвращают перечисления анонимных классов (посредством проекций), которые напрямую объявить в коде C# невозможно. Тем не менее, даже в таких случаях неявно типизированная переменная на самом деле будет строго типизированной.

В качестве связанного замечания: в главе 6 упоминалось, что `System.Object` является изначальным родительским классом внутри .NET Framework и может представлять все, что угодно. Опять-таки, объявление переменной типа `object` в результате дает строго типизированный фрагмент данных, но то, на что указывает эта переменная в памяти, может отличаться в зависимости от присваиваемой ссылки. Чтобы получить доступ к членам объекта, на который указывает ссылка в памяти, понадобится выполнить явное приведение.

Предположим, что есть простой класс по имени `Person`, в котором определены два автоматических свойства (`FirstName` и `LastName`), инкапсулирующие данные `string`. Теперь взгляните на следующий код:

```
static void UseObjectVarible()
{
  // Пусть имеется класс по имени Person.
  object o = new Person() { FirstName = "Mike", LastName = "Larson" };
  // Для получения доступа к свойствам Person переменную o
  // потребуется привести к Person.
  Console.WriteLine("Person's first name is {0}", ((Person)o).FirstName);
}
```

С выходом версии .NET 4.0 в язык C# было введено ключевое слово `dynamic`. С высокоуровневой точки значения ключевое слово `dynamic` можно трактовать как специализированную форму типа `System.Object`, в том смысле, что переменной динамического типа данных может быть присвоено любое значение. На первый взгляд, это может привести к серьезной путанице, поскольку теперь получается, что доступны три способа определения данных, внутренний тип которых явно не указан в кодовой базе. Например, следующий метод:

```
static void PrintThreeStrings()
{
  var s1 = "Greetings";
  object s2 = "From";
  dynamic s3 = "Minneapolis";

  Console.WriteLine("s1 is of type: {0}", s1.GetType());
  Console.WriteLine("s2 is of type: {0}", s2.GetType());
  Console.WriteLine("s3 is of type: {0}", s3.GetType());
}
```

в случае вызова в `Main()` приведет к такому выводу:

```
s1 is of type: System.String
s2 is of type: System.String
s3 is of type: System.String
```

Динамическая переменная и переменная, объявленная неявно или через ссылку на `System.Object`, существенно отличаются тем, что динамическая переменная *не является строго типизированной*. Выражаясь по-другому, динамические данные *не типизированы статически*. Для компилятора C# это выглядит так, что элементу данных, объявленному с ключевым словом `dynamic`, можно присваивать любое начальное зна-

чение, и на протяжении периода его существования вместо этого значения может быть присвоено любое новое (возможно, не связанное) значение. Рассмотрим показанный ниже метод и результирующий вывод:

```
static void ChangeDynamicDataType()
{
  // Объявить одиночный динамический элемент данных по имени t.
  dynamic t = "Hello!";
  Console.WriteLine("t is of type: {0}", t.GetType());

  t = false;
  Console.WriteLine("t is of type: {0}", t.GetType());

  t = new List<int>();
  Console.WriteLine("t is of type: {0}", t.GetType());
}
t is of type: System.String
t is of type: System.Boolean
t is of type: System.Collections.Generic.List`1[System.Int32]
```

Имейте в виду, что приведенный выше код успешно бы скомпилировался и дал идентичный результат, если бы переменная `t` была объявлена с типом `System.Object`. Однако, как вскоре будет показано, ключевое слово `dynamic` предлагает много дополнительных возможностей.

Вызов членов на динамически объявленных данных

С учетом того, что динамическая переменная способна принимать идентичность любого типа на лету (подобно переменной типа `System.Object`), у вас может возникнуть вопрос о способе обращения к членам такой переменной (свойствам, методам, индексаторам, событиям и т.п.). С точки зрения синтаксиса отличий нет. Нужно просто применить операцию точки к динамической переменной, указать открытый член и предоставить любые аргументы (если они требуются).

Но (и это очень важное "но") допустимость указываемых членов компилятор проверять не будет! Вспомните, что в отличие от переменной, определенной с типом `System.Object`, динамические данные не являются статически типизированными. Вплоть до времени выполнения не будет известно, поддерживают ли вызываемые динамические данные указанный член, переданы ли корректные параметры, правильно ли записано имя члена, и т.д. Таким образом, хотя это может показаться странным, следующий метод благополучно скомпилируется:

```
static void InvokeMembersOnDynamicData()
{
  dynamic textData1 = "Hello";
  Console.WriteLine(textData1.ToUpper());

  // Здесь можно было бы ожидать ошибки на этапе компиляции!
  // Но все компилируется нормально.
  Console.WriteLine(textData1.toupper());
  Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
}
```

Обратите внимание, что во втором вызове `WriteLine()` производится попытка обращения к методу по имени `toupper()` на динамическом элементе данных (при записи имени метода использовался неправильный регистр символов; оно должно выглядеть как `ToUpper()`). Как видите, переменная `textData1` имеет тип `string`, а потому известно, что у этого типа отсутствует метод с именем, записанным полностью в нижнем

регистре. Более того, тип `string` определенно не имеет метода по имени `Foo()`, который принимает параметры `int`, `string` и объект `DataTime`!

Тем не менее, компилятор C# ни о каких ошибках не сообщает. Однако если вызвать этот метод в `Main()`, возникнет ошибка времени выполнения с примерно таким сообщением:

```
Unhandled Exception: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
'string' does not contain a definition for 'toupper'
```

Необработанное исключение: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: string не содержит определения для toupper

Другое очевидное отличие между обращением к членам динамических и строго типизированных данных связано с тем, что когда к элементу динамических данных применяется операция точки, ожидаемое средство IntelliSense среды Visual Studio *не* активизируется. Взамен IDE-среда позволит вводить любое имя члена, какое только может прийти вам на ум.

Отсутствие возможности доступа к средству IntelliSense для динамических данных должно быть понятным. Тем не менее, как вы наверняка помните, это означает необходимость соблюдения предельной аккуратности при наборе кода C# для таких элементов данных. Любая опечатка или символ неправильного регистра в имени члена приведет к ошибке времени выполнения, в частности к генерации исключения типа `RuntimeBinderException`.

Роль сборки `Microsoft.CSharp.dll`

Для каждого создаваемого проекта C# среда Visual Studio автоматически устанавливает ссылку на сборку по имени `Microsoft.CSharp.dll` (ее можно увидеть, заглянув в папку References (Ссылки) внутри окна Solution Explorer). В этой небольшой библиотеке определено единственное пространство имен (`Microsoft.CSharp.RuntimeBinder`) с двумя классами (рис. 16.1).

Как можно догадаться по именам, классы являются строго типизированными исключениями. Более общий класс, `RuntimeBinderException`, представляет ошибку, которая будет сгенерирована при попытке обращения к несуществующему члену динамического типа данных (как в рассмотренной ситуации с методами `toupper()` и `Foo()`). Та же самая ошибка будет инициирована в случае указания некорректных данных параметров для члена, который существует.

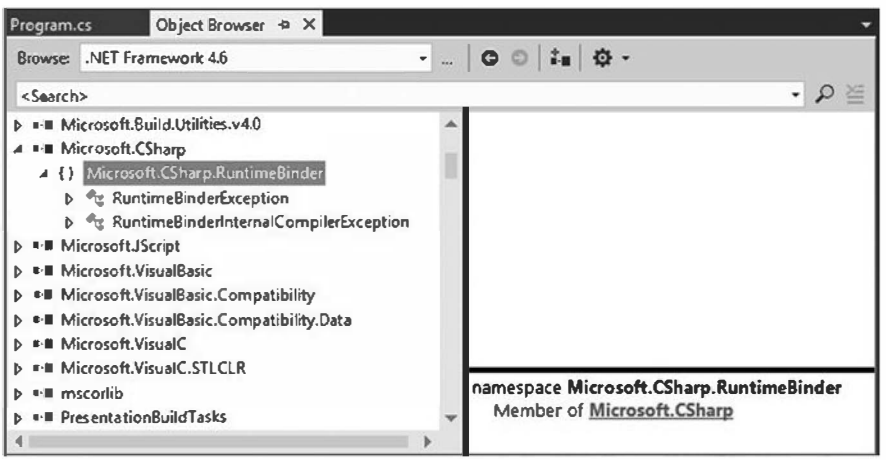


Рис. 16.1. Сборка `Microsoft.CSharp.dll`

Поскольку динамические данные настолько изменчивы, любые обращения к членам переменной, объявленной с ключевым словом dynamic, могут быть помещены внутрь подходящего блока try/catch для элегантной обработки ошибок:

```
static void InvokeMembersOnDynamicData()
{
  dynamic textData1 = "Hello";

  try
  {
    Console.WriteLine(textData1.ToUpper());
    Console.WriteLine(textData1.toupper());
    Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
  }
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Если вызвать этот метод снова, то можно заметить, что вызов ToUpper() (обратите внимание на заглавные буквы "T" и "U") работает корректно, но затем на консоль выводится сообщение об ошибке:

```
HELLO
'string' does not contain a definition for 'toupper'
string не содержит определение для toupper
```

Конечно, процесс помещения всех динамических обращений к методам в блоки try/catch довольно утомителен. Если вы тщательно следите за написанием кода и передачей параметров, то поступать так необязательно. Однако перехват исключений удобен, когда вы заранее не знаете, присутствует ли интересующий член в целевом типе.

Область применения ключевого слова dynamic

Вспомните, что неявно типизированные данные (объявленные с ключевым словом var) возможны только для локальных переменных в области действия члена. Ключевое слово var никогда не может использоваться с возвращаемым значением, параметром или членом класса/структуры. Тем не менее, это не касается ключевого слова dynamic. Взгляните на следующее определение класса:

```
class VeryDynamicClass
{
  // Динамическое поле.
  private static dynamic myDynamicField;

  // Динамическое свойство.
  public dynamic DynamicProperty { get; set; }

  // Динамический тип возврата и динамический тип параметра.
  public dynamic DynamicMethod(dynamic dynamicParam)
  {
    // Динамическая локальная переменная.
    dynamic dynamicLocalVar = "Local variable";

    int myInt = 10;
    if (dynamicParam is int)
    {
      return dynamicLocalVar;
    }
```

```
    else
    {
      return myInt;
    }
  }
}
```

Теперь можно было бы обращаться к открытым членам ожидаемым образом; однако при работе с динамическими методами и свойствами нет полной уверенности в том, каким будет тип данных! По правде говоря, определение `VeryDynamicClass` может оказаться не особенно полезным в реальном приложении, но оно иллюстрирует область, где допускается применять ключевое слово `dynamic`.

Ограничения ключевого слова `dynamic`

Хотя с использованием ключевого слова `dynamic` можно определять разнообразные сущности, с ним связаны некоторые ограничения. Хотя они не настолько впечатляющие, но следует помнить, что элементы динамических данных не могут применять лямбда-выражения или анонимные методы C# при вызове метода. Например, показанный ниже код всегда будет давать в результате ошибки, даже если целевой метод на самом деле принимает параметр типа делегата, который, в свою очередь, принимает значение `string` и возвращает `void`:

```
dynamic a = GetDynamicObject();

// Ошибка! Методы на динамических данных не могут использовать лямбда-
выражения!
a.Method(arg => Console.WriteLine(arg));
```

Чтобы обойти это ограничение, понадобится работать с лежащим в основе делегатом напрямую, используя приемы из главы 10. Еще одно ограничение заключается в том, что динамический элемент данных не может воспринимать расширяющие методы (см. главу 11). К сожалению, это касается также всех расширяющих методов из API-интерфейсов LINQ. Следовательно, переменная, объявленная с ключевым словом `dynamic`, имеет ограниченное применение в рамках LINQ to Objects и других технологий LINQ:

```
dynamic a = GetDynamicObject();

// Ошибка! Динамические данные не могут найти расширяющий метод Select()!
var data = from d in a select d;
```

Практическое использование ключевого слова `dynamic`

С учетом того, что динамические данные не являются строго типизированными, не проверяются на этапе компиляции, не имеют возможности запускать средство IntelliSense и не могут быть целью запроса LINQ, совершенно корректно предположить, что применение ключевого слова `dynamic` лишь потому, что оно существует, представляет собой плохую практику программирования.

Тем не менее, в редких обстоятельствах ключевое слово `dynamic` может радикально сократить объем вводимого вручную кода. В частности, при построении приложения .NET, в котором интенсивно используется позднее связывание (через рефлексию), ключевое слово `dynamic` может сэкономить время на наборе кода. Аналогично, при разработке приложения .NET, которое должно взаимодействовать с унаследованными библиотеками COM (такими как входящие в состав продуктов Microsoft Office), за счет использования ключевого слова `dynamic` можно значительно упростить кодовую базу.

В качестве финального примера можно привести веб-сайты, построенные с применением шаблона проектирования MVC: они часто используют тип `ViewBag`, к которому также допускается производить доступ в упрощенной манере с помощью ключевого слова `dynamic`.

Как с любым "сокращением", прежде чем его использовать, необходимо взвесить все "за" и "против". Применение ключевого слова `dynamic` — это компромисс между краткостью кода и безопасностью к типам. В то время как C# в своей основе является строго типизированным языком, динамическое поведение можно задействовать (или нет) от вызова к вызову. Всегда помните, что использовать ключевое слово `dynamic` необязательно. Тот же самый конечный результат можно получить, написав альтернативный код вручную (правда, обычно намного большего объема).

Исходный код. Проект `DynamicKeyword` доступен в подкаталоге `Chapter_16`.

Роль исполняющей среды динамического языка

Теперь, когда вы лучше понимаете сущность "динамических данных", давайте посмотрим, как их обрабатывать. Начиная с версии .NET 4.0 общеязыковая исполняющая среда (Common Language Runtime — CLR) получила дополняющую среду времени выполнения, которая называется *исполняющей средой динамического языка* (Dynamic Language Runtime — DLR). Концепция "динамической исполняющей среды" определенно не нова. На самом деле ее много лет используют такие языки программирования, как JavaScript, LISP, Ruby и Python. Выражаясь кратко, динамическая исполняющая среда предоставляет динамическим языкам возможность обнаруживать типы полностью во время выполнения без каких-либо проверок на этапе компиляции.

Если у вас есть опыт работы со строго типизированными языками (включая C# без динамических типов), понятие такой исполняющей среды может показаться неподходящим. В конце концов, вы обычно хотите выявлять ошибки на этапе компиляции, а не во время выполнения, когда только возможно. Тем не менее, динамические языки и исполняющие среды предлагают ряд интересных средств, включая перечисленные ниже.

- Исключительно гибкая кодовая база. Можно проводить рефакторинг кода, не внося многочисленных изменений в типы данных.
- Простой способ взаимодействия с разнообразными типами объектов, которые построены на разных платформах и языках программирования.
- Способ добавления или удаления членов типа в памяти во время выполнения.

Одна из задач среды DLR связана с тем, чтобы позволить различным динамическим языкам работать с исполняющей средой .NET и предоставлять им возможность взаимодействия с другим кодом .NET. Двумя популярными динамическими языками, которые используют DLR, являются IronPython и IronRuby. Эти языки находятся в "динамической вселенной", где типы определяются исключительно во время выполнения. К тому же упомянутые языки имеют доступ ко всему богатству библиотек базовых классов .NET. А еще лучше то, что благодаря наличию ключевого слова `dynamic` их кодовые базы могут взаимодействовать с языком C# (и наоборот).

На заметку! В этой главе не будет показано, как применять среду DLR для интеграции с динамическими языками. Необходимые детали вы можете найти на веб-сайтах IronPython (`http://ironpython.codeplex.com`) и IronRuby (`http://ironruby.codeplex.com/`).

Роль деревьев выражений

Для описания динамического вызова в нейтральных терминах среда DLR использует *деревья выражений*. Например, когда среда DLR встречает код C# вроде следующего:

```
dynamic d = GetSomeData();
d.SuperMethod(12);
```

она автоматически строит дерево выражения, которое, в сущности, гласит: "Вызвать метод по имени `SuperMethod` на объекте `d` с передачей числа `12` в качестве аргумента". Затем эта информация (формально называемая *полезной нагрузкой*) передается корректному связывателю времени выполнения, который может быть динамическим связывателем C#, динамическим связывателем IronPython или даже (как будет объясняться вскоре) унаследованным объектом COM.

После этого запрос отображается на необходимую структуру вызовов для целевого объекта. Замечательная характеристика деревьев выражений (помимо того, что их не приходится создавать вручную) заключается в том, что они позволяют писать фиксированный оператор кода C# и не беспокоиться о том, какой будет действительная цель (объект COM, кодовая база IronPython или IronRuby и т.д.). На рис. 16.2 проиллюстрирована высокоуровневая концепция деревьев выражений.

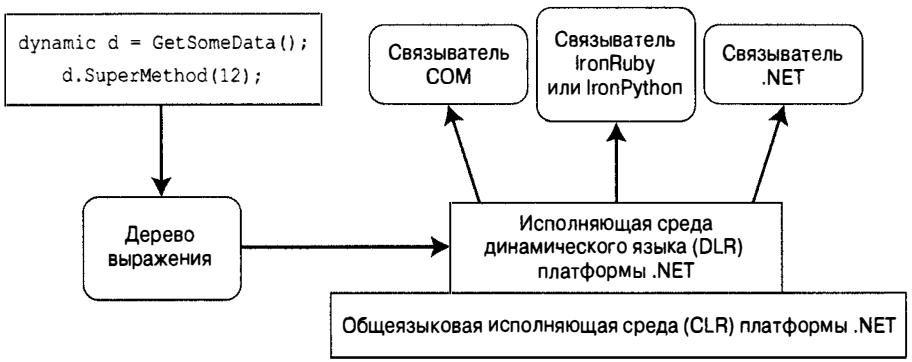


Рис. 16.2. Деревья выражений фиксируют динамические вызовы в нейтральных терминах и обрабатываются связывателями

Роль пространства имен `System.Dynamic`

Сборка `System.Core.dll` включает пространство имен под названием `System.Dynamic`. По правде говоря, шансы, что вам когда-либо придется применять типы из этого пространства имен, весьма невелики. Однако если вы являетесь производителем языка и желаете обеспечить своему динамическому языку возможность взаимодействия со средой DLR, то сможете задействовать пространство имен `System.Dynamic` для построения специального связывателя времени выполнения.

Типы из пространства имен `System.Dynamic` здесь подробно не рассматриваются, но в случае заинтересованности обращайтесь в документацию .NET Framework 4.6 SDK. Для практических нужд просто знайте, что это пространство имен предоставляет необходимую инфраструктуру, которая делает динамический язык "осведомленным о платформе .NET".

Динамический поиск в деревьях выражений во время выполнения

Как уже объяснялось, среда DLR будет передавать деревья выражений целевому объекту; тем не менее, на этот процесс диспетчеризации влияет несколько факторов. Если динамический тип данных указывает в памяти на объект COM, то дерево выражения отправляется реализации низкоуровневого интерфейса COM по имени `IDispatch`. Как вам может быть известно, упомянутый интерфейс представляет собой способ, которым COM внедряет собственный набор динамических служб. Однако объекты COM можно использовать в приложении .NET без применения DLR или ключевого слова `dynamic` языка C#. Тем не менее, такой подход (как вы увидите) сопряжен с написанием более сложного кода на C#.

Если динамические данные не указывают на объект COM, то дерево выражения может быть передано объекту, реализующему интерфейс `IDynamicObject`. Этот интерфейс используется "за кулисами", чтобы позволить языку, подобному IronRuby, принимать дерево выражения DLR и отображать его на специфические средства языка Ruby.

Наконец, если динамические данные указывают на объект, который *не* является объектом COM и *не* реализует интерфейс `IDynamicObject`, то это нормальный повседневный объект .NET. В таком случае дерево выражения передается на обработку связывателю исполняющей среды C#. Процесс отображения дерева выражений на специфические средства платформы .NET вовлекает в дело службы рефлексии.

После того как дерево выражения обработано определенным связывателем, динамические данные преобразуются в реальный тип данных в памяти, после чего вызывается корректный метод со всеми необходимыми параметрами. Теперь давайте рассмотрим несколько практических применений DLR, начав с упрощения вызовов .NET с поздним связыванием.

Упрощение вызовов с поздним связыванием посредством динамических типов

Одним из случаев, когда имеет смысл использовать ключевое слово `dynamic`, может быть работа со службами рефлексии, а именно — вызов методов с поздним связыванием. В главе 15 приводилось несколько примеров, когда вызовы методов такого рода могут быть полезными — чаще всего при построении расширяемого приложения. Там вы узнали, как применять метод `Activator.CreateInstance()` для создания объекта типа, о котором ничего не известно на этапе компиляции (помимо его отображаемого имени). Затем с помощью типов из пространства имен `System.Reflection` можно обращаться к членам объекта через механизм позднего связывания. Вспомните показанный ниже пример из главы 15:

```
static void CreateUsingLateBinding(Assembly asm)
{
  try
  {
    // Получить метаданные для типа Minivan.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");

    // Создать экземпляр Minivan на лету.
    object obj = Activator.CreateInstance(miniVan);

    // Получить информацию о TurboBoost.
    MethodInfo mi = miniVan.GetMethod("TurboBoost");
```

```
    // Вызвать метод (null означает отсутствие параметров).
    mi.Invoke(obj, null);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

В то время как этот код функционирует ожидаемым образом, нельзя не отметить его некоторую громоздкость. Здесь приходится вручную работать с классом `MethodInfo`, вручную запрашивать метаданные и т.д. В следующей версии этого метода используется ключевое слово `dynamic` и среда DLR:

```
static void InvokeMethodWithDynamicKeyword(Assembly asm)
{
  try
  {
    // Получить метаданные для типа Minivan.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");
    // Создать экземпляр Minivan на лету и вызвать метод.
    dynamic obj = Activator.CreateInstance(miniVan);
    obj.TurboBoost();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

За счет объявления переменной `obj` с ключевым словом `dynamic` вся рутинная работа, связанная с рефлексией, перекладывается на DLR.

Использование ключевого слова `dynamic` для передачи аргументов

Полезность среды DLR становится еще более очевидной, когда нужно выполнять вызовы методов с поздним связыванием, которые принимают параметры. В случае применения "многословных" обращений к рефлексии аргументы нуждаются в упаковке внутрь массива экземпляров `object`, который передается методу `Invoke()` класса `MethodInfo`.

Чтобы проиллюстрировать использование на примере, создадим новый проект консольного приложения C# по имени `LateBindingWithDynamic`. Добавим к текущему решению проект библиотеки классов (выбором пункта меню File⇨Add⇨New Project (Файл⇨Добавить⇨Новый проект)) под названием `MathLibrary`. Переименуем первоначальный файл `Class1.cs` в проекте `MathLibrary` на `SimplaMath.cs` и реализуем класс, как показано ниже:

```
public class SimpleMath
{
  public int Add(int x, int y)
  {
    return x + y;
  }
}
```

После компиляции сборки `MathLibrary.dll` поместим ее копию в подкаталог `bin\Debug` проекта `LateBindingWithDynamic`. (Щелкнув на кнопке Show All Files (Показать все

Рис. 16.3. Проект LateBindingWithDynamic содержит закрытую копию сборки MathLibrary.dll

файлы) для нужных проектов в окне Solution Explorer, можно просто перетащить файл из одного проекта в другой). В этот момент окно Solution Explorer должно выглядеть так, как показано на рис. 16.3.

На заметку! Помните, что главная цель позднего связывания — позволить приложению создать объект типа, для которого не предусмотрено записи в манифесте. Именно поэтому требуется вручную копировать сборку MathLibrary.dll в выходную папку проекта консольного приложения, а не устанавливать ссылку на сборку в Visual Studio.

Теперь импортируем пространство имен System.Reflection в файл Program.cs проекта консольного приложения. Добавим в класс Program следующий метод, который вызывает метод Add() с применением типичных обращений к API-интерфейсу рефлексии:

```
private static void AddWithReflection()
{
 Assembly asm = Assembly.Load("MathLibrary");
 try
 {
  // Получить метаданные для типа SimpleMath.
  Type math = asm.GetType("MathLibrary.SimpleMath");
  // Создать объект SimpleMath на лету.
  object obj = Activator.CreateInstance(math);
  // Получить информацию о методе Add().
  MethodInfo mi = math.GetMethod("Add");
  // Вызвать метод (с параметрами).
  object[] args = { 10, 70 };
  Console.WriteLine("Result is: {0}", mi.Invoke(obj, args));
 }
 catch (Exception ex)
 {
  Console.WriteLine(ex.Message);
 }
}
```

Ниже показано, как можно упростить предыдущую логику, используя ключевое слово dynamic:

```
private static void AddWithDynamic()
{
 Assembly asm = Assembly.Load("MathLibrary");
 try
 {
  // Получить метаданные для типа SimpleMath.
  Type math = asm.GetType("MathLibrary.SimpleMath");
  // Создать объект SimpleMath на лету.
  dynamic obj = Activator.CreateInstance(math);
  // Обратите внимание, насколько легко теперь вызывать метод Add().
  Console.WriteLine("Result is: {0}", obj.Add(10, 70));
 }
```

```
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Вызвав оба метода в `Main()`, вы получите идентичный вывод. Однако в случае применения ключевого слова `dynamic` сокращается объем кода. Благодаря динамически определяемым данным вам больше не придется вручную упаковывать аргументы внутрь массива экземпляров `object`, запрашивать метаданные сборки либо иметь дело с другими деталями подобного рода. При построении приложения, в котором интенсивно используется динамическая загрузка и позднее связывание, экономия на кодировании со временем становится еще более ощутимой.

Исходный код. Проект `LateBindingWithDynamic` доступен в подкаталоге `Chapter_16`.

Упрощение взаимодействия с COM посредством динамических данных

Давайте рассмотрим еще один полезный сценарий для ключевого слова `dynamic` в рамках проекта взаимодействия с COM. Если у вас нет опыта разработки для COM, то имейте в виду, что скомпилированная библиотека COM содержит метаданные подобно библиотеке .NET, но ее формат совершенно другой. По этой причине, когда программа .NET нуждается во взаимодействии с объектом COM, то первым делом потребуется сгенерировать так называемую *сборку взаимодействия* (описанную ниже). Задача довольно проста. Нужно открыть диалоговое окно Add Reference (Добавление ссылки), перейти на вкладку COM и отыскать библиотеку COM, которую необходимо применять (рис. 16.4).

На заметку! Имейте в виду, что некоторые важные объектные модели Microsoft (включая продукты Office) в настоящее время доступны только через взаимодействие с COM. Таким образом, даже если вы не имеете опыта непосредственного построения приложений COM, то может понадобиться потреблять их внутри программы .NET.

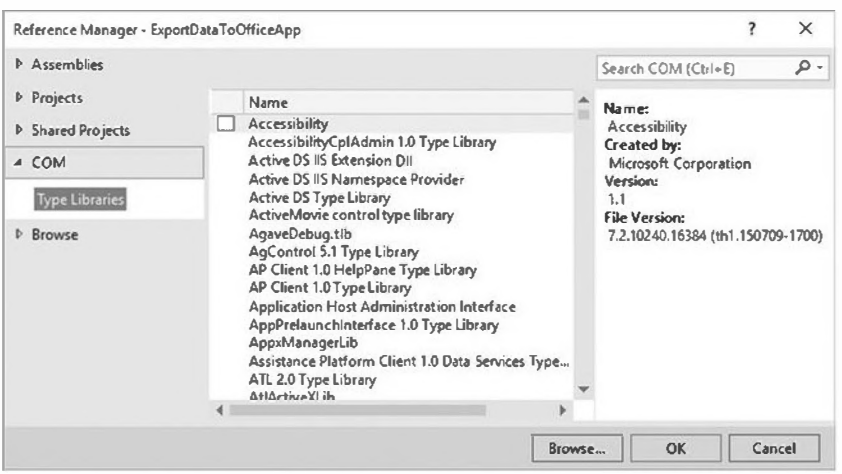


Рис. 16.4. На вкладке COM диалогового окна Add Reference отображаются все зарегистрированные библиотеки COM на машине

После выбора COM-библиотеки IDE-среда отреагирует генерацией новой сборки, которая включает описания .NET метаданных COM. Формально она называется *сборкой взаимодействия* и не содержит какого-либо кода реализации кроме небольшой порции кода, который помогает транслировать события COM в события .NET. Тем не менее, эти сборки взаимодействия полезны тем, что защищают кодовую базу .NET от сложностей внутреннего механизма COM.

В коде C# можно напрямую работать со сборкой взаимодействия, позволяя среде CLR (а если используется ключевое слово `dynamic`, то среде DLR) автоматически отображать типы данных .NET на типы COM и наоборот. "За кулисами" данные маршализируются между приложениями .NET и COM с применением вызываемой оболочки времени выполнения (Runtime Callable Wrapper — RCW), которая по существу является динамически сгенерированным прокси. Прокси RCW будет маршализировать и трансформировать типы данных .NET в типы COM и отображать любые возвращаемые значения COM на их эквиваленты .NET.

На рис. 16.5 изображена общая картина взаимодействия .NET с COM.

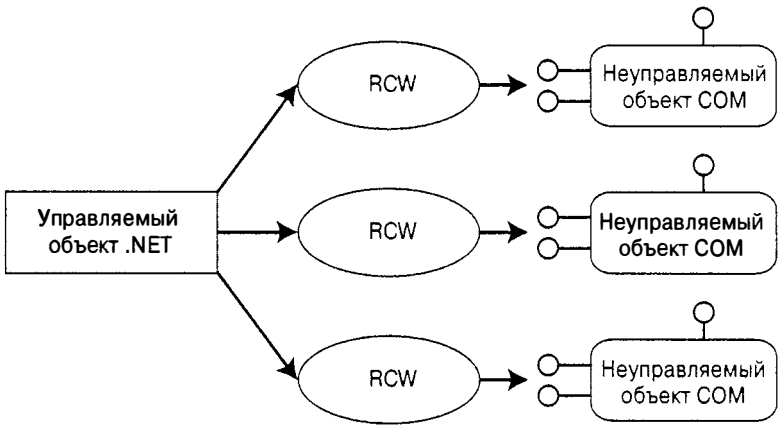


Рис. 16.5. Программы .NET взаимодействуют с объектами COM, используя прокси под названием RCW

Роль основных сборок взаимодействия

Многие библиотеки COM, созданные поставщиками библиотек COM (вроде библиотек Microsoft COM, обеспечивающих доступ к объектной модели продуктов Microsoft Office), предоставляют "официальную" сборку взаимодействия, которая называется *основной сборкой взаимодействия* (primary interop assembly — PIA). Сборки PIA — это оптимизированные сборки взаимодействия, которые приводят в порядок (и возможно расширяют) код, обычно генерируемый при добавлении ссылки на библиотеку COM с помощью диалогового окна Add Reference.

Сборки PIA обычно перечислены в разделе Assemblies (Сборки) диалогового окна Add Reference (в области Extensions (Расширения)). В действительности, если вы ссылаетесь на библиотеку COM из вкладки COM диалогового окна Add Reference, то Visual Studio не генерирует новую библиотеку взаимодействия, как это делалось бы обычно, а взамен применяет предоставленную сборку PIA. На рис. 16.6 показана сборка PIA объектной модели Microsoft Office Excel, которая будет использоваться в следующем примере.

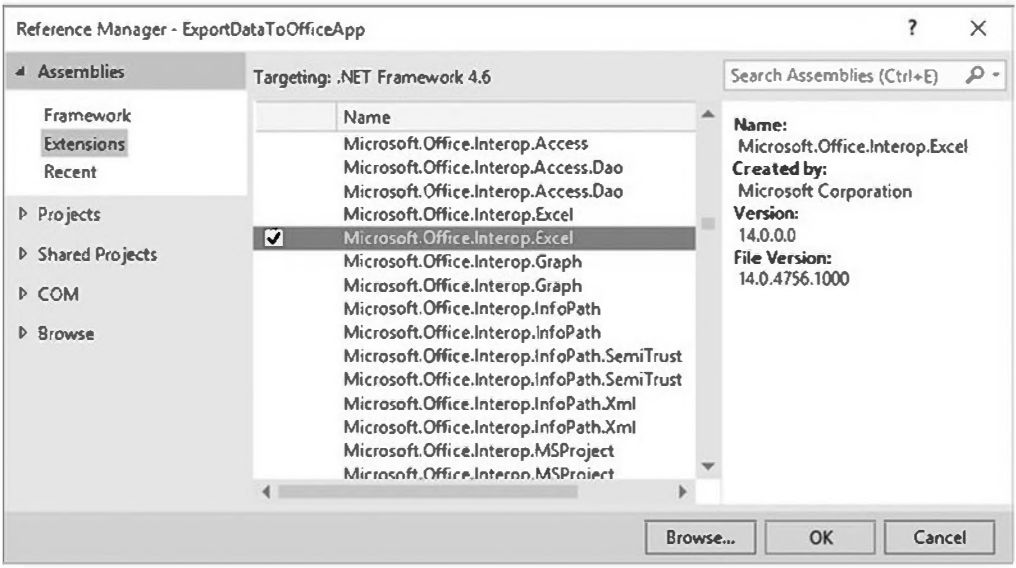


Рис. 16.6. Сборки PIA перечислены в области Extensions раздела Assemblies внутри диалогового окна Add Reference

Встраивание метаданных взаимодействия

До выхода .NET 4.0, когда приложение C# задействовало библиотеку COM (через сборку PIA или нет), на клиентской машине необходимо было обеспечить наличие копии сборки взаимодействия. Помимо увеличения размера установочного пакета приложения сценарий установки должен был также проверять, присутствуют ли сборки PIA, и если нет, то устанавливать их копии в GAC.

Однако в .NET 4.0 и последующих версиях данные взаимодействия теперь можно встраивать прямо в скомпилированное приложение .NET. В этом случае поставлять копию сборки взаимодействия вместе с приложением .NET больше не понадобится, т.к. все необходимые метаданные взаимодействия жестко закодированы в приложении .NET.

По умолчанию после выбора в диалоговом окне Add Reference библиотеки COM (со сборкой PIA или без нее) IDE-среда автоматически устанавливает свойство Embed Interop Types (Встраивать типы взаимодействия) для библиотеки в `True`. Чтобы увидеть эту настройку, нужно выбрать ссылаемую сборку взаимодействия в папке References (Ссылки) внутри окна Solution Explorer и открыть ее окно свойств (рис. 16.7).

Компилятор C# будет включать только те части библиотеки взаимодействия, которые действительно используются. Таким образом, даже если реальная библиотека взаимодействия содержит описания .NET сотен объектов COM, в приложение попадет только подмножество определений, которые на самом деле применяются в написанном коде C#. Помимо сокращения размера приложения, поставляемого клиенту, также упрощается процесс установки, потому что не придется устанавливать сборки PIA, которые отсутствуют на целевой машине.

Общие сложности взаимодействия с COM

Давайте перед переходом к следующему примеру рассмотрим еще одну подготовительную тему. До выпуска DLR при написании кода C#, обращающегося к библиотеке COM (через сборку взаимодействия), неизбежно возникало несколько сложностей.

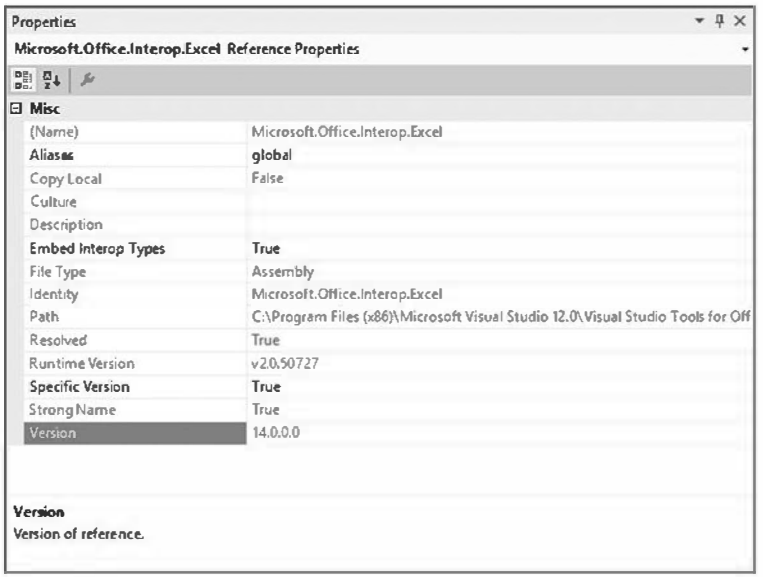


Рис. 16.7. Логику сборки взаимодействия можно встроить непосредственно в приложение .NET

Например, многие библиотеки COM определяли методы, принимающие необязательные аргументы, которые вплоть до версии .NET 3.5 в языке C# не поддерживались. Это требовало указания значения `Type.Missing` для каждого вхождения необязательного аргумента. Скажем, если метод COM принимал пять аргументов, и все они были необязательными, приходилось писать следующий код C#, чтобы принимать стандартные значения:

```
myComObj.SomeMethod(Type.Missing, Type.Missing, Type.Missing,
  Type.Missing, Type.Missing);
```

К счастью, теперь имеется возможность записывать показанный ниже упрощенный код, поскольку если не указано специфичное значение, то на этапе компиляции вместо него вставляется `Type.Missing`:

```
myComObj.SomeMethod();
```

В связи с этим следует отметить, что многие методы COM поддерживают именованные аргументы, которые, как объяснялось в главе 4, позволяют передавать значения членам в любом порядке. Учитывая наличие поддержки той же самой возможности в языке C#, допускается просто "пропускать" необязательные аргументы, которые неважны, и устанавливать только те из них, которые нужны в текущий момент.

Еще одна распространенная сложность взаимодействия с COM была связана с тем фактом, что многие методы COM были спроектированы так, чтобы принимать и возвращать специфический тип данных по имени `Variant`. Во многом похоже на ключевое слово `dynamic` языка C#, типу данных `Variant` может быть присвоен на лету любой тип данных COM (строка, ссылка на интерфейс, числовое значение и т.д.). До появления ключевого слова `dynamic` передача или прием элементов данных типа `Variant` требовал немалых ухищрений, обычно связанных с многочисленными операциями приведения.

Когда свойство Embed Interop Types установлено в `True`, все COM-типы `Variant` автоматически отображаются на динамические данные. В итоге не только сокращается потребность в паразитных операциях приведения при работе с типами данных `Variant`, но также еще больше скрываются некоторые сложности, присущие COM, вроде работы с индексаторами COM.

Чтобы продемонстрировать, каким образом необязательные аргументы, именованные аргументы и ключевое слово `dynamic` совместно способствуют упрощению взаимодействия с COM, мы построим приложение, в котором используется объектная модель Microsoft Office. В этом примере будет шанс применить новые средства, а также обойтись без них, и затем сравнить объем работы в обоих случаях.

На заметку! Если вы не имеете опыта построения графических пользовательских интерфейсов с помощью Windows Forms, то можете просто загрузить готовое решение в Visual Studio и экспериментировать с кодом, а не создавать приложение вручную.

Взаимодействие с COM с использованием динамических данных C#

Предположим, что есть приложение Windows Forms (под названием `ExportDataToOfficeApp`) с графическим пользовательским интерфейсом, главное окно которого содержит элемент управления `DataGridView` по имени `dataGridCars`. В том же окне присутствуют два элемента управления `Button`, один из которых обеспечивает открытие специального диалогового окна для вставки новой строки данных в сетку, а другой отвечает за экспортирование данных сетки в электронную таблицу Excel. На рис. 16.8 показан завершенный графический пользовательский интерфейс.

Рис. 16.8. Графический пользовательский интерфейс для примера взаимодействия с COM

Элемент управления `DataGridView` заполняется начальными данными в обработчике события `Load` формы (класс `Car`, применяемый в качестве параметра типа для обобщенного `List<T>` — это простой класс со свойствами `Color`, `Make` и `PetName`):

```
public partial class MainForm : Form
{
  List<Car> carsInStock = null;
  public MainForm()
  {
    InitializeComponent();
  }
```

```
  private void MainForm_Load(object sender, EventArgs e)
  {
    carsInStock = new List<Car>
    {
      new Car {Color="Green", Make="VW", PetName="Mary"},
      new Car {Color="Red", Make="Saab", PetName="Mel"},
      new Car {Color="Black", Make="Ford", PetName="Hank"},
      new Car {Color="Yellow", Make="BMW", PetName="Davie"}
    };
    UpdateGrid();
  }
  private void UpdateGrid()
  {
    // Сбросить источник данных.
    dataGridCars.DataSource = null;
    dataGridCars.DataSource = carsInStock;
  }
}
```

В обработчике события `Click` кнопки Add New Entry to Inventory (Добавить новую запись в инвентарную ведомость) открывается специальное диалоговое окно, которое позволяет пользователю ввести новые данные для объекта `Car`; после щелчка на кнопке OK данные добавляются в сетку. (Код этого диалогового окна здесь не приводится, так что за подробностями обращайтесь в решение, доступное в рамках загружаемого кода примеров.) В случае если вы прорабатываете настоящий пример, включите в проект файлы `NewCarDialog.cs`, `NewCarDialog.designer.cs` и `NewCarDialog.resx` (все они находятся в загружаемом коде примеров). После этого можно реализовать обработчик щелчка на кнопке Add New Entry to Inventory:

```
private void btnAddNewCar_Click(object sender, EventArgs e)
{
  NewCarDialog d = new NewCarDialog();
  if (d.ShowDialog() == DialogResult.OK)
  {
    // Добавить в ведомость запись о новом автомобиле.
    carsInStock.Add(d.theCar);
    UpdateGrid();
  }
}
```

Центральной частью примера является обработчик события `Click` для кнопки Export Current Inventory to Excel (Экспортировать текущую инвентарную ведомость в Excel). Используя диалоговое окно Add Reference, добавим ссылку на основную сборку взаимодействия `Microsoft.Office.Interop.Excel.dll` (как было показано ранее на рис. 16.6). Добавим приведенный ниже псевдоним пространства имен в главный файл кода формы. Имейте в виду, что при взаимодействии с библиотеками COM псевдоним определять не обязательно. Тем не менее, поступая так, вы получаете квалификатор для всех импортированных объектов COM, который удобен, если часть объектов COM имеют имена, конфликтующие с именами ваших типов .NET.

```
// Создать псевдоним для объектной модели Excel.
using Excel = Microsoft.Office.Interop.Excel;
```

Реализуйте следующий обработчик события `Click`, чтобы он вызывал закрытый вспомогательный метод по имени `ExportToExcel()`:

```
private void btnExportToExcel_Click(object sender, EventArgs e)
{
  ExportToExcel(carsInStock);
}
```

Поскольку библиотека COM была импортирована с применением Visual Studio, сборка PIA автоматически сконфигурирована так, что используемые метаданные будут встраиваться в приложение .NET (вспомните о роли свойства Embed Interop Types). Таким образом, все COM-типы `Variant` будут реализованы как типы данных `dynamic`. Более того, можно применять необязательные и именованные аргументы C#. С учетом всего сказанного вот как будет выглядеть реализация метода `ExportToExcel()`:

```
static void ExportToExcel(List<Car> carsInStock)
{
  // Загрузить Excel и затем создать новую пустую рабочую книгу.
  Excel.Application excelApp = new Excel.Application();
  excelApp.Workbooks.Add();
  // В этом примере используется единственный рабочий лист.
  Excel._Worksheet workSheet = excelApp.ActiveSheet;
  // Установить заголовки столбцов в ячейках.
  workSheet.Cells[1, "A"] = "Make";
  workSheet.Cells[1, "B"] = "Color";
  workSheet.Cells[1, "C"] = "Pet Name";
  // Отобразить все данные в List<Car> на ячейки электронной таблицы.
  int row = 1;
  foreach (Car c in carsInStock)
  {
    row++;
    workSheet.Cells[row, "A"] = c.Make;
    workSheet.Cells[row, "B"] = c.Color;
    workSheet.Cells[row, "C"] = c.PetName;
  }
  // Придать симпатичный вид табличным данным.
  workSheet.Range["A1"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
  // Сохранить файл, завершить работу Excel и отобразить сообщение пользователю.
  workSheet.SaveAs(string.Format(@"{0}\Inventory.xlsx",
                   Environment.CurrentDirectory));
  excelApp.Quit();
  MessageBox.Show("The Inventory.xslx file has been saved to your app folder",
    "Export complete!");  // Файл Inventory.xslx сохранен в папке приложения.
}
```

Метод начинается с загрузки приложения Excel в память; однако на рабочем столе оно не отобразится. В данном приложении нас интересует только работа с объектной моделью Excel. Тем не менее, если необходимо отобразить пользовательский интерфейс Excel, то метод понадобится дополнить следующей строкой кода:

```
static void ExportToExcel(List<Car> carsInStock)
{
  // Загрузить Excel и затем создать новую пустую рабочую книгу.
  Excel.Application excelApp = new Excel.Application();
  // Сделать пользовательский интерфейс Excel видимым на рабочем столе.
  excelApp.Visible = true;
  ...
}
```

После создания пустого рабочего листа добавляются три столбца, именованные в соответствии со свойствами класса `Car`. Затем ячейки наполняются данными `List<Car>`, и файл сохраняется с жестко закодированным именем `Inventory.xlsx`.

Если теперь запустить приложение, добавить несколько записей и экспортировать данные в Excel, то в подкаталоге `bin\Debug` приложения Windows Forms появится файл `Inventory.xlsx`, который можно открыть в приложении Excel (рис. 16.9).

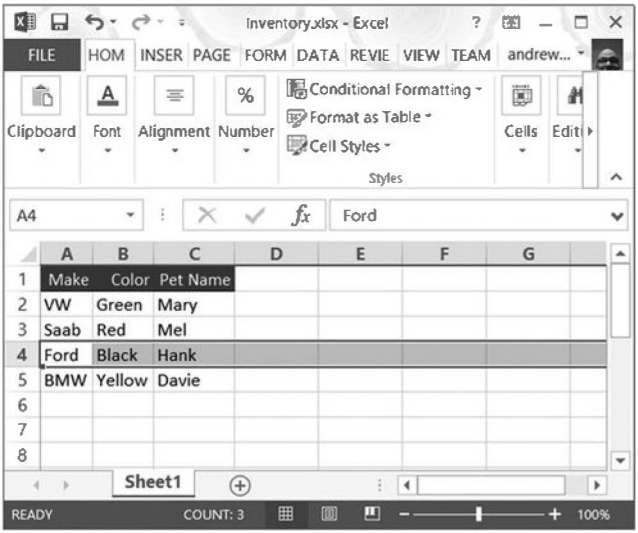


Рис. 16.9. Экспортирование данных в файл Excel

Взаимодействие с COM без использования динамических данных C#

В случае выбора сборки `Microsoft.Office.Interop.Excel.dll` (в окне Solution Explorer) и установки ее свойства Embed Interop Type в `False` появятся новые сообщения об ошибках компиляции, т.к. COM-данные `Variant` больше не воспринимаются как динамические данные, а считаются переменными типа `System.Object`. Это потребует добавления в метод `ExportToExcel()` нескольких явных операций приведения.

Кроме того, если скомпилировать проект для платформы .NET 3.5 или предшествующих версий, то утратятся преимущества необязательных/именованных параметров и потребуется явно помечать все пропущенные аргументы. Вот версия метода `ExportToExcel()`, предназначенная для более ранних версий C# (обратите внимание на возросшую сложность кода):

```
static void ExportToExcel2008(List<Car> carsInStock)
{
  Excel.Application excelApp = new Excel.Application();
  // Потребуется пометить пропущенные параметры!
  excelApp.Workbooks.Add(Type.Missing);
  // Потребуется привести Object к _Worksheet!
  Excel._Worksheet workSheet = (Excel._Worksheet)excelApp.ActiveSheet;
  // Потребуется привести каждый Object к Range и затем
  // обратиться к низкоуровневому свойству Value2!
  ((Excel.Range)excelApp.Cells[1, "A"]).Value2 = "Make";
  ((Excel.Range)excelApp.Cells[1, "B"]).Value2 = "Color";
  ((Excel.Range)excelApp.Cells[1, "C"]).Value2 = "Pet Name";
  int row = 1;
  foreach (Car c in carsInStock)
  {
    row++;
    // Потребуется привести каждый Object к Range и затем
    // обратиться к низкоуровневому свойству Value2!
    ((Excel.Range)workSheet.Cells[row, "A"]).Value2 = c.Make;
    ((Excel.Range)workSheet.Cells[row, "B"]).Value2 = c.Color;
```

```
    ((Excel.Range)workSheet.Cells[row, "C"]).Value2 = c.PetName;
  }
  // Потребуется вызвать метод get_Range()
  // с указанием всех пропущенных аргументов!
  excelApp.get_Range("A1", Type.Missing).AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2,
    Type.Missing, Type.Missing, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing);
  // Потребуется указать все пропущенные необязательные аргументы!
  workSheet.SaveAs(string.Format(@"{0}\Inventory.xlsx",
                   Environment.CurrentDirectory),
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing);
  excelApp.Quit();
  MessageBox.Show("The Inventory.xslx file has been saved to your app folder",
    "Export complete!"); // Файл Inventory.xslx сохранен в папке приложения.
}
```

Несмотря на то что конечный результат выполнения программы идентичен, очевидно, что эта версия метода намного более многословна. На этом рассмотрение ключевого слова `dynamic` языка C# и среды DLR завершено. Вы увидели, насколько данные средства способны упростить сложные задачи программирования, и (что возможно даже важнее) уяснили сопутствующие компромиссы. Делая выбор в пользу динамических данных, вы теряете изрядную часть безопасности типов, и ваша кодовая база предрасположена к намного большему числу ошибок времени выполнения.

В то время как о среде DLR можно еще рассказать многое, основное внимание в главе было сосредоточено на темах, практичных и полезных в повседневном программировании. Если вы хотите изучить расширенные средства DLR, такие как интеграция с языками написания сценариев, обратитесь в документацию .NET Framework 4.6 SDK (начните с поиска темы "Dynamic Language Runtime Overview" ("Обзор исполняющей среды динамического языка")).

Исходный код. Проект `ExportDataToOfficeApp` доступен в подкаталоге `Chapter_16`.

Резюме

Ключевое слово `dynamic`, появившееся в версии C# 4.0, позволяет определять данные, подлинная идентичность которых не известна вплоть до времени выполнения. При обработке новой исполняющей средой динамического языка (DLR) автоматически создаваемое "дерево выражения" будет передаваться подходящему связывателю динамического языка, причем полезная нагрузка будет распакована и отправлена правильному члену объекта.

С применением динамических данных и среды DLR многие сложные задачи программирования C# могут быть радикально упрощены, особенно действие по включению библиотек COM в состав приложений .NET. Кроме того, в главе было показано, что платформа .NET 4.0 и последующих версий предлагает несколько дальнейших упрощений взаимодействия с COM (которые не имеют отношения к динамическим данным), в том числе встраивание данных взаимодействия COM в разрабатываемые приложения, необязательные, а также и именованные аргументы.

Хотя все эти средства определенно могут упростить код, всегда помните о том, что динамические данные существенно снижают безопасность к типам в коде C# и открывают возможности для возникновения ошибок времени выполнения. Тщательно взвешивайте все "за" и "против" использования динамических данных в своих проектах C# и тестируйте их соответствующим образом!

ГЛАВА 17

Процессы, домены приложений и объектные контексты

В главах 14 и 15 вы изучили шаги, которые среда CLR предпринимает при выяснении местоположения ссылаемой внешней сборки, а также роль метаданных .NET. В текущей главе будут представлены детали обслуживания сборки средой CLR и отношения между процессами, доменами приложений и объектными контекстами.

Выражаясь кратко, *домены приложений* (Application Domain или просто AppDomain) — это логические подразделы внутри заданного процесса, который обслуживает набор связанных сборок .NET. Как вы увидите, каждый домен приложения в дальнейшем подразделяется на *контекстные границы*, которые используются для группирования вместе похожих по смыслу объектов .NET. Благодаря понятию контекста среда CLR способна обеспечивать надлежащую обработку объектов со специальными требованиями времени выполнения.

Хотя вполне справедливо утверждать, что многие повседневные задачи программирования не предусматривают работу с процессами, доменами приложений или объектными контекстами напрямую, их понимание важно при взаимодействии с многочисленными API-интерфейсами .NET, включая Windows Communication Foundation (WCF), многопоточную и параллельную обработку, а также сериализацию объектов.

Роль процесса Windows

Концепция "процесса" существовала в операционных системах Windows задолго до выпуска платформы .NET. Пользуясь простыми терминами, *процесс* — это выполняющаяся программа. Тем не менее, формально процесс представляет собой концепцию уровня операционной системы, которая применяется для описания набора ресурсов (таких как внешние библиотеки кода и главный поток) и необходимых распределений памяти, используемой функционирующим приложением. Для каждого загруженного в память файла `*.exe` операционная система создает отдельный изолированный процесс для применения на протяжении всего его времени существования.

При использовании такого подхода к изоляции приложений в результате получается намного более надежная и устойчивая исполняющая среда, поскольку отказ одного процесса не влияет на работу других процессов. Более того, данные в одном процессе не доступны напрямую другим процессам, если только не применяется программный API-интерфейс для распределенных вычислений вроде Windows Communication Foundation.

С учетом указанных моментов процесс можно рассматривать как фиксированную и безопасную границу для выполняющегося приложения.

Каждый процесс Windows получает уникальный идентификатор процесса (process identifier — PID) и может по мере необходимости независимо загружаться и выгружаться операционной системой (а также программно). Как вам возможно известно, в окне диспетчера задач Windows (открываемом по нажатию комбинации клавиш <Ctrl+Shift+Esc>) имеется вкладка Processes (Процессы), на которой можно просматривать разнообразные статические данные о процессах, функционирующих на машине. На вкладке Details (Подробности) можно видеть назначенный идентификатор PID и имя образа (рис. 17.1).

На заметку! В версии Windows 10 идентификатор PID процесса можно просматривать на вкладке Services (Службы).

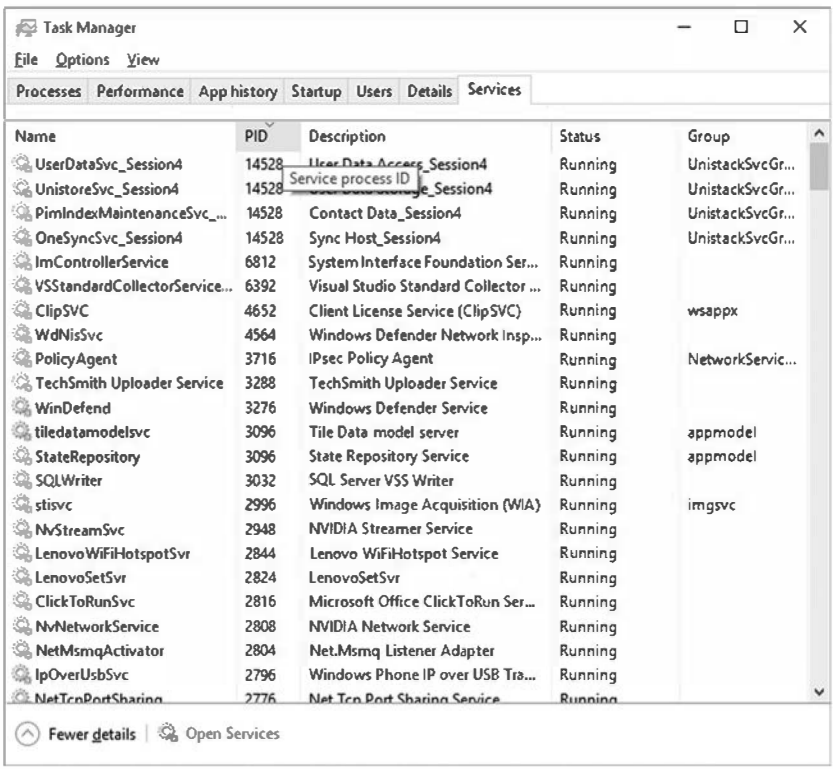


Рис. 17.1. Диспетчер задач Windows

Роль потоков

Каждый процесс Windows содержит начальный "поток", который действует как точка входа для приложения. Особенности построения многопоточных приложений на платформе .NET рассматриваются в главе 19; однако для понимания материала настоящей главы необходимо ознакомиться с несколькими рабочими определениями. *Поток* — это путь выполнения внутри процесса. Выражаясь формально, первый поток, созданный точкой входа процесса, называется *главным потоком*. В любой исполняемой программе .NET (консольном приложении, Windows-службе, приложении WPF и т.д.) точка входа

помечается с помощью метода `Main()`. При вызове этого метода главный поток создается автоматически.

Процессы, которые содержат единственный главный поток выполнения, по своей сути *безопасны в отношении потоков*, т.к. в каждый момент времени доступ к данным приложения может получать только один поток. Тем не менее, однопоточный процесс (особенно с графическим пользовательским интерфейсом) часто замедленно реагирует на действия пользователя, когда его единственный поток выполняет сложную операцию (такую как печать длинного текстового файла, сложные математические вычисления или попытка подключения к удаленному серверу, находящемуся на расстоянии тысяч километров).

Учитывая такой потенциальный недостаток однопоточных приложений, API-интерфейс Windows (а также платформа .NET) предоставляет главному потоку возможность порождения дополнительных вторичных потоков (называемых *рабочими потоками*) с использованием нескольких функций из API-интерфейса Windows наподобие `CreateThread()`. Каждый поток (первичный или вторичный) становится уникальным путем выполнения в процессе и имеет параллельный доступ ко всем разделяемым элементам данных внутри этого процесса.

Нетрудно догадаться, что разработчики обычно создают дополнительные потоки для улучшения общей степени отзывчивости программы. Многопоточные процессы обеспечивают иллюзию того, что выполнение многочисленных действий происходит более или менее одновременно. Например, приложение может порождать дополнительный рабочий поток для выполнения трудоемкой единицы работы (вроде вывода на печать крупного текстового файла). После запуска вторичного потока главный поток продолжает реагировать на пользовательский ввод, что дает всему процессу возможность достигать более высокой производительности. Однако на самом деле так происходит не всегда: применение слишком большого количества потоков в одном процессе может приводить к *ухудшению* производительности из-за того, что центральный процессор должен переключаться между активными потоками внутри процесса (а это отнимает время).

На некоторых машинах многопоточность по большей части является иллюзией, обеспечиваемой операционной системой. Машины с единственным (не поддерживающим гиперпотоки) центральным процессором не обладают возможностью обработки множества потоков в одно и то же время. Вместо этого один центральный процессор выполняет по одному потоку за единицу времени (называемую *квантом времени*), частично основываясь на приоритете потока. По истечении выделенного кванта времени выполнение существующего потока приостанавливается, позволяя другому потоку выполнять свою работу. Чтобы поток не забывал, что происходило до того, как его выполнение было приостановлено, ему предоставляется возможность записывать данные в локальное хранилище потоков (Thread Local Storage — TLS) и выделяется отдельный стек вызовов (рис. 17.2).

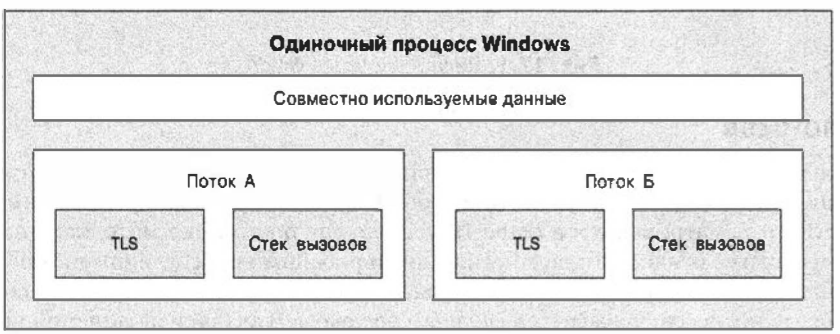


Рис. 17.2. Отношения между потоками и процессами в Windows

Если тема потоков для вас нова, не стоит беспокоиться о деталях. На этом этапе просто запомните, что любой поток представляет собой уникальный путь выполнения внутри процесса Windows. Каждый процесс имеет главный поток (созданный посредством точки входа исполняемого файла) и может содержать дополнительные потоки, которые создаются программно.

Взаимодействие с процессами на платформе .NET

Несмотря на то что с процессами и потоками не связано ничего нового, способ взаимодействия с ними в рамках платформы .NET значительно изменился (в лучшую сторону). Чтобы подготовить почву для понимания области построения многопоточных сборок (см. главу 19), давайте начнем с выяснения способов взаимодействия с процессами, используя библиотеки базовых классов .NET.

В пространстве имен `System.Diagnostics` определено несколько типов, которые позволяют программно взаимодействовать с процессами и разнообразными типами, связанными с диагностикой, такими как журнал событий системы и счетчики производительности. В этой главе нас интересуют только типы, связанные с процессами, которые описаны в табл. 17.1.

Таблица 17.1. Избранные типы пространства имен `System.Diagnostics`

Тип	Описание
`Process`	Предоставляет доступ к локальным и удаленным процессам, а также позволяет программно запускать и останавливать процессы
`ProcessModule`	Представляет модуль (`*.dll` или `*.exe`), загруженный в определенный процесс. Важно понимать, что тип `ProcessModule` может представлять *любой* модуль, т.е. двоичные сборки, основанные на COM, .NET или традиционном C
`ProcessModuleCollection`	Предоставляет строго типизированную коллекцию объектов `ProcessModule`
`ProcessStartInfo`	Указывает набор значений, применяемых при запуске процесса с помощью метода `Process.Start()`
`ProcessThread`	Представляет поток внутри заданного процесса. Имейте в виду, что тип `ProcessThread` используется для диагностирования набора потоков процесса, но не для порождения новых потоков выполнения в рамках процесса
`ProcessThreadCollection`	Предоставляет строго типизованную коллекцию объектов `ProcessThread`

Класс `System.Diagnostics.Process` позволяет анализировать процессы, выполняющиеся на заданной машине (локальные или удаленные). В классе `Process` также определены члены, предназначенные для программного запуска и завершения процессов, просмотра (или модификации) уровня приоритета процесса и получения списка активных потоков и/или загруженных модулей внутри указанного процесса. В табл. 17.2 перечислены некоторые основные свойства класса `System.Diagnostics.Process`.

Таблица 17.2. Избранные свойства класса `Process`

Свойство	Описание
`ExitTime`	Позволяет извлекать метку времени, ассоциированную с процессом, который был завершен (представленную с помощью типа `DateTime`)
`Handle`	Возвращает дескриптор (представленный типом `IntPtr`), который был назначен процессу операционной системой. Это может быть полезно при построении приложений .NET, нуждающихся во взаимодействии с неуправляемым кодом
`Id`	Позволяет получать идентификатор PID связанного процесса
`MachineName`	Позволяет получать имя компьютера, на котором выполняется связанный процесс
`MainWindowTitle`	Позволяет получать заголовок главного окна процесса (если у процесса нет главного окна, то возвращается пустая строка)
`Modules`	Предоставляет доступ к строго типизованной коллекции `ProcessModuleCollection`, представляющей набор модулей (`*.dll` или `*.exe`), которые были загружены внутри текущего процесса
`ProcessName`	Позволяет получать имя процесса (которое, как и можно было предполагать, представляет собой имя самого приложения)
`Responding`	Позволяет получать значение, которое указывает, реагирует ли пользовательский интерфейс процесса на пользовательский ввод (или в текущий момент находится в "зависшем" состоянии)
`StartTime`	Позволяет получать значение времени, когда был запущен процесс (представленное с помощью типа `DateTime`)
`Threads`	Позволяет получать набор потоков, выполняемых в связанном процессе (представленный посредством коллекции объектов `ProcessThread`)

Кроме перечисленных выше свойств в классе `System.Diagnostics.Process` определено несколько полезных методов (табл. 17.3).

Таблица 17.3. Избранные методы класса `Process`

Метод	Описание
`CloseMainWindow()`	Этот метод закрывает процесс, который содержит пользовательский интерфейс, отправляя его главному окну сообщение о закрытии
`GetCurrentProcess()`	Этот статический метод возвращает новый объект `Process`, который представляет процесс, активный в текущий момент
`GetProcesses()`	Этот статический метод возвращает массив объектов `Process`, представляющих процессы, которые выполняются на заданной машине
`Kill()`	Этот метод немедленно останавливает связанный процесс
`Start()`	Этот метод запускает процесс

Перечисление выполняющихся процессов

Для иллюстрации способа манипулирования объектами `Process` создадим новый проект консольного приложения C# по имени `ProcessManipulator` и определим следующий вспомогательный статический метод в классе `Program` (не забудьте импортировать в файл кода пространство имен `System.Diagnostics`):

```
static void ListAllRunningProcesses()
{
  // Получить все процессы на локальной машине, упорядоченные по PID.
  var runningProcs =
    from proc in Process.GetProcesses(".") orderby proc.Id select proc;
  // Вывести для каждого процесса идентификатор PID и имя.
  foreach(var p in runningProcs)
  {
    string info = string.Format("-> PID: {0}\tName: {1}",
      p.Id, p.ProcessName);
    Console.WriteLine(info);
  }
  Console.WriteLine("*************************************\n");
}
```

Статический метод `Process.GetProcesses()` **возвращает массив объектов** `Process`, **которые представляют выполняющиеся процессы на целевой машине (передаваемая методу строка "." обозначает локальный компьютер). После получения массива объектов** `Process` **можно обращаться к любым членам, описанным в табл. 17.2 и 17.3. Здесь просто для каждого процесса выводятся идентификатор PID и имя с упорядочением по PID. Добавив в** `Main()` **вызов метода** `ListAllRunningProcesses()`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Processes *****\n");
  ListAllRunningProcesses();
  Console.ReadLine();
}
```

можно будет увидеть список имен и идентификаторов PID всех процессов, запущенных на локальной машине. Ниже показана часть вывода (ваш вывод наверняка будет отличаться):

```
***** Fun with Processes *****

-> PID: 0      Name: Idle
-> PID: 4      Name: System
-> PID: 108    Name: iexplore
-> PID: 268    Name: smss
-> PID: 432    Name: csrss
-> PID: 448    Name: svchost
-> PID: 472    Name: wininit
-> PID: 504    Name: csrss
-> PID: 536    Name: winlogon
-> PID: 560    Name: services
-> PID: 584    Name: lsass
-> PID: 592    Name: lsm
-> PID: 660    Name: devenv
-> PID: 684    Name: svchost
-> PID: 760    Name: svchost
-> PID: 832    Name: svchost
-> PID: 844    Name: svchost
-> PID: 856    Name: svchost
-> PID: 900    Name: svchost
-> PID: 924    Name: svchost
-> PID: 956    Name: VMwareService
-> PID: 1116   Name: spoolsv
-> PID: 1136   Name: ProcessManipulator.vshost
*************************************
```

Исследование конкретного процесса

В дополнение к полному списку всех выполняющихся процессов на заданной машине статический метод `Process.GetProcessById()` позволяет получать одиночный объект `Process` по ассоциированному идентификатору PID. В случае запроса несуществующего PID генерируется исключение `ArgumentException`. Например, чтобы получить объект `Process`, представляющий процесс с PID, равным 987, можно написать следующий код:

```
// Если процесс с PID, равным 987, не существует,
// сгенерируется исключение во время выполнения.
static void GetSpecificProcess()
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(987);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

К этому моменту вы уже знаете, как получить список всех процессов, а также специфичный процесс на машине посредством поиска по PID. Наряду с выяснением идентификаторов PID и имен процессов класс `Process` позволяет просматривать набор текущих потоков и библиотек, применяемых внутри заданного процесса. Давайте посмотрим, как это делается.

Исследование набора потоков процесса

Набор потоков представлен в виде строго типизованной коллекции `ProcessThreadCollection`, которая содержит определенное количество отдельных объектов `ProcessThread`. Для примера предположим, что в текущее приложение добавлен приведенный ниже вспомогательный статический метод:

```
static void EnumThreadsForPid(int pID)
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(pID);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }

  // Вывести статистические сведения по каждому потоку в указанном процессе.
  Console.WriteLine("Here are the threads used by: {0}",
    theProc.ProcessName);
  ProcessThreadCollection theThreads = theProc.Threads;

  foreach(ProcessThread pt in theThreads)
  {
    string info =
```

```
      string.Format("-> Thread ID: {0}\tStart Time: {1}\tPriority: {2}",
        pt.Id , pt.StartTime.ToShortTimeString(), pt.PriorityLevel);
    Console.WriteLine(info);
  }
  Console.WriteLine("*************************************\n");
}
```

Как видите, свойство `Threads` в типе `System.Diagnostics.Process` предоставляет доступ к классу `ProcessThreadCollection`. Здесь для каждого потока внутри указанного клиентом процесса выводится назначенный идентификатор потока, время запуска и уровень приоритета. Обновим метод `Main()` в классе `Program` так, чтобы он запрашивал у пользователя идентификатор PID подлежащего исследованию процесса:

```
static void Main(string[] args)
{
  ...
  // Запросить у пользователя PID и вывести набор активных потоков.
  Console.WriteLine("***** Enter PID of process to investigate *****");
  Console.Write("PID: ");
  string pID = Console.ReadLine();
  int theProcID = int.Parse(pID);

  EnumThreadsForPid(theProcID);
  Console.ReadLine();
}
```

После запуска приложения можно вводить PID любого процесса на машине и просматривать имеющиеся внутри него потоки. В следующем выводе показаны потоки, используемые процессом с PID, равным 108, который (так случилось) обслуживает Microsoft Internet Explorer:

```
***** Enter PID of process to investigate *****
PID: 108
Here are the threads used by: iexplore
-> Thread ID: 680      Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 2040     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 880      Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 3380     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 3376     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 3448     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 3476     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 2264     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 2380     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 2384     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 2308     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 3096     Start Time: 9:07 AM     Priority: Highest
-> Thread ID: 3600     Start Time: 9:45 AM     Priority: Normal
-> Thread ID: 1412     Start Time: 10:02 AM    Priority: Normal
```

Помимо `Id`, `StartTime` и `PriorityLevel` тип `ProcessThread` содержит дополнительные члены, наиболее интересные из которых перечислены в табл. 17.4.

Прежде чем двигаться дальше, необходимо уяснить, что тип `ProcessThread` *не является* сущностью, применяемой для создания, приостановки или уничтожения потоков на платформе .NET. Этот тип скорее представляет собой средство, позволяющее получать диагностическую информацию по активным потокам Windows внутри выполняющегося процесса. Более подробные сведения о том, как создавать многопоточные приложения с использованием пространства имен `System.Threading`, приводятся в главе 19.

Таблица 17.4. Избранные члены типа `ProcessThread`

Член	Описание
`CurrentPriority`	Получает текущий приоритет потока
`Id`	Получает уникальный идентификатор потока
`IdealProcessor`	Устанавливает предпочитаемый процессор для выполнения заданного потока
`PriorityLevel`	Получает или устанавливает уровень приоритета потока
`ProcessorAffinity`	Устанавливает процессоры, на которых может выполняться связанный поток
`StartAddress`	Получает адрес памяти функции, вызванной операционной системой, которая запустила данный поток
`StartTime`	Получает время, когда операционная система запустила поток
`ThreadState`	Получает текущее состояние потока
`TotalProcessorTime`	Получает общее время, потраченное данным потоком на использование процессора
`WaitReason`	Получает причину, по которой поток находится в состоянии ожидания

Исследование набора модулей процесса

Теперь давайте посмотрим, как реализовать проход по загруженным модулям, которые размещены внутри конкретного процесса. Когда речь идет о процессах, *модуль* — это общий термин, применяемый для описания заданной сборки `*.dll` (или самого файла `*.exe`), которая обслуживается специфичным процессом. Когда производится доступ к коллекции `ProcessModuleCollection` через свойство `Process.Modules`, появляется возможность перечисления *всех модулей*, размещенных внутри процесса: библиотек на основе .NET, COM и традиционного C. Взгляните на показанный ниже дополнительный вспомогательный метод, который будет перечислять модули в процессе с указанным идентификатором PID:

```
static void EnumModsForPid(int pID)
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(pID);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }
  Console.WriteLine("Here are the loaded modules for: {0}",
    theProc.ProcessName);
  ProcessModuleCollection theMods = theProc.Modules;
  foreach(ProcessModule pm in theMods)
  {
    string info = string.Format("-> Mod Name: {0}", pm.ModuleName);
    Console.WriteLine(info);
  }
  Console.WriteLine("*************************************\n");
}
```

Чтобы получить какой-то вывод, давайте просмотрим загружаемые модули для процесса, обслуживающего программу текущего примера (ProcessManipulator). Для этого нужно запустить приложение, выяснить идентификатор PID, назначенный ProcessManipulator.exe (посредством диспетчера задач), и передать это значение методу EnumModsForPid() (соответствующим образом обновив метод Main()). Вас может удивить, что с простым консольным приложением связан настолько внушительный список библиотек *.dll (GDI32.dll, USER32.dll, ole32.dll и т.д.):

```
Here are the loaded modules for: ProcessManipulator
-> Mod Name: ProcessManipulator.exe
-> Mod Name: ntdll.dll
-> Mod Name: MSCOREE.DLL
-> Mod Name: KERNEL32.dll
-> Mod Name: KERNELBASE.dll
-> Mod Name: ADVAPI32.dll
-> Mod Name: msvcrt.dll
-> Mod Name: sechost.dll
-> Mod Name: RPCRT4.dll
-> Mod Name: SspiCli.dll
-> Mod Name: CRYPTBASE.dll
-> Mod Name: mscoreei.dll
-> Mod Name: SHLWAPI.dll
-> Mod Name: GDI32.dll
-> Mod Name: USER32.dll
-> Mod Name: LPK.dll
-> Mod Name: USP10.dll
-> Mod Name: IMM32.DLL
-> Mod Name: MSCTF.dll
-> Mod Name: clr.dll
-> Mod Name: MSVCR100_CLR0400.dll
-> Mod Name: mscorlib.ni.dll
-> Mod Name: nlssorting.dll
-> Mod Name: ole32.dll
-> Mod Name: clrjit.dll
-> Mod Name: System.ni.dll
-> Mod Name: System.Core.ni.dll
-> Mod Name: psapi.dll
-> Mod Name: shfolder.dll
-> Mod Name: SHELL32.dll
*********************************
```

Запуск и останов процессов программным образом

Финальными аспектами класса System.Diagnostics.Process, которые мы здесь исследуем, являются методы Start() и Kill(). Эти методы позволяют программно запускать и завершать процесс. Для примера создадим вспомогательный статический метод StartAndKillProcess() со следующим кодом.

На заметку! Для того чтобы запускать новые процессы, среда Visual Studio должна выполняться с правами администратора. В противном случае во время выполнения возникает ошибка.

```
static void StartAndKillProcess()
{
  Process ieProc = null;
  // Запустить Internet Explorer и перейти на сайт facebook.com.
```

```
    try
    {
      ieProc = Process.Start("IExplore.exe", "www.facebook.com");
    }
    catch (InvalidOperationException ex)
    {
      Console.WriteLine(ex.Message);
    }
    Console.Write("--> Hit enter to kill {0}...", ieProc.ProcessName);
    Console.ReadLine();
    // Уничтожить процесс iexplore.exe.
    try
    {
      ieProc.Kill();
    }
    catch (InvalidOperationException ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
```

Статический метод `Process.Start()` имеет несколько перегруженных версий. Как минимум, необходимо указывать дружественное имя запускаемого процесса (наподобие `iexplore.exe` для Microsoft Internet Explorer). В данном примере используется версия метода `Start()`, которая позволяет задавать любые дополнительные аргументы, подлежащие передаче в точку входа программы (т.е. методу `Main()`).

В результате вызова метода `Start()` возвращается ссылка на новый запущенный процесс. Чтобы завершить этот процесс, потребуется просто вызвать метод `Kill()` уровня экземпляра. Здесь вызовы `Start()` и `Kill()` помещены внутрь блока `try/catch` с обработкой любых исключений `InvalidOperationException`. Это особенно важно при вызове метода `Kill()`, потому что такое исключение генерируется, если процесс был завершен до вызова `Kill()`.

Управление запуском процесса с использованием класса `ProcessStartInfo`

Метод `Start()` позволяет также передавать объект типа `System.Diagnostics.ProcessStartInfo` для указания дополнительной информации относительно запуска определенного процесса. Ниже приведено частичное определение `ProcessStartInfo` (полное определение можно найти в документации .NET Framework 4.6 SDK):

```
public sealed class ProcessStartInfo : object
{
  public ProcessStartInfo();
  public ProcessStartInfo(string fileName);
  public ProcessStartInfo(string fileName, string arguments);
  public string Arguments { get; set; }
  public bool CreateNoWindow { get; set; }
  public StringDictionary EnvironmentVariables { get; }
  public bool ErrorDialog { get; set; }
  public IntPtr ErrorDialogParentHandle { get; set; }
  public string FileName { get; set; }
  public bool LoadUserProfile { get; set; }
  public SecureString Password { get; set; }
  public bool RedirectStandardError { get; set; }
```

```
  public bool RedirectStandardInput { get; set; }
  public bool RedirectStandardOutput { get; set; }
  public Encoding StandardErrorEncoding { get; set; }
  public Encoding StandardOutputEncoding { get; set; }
  public bool UseShellExecute { get; set; }
  public string Verb { get; set; }
  public string[] Verbs { get; }
  public ProcessWindowStyle WindowStyle { get; set; }
  public string WorkingDirectory { get; set; }
}
```

Чтобы проиллюстрировать настройку запуска процесса, модифицируем метод `StartAndKillProcess()` для загрузки браузера Microsoft Internet Explorer, перехода на сайт `www.facebook.com` и отображения окна браузера в развернутом на весь экран виде:

```
static void StartAndKillProcess()
{
  Process ieProc = null;

  // Запустить Internet Explorer и перейти на сайт facebook.com
  // с развернутым на весь экран окном.
  try
  {
    ProcessStartInfo startInfo = new
      ProcessStartInfo("IExplore.exe", "www.facebook.com");
    startInfo.WindowStyle = ProcessWindowStyle.Maximized;

    ieProc = Process.Start(startInfo);
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine(ex.Message);
  }
  ...
}
```

Теперь, когда вы понимаете роль процессов Windows и знаете способы взаимодействия с ними из кода C#, можно переходить к изучению концепции доменов приложений .NET.

Исходный код. Проект `ProcessManipulator` доступен в подкаталоге `Chapter_17`.

Домены приложений .NET

На платформе .NET исполняемые файлы не размещаются прямо внутри процесса Windows, как это происходит в случае традиционных неуправляемых приложений. Взамен исполняемый файл .NET попадает в отдельный логический раздел внутри процесса, который называется *доменом приложения*. Вы увидите, что один процесс может содержать несколько доменов приложений, каждый из которых обслуживает свой исполняемый файл .NET. Такое дополнительное разделение традиционного процесса Windows обеспечивает несколько преимуществ.

- Домены приложений являются ключевым аспектом нейтральной к операционным системам природы платформы .NET, поскольку такое логическое разделение абстрагирует отличия в том, как лежащая в основе операционная система представляет загруженный исполняемый файл.

- Домены приложений оказываются гораздо менее затратными в смысле вычислительных ресурсов и памяти по сравнению с полноценными процессами. Таким образом, среда CLR способна загружать и выгружать домены приложений намного быстрее, чем формальный процесс и может значительно улучшить масштабируемость серверных приложений.
- Домены приложений обеспечивают более глубокий уровень изоляции при размещении загруженных приложений. Если один домен приложения внутри процесса терпит отказ, то остальные домены приложений остаются работоспособными.

Как уже упоминалось, одиночный процесс может размещать любое количество доменов приложений, каждый из которых полностью изолирован от остальных доменов внутри данного (или любого другого) процесса. Учитывая этот факт, имейте в виду, что приложение, выполняющееся в одном домене приложения, не может получать данные любого рода (глобальные переменные или статические поля) из другого домена приложения, если только не применяется какой-нибудь протокол распределенного программирования (вроде Windows Communication Foundation).

Хотя в одном процессе *может* находиться множество доменов приложений, обычно подобное не происходит. Самое меньшее процесс операционной системы будет обслуживать так называемый *стандартный домен приложения*. Этот специфический домен приложения автоматически создается средой CLR во время запуска процесса. Затем среда CLR создает дополнительные домены приложений по мере необходимости.

Класс `System.AppDomain`

Платформа .NET позволяет программно отслеживать домены приложений, создавать новые домены приложений (или выгружать их) во время выполнения, загружать сборки в домены приложений и решать целый ряд других задач с использованием класса `AppDomain` из пространства имен `System`, которое находится в сборке `mscorlib.dll`. В табл. 17.5 описаны некоторые полезные методы этого класса (полная информация доступна в документации .NET Framework 4.6 SDK).

Таблица 17.5. Избранные методы класса `AppDomain`

Метод	Описание
`CreateDomain()`	Этот статический метод позволяет создавать новый домен приложения в текущем процессе
`CreateInstance()`	Этот метод позволяет создавать экземпляр типа из внешней сборки после загрузки данной сборки в вызывающий домен приложения
`ExecuteAssembly()`	Этот метод выполняет сборку `*.exe` внутри домена приложения, получив ее имя файла
`GetAssemblies()`	Этот метод получает набор сборок .NET, которые были загружены в данный домен приложения (двоичные сборки на основе COM и C++ игнорируются)
`GetCurrentThreadId()`	Этот статический метод возвращает идентификатор активного потока в текущем домене приложения
`Load()`	Этот метод применяется для динамической загрузки сборки в текущий домен приложения
`Unload()`	Этот статический метод позволяет выгрузить указанный домен приложения из заданного процесса

На заметку! Платформа .NET не позволяет выгружать конкретную сборку из памяти. Единственный способ программной выгрузки библиотек предусматривает уничтожение размещающего домена приложения посредством метода `Unload()`.

Вдобавок класс `AppDomain` определяет набор свойств, которые могут быть удобны при мониторинге действий заданного домена приложения. Наиболее интересные свойства описаны в табл. 17.6.

Таблица 17.6. Избранные свойства класса `AppDomain`

Свойство	Описание
`BaseDirectory`	Это свойство позволяет получить путь к каталогу, который распознаватель сборок использует для зондирования сборок
`CurrentDomain`	Это статическое свойство позволяет получить домен приложения, применяемый для текущего выполняющегося потока
`FriendlyName`	Это свойство позволяет получить дружественное имя текущего домена приложения
`MonitoringIsEnabled`	Это свойство позволяет получить или установить значение, которое указывает, включен ли мониторинг ресурсов центрального процессора и памяти, потребляемых доменами приложений, для текущего процесса. После того, как мониторинг для процесса включен, отключить его невозможно
`SetupInformation`	Это свойство позволяет получить детали конфигурации для указанного домена приложения, представленные в виде объекта `AppDomainSetup`

И, наконец, класс `AppDomain` поддерживает набор событий, которые соответствуют различным аспектам жизненного цикла домена приложения. Некоторые наиболее полезные события, доступные для привязки, представлены в табл. 17.7.

Таблица 17.7. Избранные события класса `AppDomain`

Событие	Описание
`AssemblyLoad`	Происходит, когда сборка загружается в память
`AssemblyResolve`	Возникает, когда распознаватель сборок не может найти местоположение обязательной сборки
`DomainUnload`	Происходит перед началом выгрузки домена приложения из обслуживающего процесса
`FirstChanceException`	Позволяет получать уведомление о том, что в домене приложения было сгенерировано исключение, перед тем как среда CLR начнет поиск подходящего оператора `catch`
`ProcessExit`	Возникает в стандартном домене приложения, когда его родительский процесс завершается
`UnhandledException`	Происходит, когда исключение не было перехвачено обработчиком исключений

Взаимодействие со стандартным доменом приложения

Вспомните, что после запуска исполняемого файла .NET среда CLR автоматически помещает его в стандартный домен приложения размещающего процесса. Все делается автоматически и прозрачно, и писать для этого какой-то специальный код не понадобится. Тем не менее, с помощью статического свойства `AppDomain.CurrentDomain` можно получать доступ к стандартному домену приложения. Получив такую точку доступа, появляется возможность привязываться к любым интересующим событиям либо использовать методы и свойства `AppDomain` для проведения диагностики во время выполнения.

Чтобы научиться взаимодействовать со стандартным доменом приложения, начнем с создания нового проекта консольного приложения по имени `DefaultAppDomainApp`. Модифицируем класс `Program`, поместив в него следующий код, который просто выводит детальные сведения о стандартном домене приложения с применением нескольких членов класса `AppDomain`:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with the default AppDomain *****\n");
    DisplayDADStats();
    Console.ReadLine();
  }

  private static void DisplayDADStats()
  {
    // Получить доступ к домену приложения для текущего потока.
    AppDomain defaultAD = AppDomain.CurrentDomain;

    // Вывести разнообразные статистические данные об этом домене.
    Console.WriteLine("Name of this domain: {0}", defaultAD.FriendlyName);
                      // Дружественное имя
    Console.WriteLine("ID of domain in this process: {0}", defaultAD.Id);
                       // Идентификатор
    Console.WriteLine("Is this the default domain?: {0}",
                      // Является ли стандартным
      defaultAD.IsDefaultAppDomain());
Console.WriteLine("Base directory of this domain: {0}", defaultAD.BaseDirectory);
                // Базовый каталог
  }
}
```

Ниже приведен вывод:

```
***** Fun with the default AppDomain *****

Name of this domain: DefaultAppDomainApp.exe
ID of domain in this process: 1
Is this the default domain?: True
Base directory of this domain: E:\MyCode\DefaultAppDomainApp\bin\Debug\
```

Обратите внимание, что имя стандартного домена приложения будет идентичным имени содержащегося внутри него исполняемого файла (`DefaultAppDomainApp.exe` в этом примере). Кроме того, значение базового каталога, которое будет использоваться для зондирования обязательных внешних закрытых сборок, отображается на текущее местоположение развернутого исполняемого файла.

Перечисление загруженных сборок

С применением метода `GetAssemblies()` уровня экземпляра можно просмотреть все сборки .NET, загруженные в указанный домен приложения. Метод возвращает массив объектов типа `Assembly`, который, как было показано в главе 15, является членом пространства имен `System.Reflection` (так что не забудьте импортировать это пространство имен в файл кода C#).

В целях иллюстрации определим в классе `Program` новый вспомогательный метод по имени `ListAllAssembliesInAppDomain()`. Он будет получать список всех загруженных сборок и выводить для каждой из них дружественное имя и номер версии:

```
static void ListAllAssembliesInAppDomain()
{
  // Получить доступ к домену приложения для текущего потока.
  AppDomain defaultAD = AppDomain.CurrentDomain;

  // Извлечь все сборки, загруженные в стандартный домен приложения.
  Assembly[] loadedAssemblies = defaultAD.GetAssemblies();
  Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
    defaultAD.FriendlyName);
  foreach(Assembly a in loadedAssemblies)
  {
    Console.WriteLine("-> Name: {0}", a.GetName().Name);          // Имя
    Console.WriteLine("-> Version: {0}\n", a.GetName().Version); // Версия
  }
}
```

Добавив в метод `Main()` вызов этого нового метода, можно увидеть все библиотеки .NET, которые используются в домене приложения, обслуживающем исполняемую сборку:

```
***** Here are the assemblies loaded in DefaultAppDomainApp.exe *****

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: DefaultAppDomainApp
-> Version: 1.0.0.0
```

Важно понимать, что список загруженных сборок может изменяться в любой момент по мере написания нового кода C#. Например, предположим, что метод `ListAllAssembliesInAppDomain()` модифицирован так, чтобы задействовать запрос LINQ, который упорядочивает загруженные сборки по имени:

```
static void ListAllAssembliesInAppDomain()
{
  // Получить доступ к домену приложения для текущего потока.
  AppDomain defaultAD = AppDomain.CurrentDomain;

  // Извлечь все сборки, загруженные в стандартный домен приложения.
  var loadedAssemblies = from a in defaultAD.GetAssemblies()
    orderby a.GetName().Name select a;

  Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
     defaultAD.FriendlyName);
  foreach (var a in loadedAssemblies)
  {
    Console.WriteLine("-> Name: {0}", a.GetName().Name);          // Имя
    Console.WriteLine("-> Version: {0}\n", a.GetName().Version); // Версия
  }
}
```

Запустив приложение еще раз, можно заметить, что в память также были загружены сборки `System.Core.dll` и `System.dll`, т.к. они требуются для API-интерфейса LINQ to Objects:

```
***** Here are the assemblies loaded in DefaultAppDomainApp.exe *****

-> Name: DefaultAppDomainApp
-> Version: 1.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: System
-> Version: 4.0.0.0

-> Name: System.Core
-> Version: 4.0.0.0
```

Получение уведомлений о загрузке сборок

Если вы хотите получать от среды CLR уведомление о загрузке новой сборки в заданный домен приложения, то должны обработать событие `AssemblyLoad`. Это событие имеет тип делегата `AssemblyLoadEventHandler`, который может указывать на любой метод, принимающий `System.Object` в первом параметре и `AssemblyLoadEventArgs` — во втором.

Давайте добавим в текущий класс `Program` последний метод по имени `InitDAD()`, который будет инициализировать стандартный домен приложения, обрабатывая событие `AssemblyLoad` посредством подходящего лямбда-выражения:

```
private static void InitDAD()
{
  // Эта логика будет выводить имя любой сборки,
  // загруженной в домен приложения после его создания.
  AppDomain defaultAD = AppDomain.CurrentDomain;
  defaultAD.AssemblyLoad += (o, s) =>
  {
    Console.WriteLine("{0} has been loaded!", s.LoadedAssembly.GetName().Name);
  };
}
```

После запуска модифицированного приложения при загрузке новой сборки будет отображаться соответствующее уведомление. Здесь просто выводится дружественное имя сборки с применением свойства `LoadedAssembly` входного параметра `AssemblyLoadedEventArgs`.

Исходный код. Проект `DefaultAppDomainApp` доступен в подкаталоге `Chapter_17`.

Создание новых доменов приложений

Вспомните, что единственный процесс способен размещать множество доменов приложений, создаваемых с помощью статического метода `AppDomain.CreateDomain()`. Хотя потребность в создании новых доменов приложений на лету в большинстве приложений .NET возникает довольно редко, важно понимать основы того, как это делается. Например, в главе 18 будет показано, что создаваемые *динамические сборки* должны устанавливаться в специальный домен приложения. Кроме того, многие API-интерфейсы, связанные с безопасностью .NET, требуют знания того, как конструировать новые домены приложений для изоляции сборок на основе предоставляемых учетных данных безопасности.

Чтобы посмотреть, каким образом конструировать новые домены приложений на лету (и загружать в них новые сборки), создадим новый проект консольного приложения по имени CustomAppDomains. Метод AppDomain.CreateDomain() имеет несколько перегруженных версий. Минимум придется указать дружественное имя создаваемого домена приложения. Ниже приведен код, который должен быть помещен в класс Program. Здесь используется метод ListAllAssembliesInAppDomain() из предыдущего примера, но на этот раз в качестве входного аргумента ему передается объект AppDomain, подлежащий анализу.

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Custom AppDomains *****\n");

    // Вывести все сборки, загруженные в стандартный домен приложения.
    AppDomain defaultAD = AppDomain.CurrentDomain;
    ListAllAssembliesInAppDomain(defaultAD);

    // Создать новый домен приложения.
    MakeNewAppDomain();
    Console.ReadLine();
  }

  private static void MakeNewAppDomain()
  {
    // Создать новый домен приложения в текущем процессе
    // и вывести список загруженных сборок.
    AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");
    ListAllAssembliesInAppDomain(newAD);
  }

  static void ListAllAssembliesInAppDomain(AppDomain ad)
  {
    // Получить все сборки, загруженные в стандартный домен приложения.
    var loadedAssemblies = from a in ad.GetAssemblies()
                           orderby a.GetName().Name select a;

    Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
      ad.FriendlyName);
    foreach (var a in loadedAssemblies)
    {
      Console.WriteLine("-> Name: {0}", a.GetName().Name);         // Имя
      Console.WriteLine("-> Version: {0}\n", a.GetName().Version); // Версия
    }
  }
}
```

Запустив приложение, вы увидите, что в стандартный домен приложения (CustomAppDomains.exe) были загружены сборки mscorlib.dll, System.dll, System.Core.dll и CustomAppDomains.exe, учитывая кодовую базу C# текущего проекта. Однако новый домен приложения содержит только сборку mscorlib.dll, которая, как вы помните, является одной из сборок .NET, всегда загружаемых средой CLR в любой домен приложения.

```
***** Fun with Custom AppDomains *****

***** Here are the assemblies loaded in CustomAppDomains.exe *****

-> Name: CustomAppDomains
```

```
-> Version: 1.0.0.0
-> Name: mscorlib
-> Version: 4.0.0.0
-> Name: System
-> Version: 4.0.0.0
-> Name: System.Core
-> Version: 4.0.0.0
***** Here are the assemblies loaded in SecondAppDomain *****
-> Name: mscorlib
-> Version: 4.0.0.0
```

На заметку! Если вы начнете отладку этого проекта (нажатием <F5>), то обнаружите, что во все домены приложений загружаются многие дополнительные сборки, которые задействованы процессом отладки Visual Studio. Запуск проект на выполнение (нажатием <Ctrl+F5>) приводит к отображению только сборок, напрямую загруженных в каждый домен приложения.

При наличии опыта построения традиционных приложений Windows такое поведение может показаться нелогичным (предполагается, что оба домена приложений имеют доступ к одному и тому же набору сборок). Тем не менее, вспомните, что сборка загружается в *домен приложения*, а не прямо в сам процесс.

Загрузка сборок в специальные домены приложений

Среда CLR будет загружать сборки в стандартный домен приложения, когда это требуется. Однако в случае создания новых доменов приложений вручную загружать сборки в указанный домен можно с помощью метода `AppDomain.Load()`. Кроме того, не следует забывать о возможности вызова метода `AppDomain.ExecuteAssembly()`, который позволяет загрузить сборку `*.exe` и выполнить метод `Main()`.

Предположим, что необходимо загрузить сборку `CarLibrary.dll` в новый вторичный домен приложения. При условии, что эта библиотека скопирована в папку `bin\Debug` текущего приложения, модифицируем метод `MakeNewAppDomain()` следующим образом (не забыв импортировать пространство имен `System.IO` для получения доступа к классу `FileNotFoundException`):

```
private static void MakeNewAppDomain()
{
  // Создать новый домен приложения в текущем процессе.
  AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");
  try
  {
    // Загрузить CarLibrary.dll в этот новый домен.
    newAD.Load("CarLibrary");
  }
  catch (FileNotFoundException ex)
  {
    Console.WriteLine(ex.Message);
  }
  // Вывести список всех сборок.
  ListAllAssembliesInAppDomain(newAD);
}
```

На этот раз вывод приложения выглядит так (обратите внимание на присутствие сборки `CarLibrary.dll`):

```
***** Fun with Custom AppDomains *****

***** Here are the assemblies loaded in CustomAppDomains.exe *****

-> Name: CustomAppDomains
-> Version: 1.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: System
-> Version: 4.0.0.0

-> Name: System.Core
-> Version: 4.0.0.0

***** Here are the assemblies loaded in SecondAppDomain *****

-> Name: CarLibrary
-> Version: 2.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0
```

На заметку! Не следует забывать, что во время отладки этого приложения в каждый домен приложения будут загружаться многие дополнительные библиотеки.

Выгрузка доменов приложений программным образом

Важно отметить, что среда CLR не разрешает выгружать индивидуальные сборки .NET. Тем не менее, с помощью метода `AppDomain.Unload()` можно избирательно выгрузить заданный домен приложения из размещающего процесса. В таком случае вместе с доменом приложения по очереди будут выгружаться содержащиеся в нем сборки.

Вспомните, что в типе `AppDomain` определено событие `DomainUnload`, которое инициируется при выгрузке специального домена приложения из содержащего его процесса. Еще одним интересным событием является `ProcessExit`, которое возникает, когда стандартный домен приложения выгружается из процесса (что вполне очевидно влечет за собой завершение самого процесса).

Чтобы реализовать программную выгрузку домена `newAD` из размещающего процесса с получением уведомления об его уничтожении, добавим в метод `MakeNewAppDomain()` приведенную далее логику:

```
private static void MakeNewAppDomain()
{
  // Создать новый домен приложения в текущем процессе.
  AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");
  newAD.DomainUnload += (o, s) =>
  {
    Console.WriteLine("The second AppDomain has been unloaded!");
  };
  try
  {
    // Загрузить CarLibrary.dll в этот новый домен.
    newAD.Load("CarLibrary");
  }
  catch (FileNotFoundException ex)
  {
    Console.WriteLine(ex.Message);
  }
```

```
  // Вывести список всех сборок.
  ListAllAssembliesInAppDomain(newAD);

  // Уничтожить этот домен приложения.
  AppDomain.Unload(newAD);
}
```

Если нужно обеспечить уведомление при выгрузке стандартного домена приложения, то метод `Main()` понадобится модифицировать для обработки события `ProcessEvent` упомянутого домена:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom AppDomains *****\n");

  // Вывести все сборки, загруженные в стандартный домен приложения.
  AppDomain defaultAD = AppDomain.CurrentDomain;
  defaultAD.ProcessExit += (o, s) =>
  {
    Console.WriteLine("Default AD unloaded!");
  };

  ListAllAssembliesInAppDomain(defaultAD);

  MakeNewAppDomain();
  Console.ReadLine();
}
```

На этом рассмотрение доменов приложений .NET завершено. Напоследок мы рассмотрим еще один уровень разделения, который применяется для группирования объектов в контекстные границы.

Исходный код. Проект `CustomAppDomains` доступен в подкаталоге `Chapter_17`.

Контекстные границы объектов

Как было только что показано, домены приложений — это логические разделы внутри процесса, используемого для размещения сборок .NET. Однако каждый домен приложения может быть дополнительно разделен на многочисленные контекстные границы. По существу контекст .NET предоставляет одиночному домену приложения возможность установки "специфического местоположения" для заданного объекта.

На заметку! В то время как понимание концепции процессов и доменов приложений довольно важно, в большинстве приложений .NET работа с объектными контекстами не требуется. Обзорный материал по объектным контекстам здесь включен лишь для предоставления более полной картины.

Применяя контекст, среда CLR способна обеспечить надлежащую и согласованную обработку объектов, которые имеют специальные требования времени выполнения, за счет перехвата обращений к методам в и из конкретного контекста. Этот уровень перехвата позволяет среде CLR подстраивать текущий вызов метода так, чтобы он соответствовал контекстным настройкам заданного объекта. Например, если определен класс C#, который требует автоматической безопасности к потокам (с использованием атрибута `[Synchronization]`), то во время выделения памяти для его экземпляра среда CLR будет создавать "синхронизированный контекст".

Подобно тому, как процесс определяет стандартный домен приложения, каждый домен приложения имеет стандартный контекст. Этот стандартный контекст (иногда называемый *контекстом 0*, т.к. в домене приложения он всегда создается первым) применяется для группирования вместе объектов .NET, которые не имеют никаких специфических или уникальных контекстных потребностей. Как и можно было ожидать, подавляющее большинство объектов .NET загружается именно в контекст 0. Если среда CLR определяет, что вновь созданный объект предъявляет специальные требования, она создает внутри размещающего домена приложения новую контекстную границу. На рис. 17.3 показаны отношения между процессом, доменом приложения и контекстом.

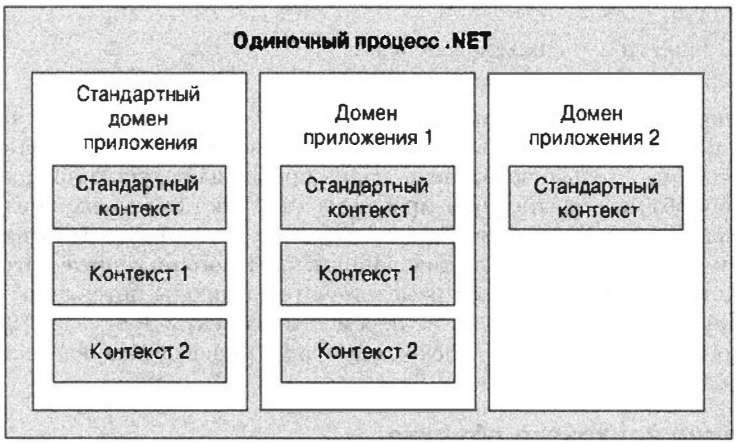


Рис. 17.3. Процессы, домены приложений и контекстные границы

Контекстно-свободные и контекстно-связанные типы

Объекты .NET, которые не требуют какой-то особой контекстной трактовки, называются *контекстно-свободными*. Такие объекты могут быть доступны из любого места внутри размещающего домена приложения без вмешательства со стороны требований времени выполнения, связанных с объектами. Контекстно-свободные объекты строятся легко, потому что для этого просто ничего не нужно делать (в частности, не декорировать тип контекстными атрибутами и не порождать его от базового класса `System.ContextBoundObject`). Вот пример:

```
// Контекстно-свободный объект загружается в контекст 0.
class SportsCar{}
```

С другой стороны, объекты, которые действительно требуют выделения контекста, называются *контекстно-связанными* и *должны* быть производными от базового класса `System.ContextBoundObject`. Этот базовый класс закрепляет тот факт, что интересующий объект может функционировать надлежащим образом только внутри контекста, в котором он был создан. Учитывая роль контекста .NET, должно быть ясно, что в случае попадания контекстно-связанного объекта по какой-то причине в несовместимый контекст обязательно произойдет что-то неприемлемое, причем в самый неподходящий момент.

Вдобавок к порождению от `System.ContextBoundObject` контекстно-связанный тип будет также декорирован атрибутами .NET специальной категории, которая называется *контекстными атрибутами*. Все контекстные атрибуты являются производными от базового класса `ContextAttribute`. Давайте рассмотрим пример.

Определение контекстно-связанного объекта

Предположим, что необходимо определить класс (SportsCarTS), который по своей природе автоматически является безопасным к потокам, даже если внутри реализации его членов не была жестко закодирована логика синхронизации потоков. Для этого класс SportsCarTS нужно унаследовать от ContextBoundObject и применить к нему атрибут [Synchronization]:

```
using System.Runtime.Remoting.Contexts;

// Этот контекстно-связанный тип будет загружаться только в синхронизированный
// (следовательно, безопасный к потокам) контекст.
[Synchronization]
class SportsCarTS : ContextBoundObject
{}
```

Типы, оснащенные атрибутом [Synchronization], загружаются в безопасный к потокам контекст. Учитывая специальные контекстные потребности класса MyThreadSafeObject, только вообразите, какие проблемы возникли бы в случае переноса созданного объекта из синхронизированного контекста в несинхронизированный. Объект неожиданно перестает быть безопасным к потокам и соответственно становится кандидатом на массовое повреждение данных, т.к. многочисленные потоки пытаются взаимодействовать с (теперь уже изменчивым со стороны потоков) объектом. Для гарантирования того, что среда CLR не переместит объекты SportsCarTS за пределы синхронизированного контекста, необходимо просто унаследовать класс SportsCarTS от ContextBoundObject.

Исследование контекста объекта

Хотя необходимость в программном взаимодействии с контекстом будет возникать лишь в немногих разрабатываемых вами приложениях, рассмотрим иллюстративный пример. Создадим новый проект консольного приложения по имени ObjectContextApp и определим в нем контекстно-свободный класс SportsCar, а также контекстно-связанный класс SportsCarTS:

```
using System;
using System.Runtime.Remoting.Contexts;     // Для типа Context.
using System.Threading;                     // Для типа Thread.

// Класс SportsCar не имеет никаких специальных
// контекстных потребностей и будет загружаться
// в стандартный контекст домена приложения.
class SportsCar
{
  public SportsCar()
  {
    // Получить информацию о контексте и вывести идентификатор контекста.
    Context ctx = Thread.CurrentContext;
    Console.WriteLine("{0} object in context {1}",
      this.ToString(), ctx.ContextID);
    foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
      Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
  }
}

// SportsCarTS требует загрузки в синхронизированный контекст.
[Synchronization]
```

```
class SportsCarTS : ContextBoundObject
{
  public SportsCarTS()
  {
    // Получить информацию о контексте и вывести идентификатор контекста.
    Context ctx = Thread.CurrentContext;
    Console.WriteLine("{0} object in context {1}",
      this.ToString(), ctx.ContextID);
    foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
      Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
  }
}
```

Обратите внимание, что каждый конструктор получает объект `Context` из текущего потока выполнения с помощью статического свойства `Thread.CurrentContext`. Имея объект `Context`, можно вывести статистические данные о контекстной границе, такие как назначенный идентификатор и набор дескрипторов, полученных через свойство `Context.ContextProperties`. Это свойство возвращает массив объектов, реализующих интерфейс `IContextProperty`, который открывает доступ к каждому дескриптору через свойство `Name`. Добавим в метод `Main()` код для размещения экземпляра каждого из этих классов в памяти:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Context *****\n");

  // Объекты при создании будут отображать контекстную информацию.
  SportsCar sport = new SportsCar();
  Console.WriteLine();

  SportsCar sport2 = new SportsCar();
  Console.WriteLine();

  SportsCarTS synchroSport = new SportsCarTS();
  Console.ReadLine();
}
```

По мере создания объектов конструкторы классов будут выводить различные фрагменты контекстной информации (выводимое свойство `LeaseLifeTimeServiceProperty` представляет собой низкоуровневый аспект уровня удаленной обработки .NET и может быть проигнорировано):

```
***** Fun with Object Context *****

ObjectContextApp.SportsCar object in context 0
-> Ctx Prop: LeaseLifeTimeServiceProperty

ObjectContextApp.SportsCar object in context 0
-> Ctx Prop: LeaseLifeTimeServiceProperty

ObjectContextApp.SportsCarTS object in context 1
-> Ctx Prop: LeaseLifeTimeServiceProperty
-> Ctx Prop: Synchronization
```

Поскольку класс `SportsCar` не был снабжен атрибутом контекста, среда CLR разместила объекты `sport` и `sport2` в контексте 0 (т.е. в стандартном контексте). Тем не менее, объект `SportsCarTS` загрузился в уникальную контекстную границу (которой был назначен идентификатор контекста, равный 1) с учетом того, что данный контекстно-связанный тип был декорирован атрибутом `[Synchronization]`.

Исходный код. Проект `ObjectContextApp` доступен в подкаталоге `Chapter_17`.

Итоговые сведения о процессах, доменах приложений и контекстах

К этому времени вы должны иметь намного лучшее представление о том, как сборка .NET размещается средой CLR. Ниже перечислены основные моменты.

- *Процесс .NET размещает один и более доменов приложений.* Каждый домен приложения способен размещать любое количество связанных сборок .NET. Домены приложений могут независимо загружаться и выгружаться средой CLR (или программно с помощью типа `System.AppDomain`).
- *Домен приложения может состоять из одного и более контекстов.* Используя контекст, среда CLR имеет возможность помещать объект с "особыми потребностями" в логический контейнер, чтобы обеспечить удовлетворение его требований времени выполнения.

Если изложенный материал показался слишком низкоуровневым, не переживайте. По большей части среда CLR самостоятельно занимается всеми деталями процессов, доменов приложений и контекстов. Однако эта информация формирует хороший фундамент для понимания многопоточного программирования на платформе .NET.

Резюме

Задачей главы было исследование особенностей размещения образа исполняемой сборки .NET платформой .NET. Как вы видели, давно существующее понятие процесса Windows было внутренне изменено и адаптировано под потребности CLR. Одиночный процесс (которым можно программно манипулировать посредством типа `System.Diagnostics.Process`) теперь состоит из одного или большего числа доменов приложений, которые представляют изолированные и независимые границы внутри процесса.

Вы узнали, что одиночный процесс может обслуживать несколько доменов приложений, каждый из которых способен размещать и выполнять любое количество связанных сборок. Кроме того, один домен приложения может содержать любое число контекстных границ. Благодаря такому дополнительному уровню изоляции типов среда CLR обеспечивает надлежащую обработку объектов с особыми потребностями во время выполнения.

ГЛАВА 18

Язык CIL и роль динамических сборок

При построении полномасштабного приложения .NET вы почти наверняка будете использовать C# (или другой управляемый язык, такой как Visual Basic) из-за присущей ему продуктивности и простоты применения. Однако в первой главе было показано, что роль управляемого компилятора заключается в трансляции файлов кода `*.cs` в код CIL, метаданные типов и манифест сборки. Как выяснилось, CIL представляет собой полноценный язык программирования .NET, который имеет собственный синтаксис, семантику и компилятор (`ilasm.exe`).

В настоящей главе будет предложен краткий экскурс по этому родному языку платформы .NET. Здесь вы узнаете о различиях между *директивой*, *атрибутом* и *кодом операции* CIL. Затем вы ознакомитесь с ролью возвратного проектирования сборок .NET и разнообразных инструментов программирования на CIL. Остаток главы посвящен основам определения пространств имен, типов и членов с использованием грамматики CIL. В завершение главы будет исследоваться роль пространства имен `System.Reflection.Emit` и возможность динамического конструирования сборок (с помощью инструкций CIL) во время выполнения.

Конечно, необходимость работать с низкоуровневым кодом CIL на повседневной основе будет возникать только у очень немногих программистов. Глава начинается с описания причин, по которым изучение синтаксиса и семантики этого языка .NET может оказаться полезным.

Причины для изучения грамматики языка CIL

Язык CIL является подлинным родным языком платформы .NET. При построении сборки .NET с помощью выбранного управляемого языка (C#, VB, F# и т.д.) соответствующий компилятор транслирует исходный код в инструкции CIL. Подобно любому языку программирования CIL предоставляет многочисленные лексемы, связанные со структурированием и реализацией. Поскольку CIL представляет собой просто еще один язык программирования .NET, не должно вызывать удивление то, что сборки .NET можно создавать прямо на CIL и компилировать их с помощью компилятора CIL (`ilasm.exe`), который поставляется в составе .NET Framework SDK.

Хотя трудно оспорить тот факт, что на построение завершенного приложения .NET непосредственно на CIL решаются лишь немногие программисты (если вообще такие находятся), изучение этого языка все равно является чрезвычайно интересным занятием. Попросту говоря, чем лучше вы понимаете грамматику CIL, тем больше способны погрузиться в мир расширенной разработки приложений .NET. Обратившись к конкрет-

ным примерам, можно утверждать, что разработчики, разбирающиеся в CIL, обладают следующими навыками.

- Умеют дизассемблировать существующую сборку .NET, редактировать код CIL в ней и заново компилировать обновленную кодовую базу в модифицированный двоичный файл .NET. Скажем, некоторые сценарии могут требовать модификации кода CIL для взаимодействия с расширенными средствами COM.
- Умеют строить динамические сборки с применением пространства имен `System.Reflection.Emit`. Данный API-интерфейс позволяет генерировать в памяти сборку .NET, которая дополнительно может быть сохранена на диск. Это полезный прием для разработчиков инструментов, которым необходимо генерировать сборки на лету.
- Понимают аспекты CTS, которые не поддерживаются высокоуровневыми управляемыми языками, но существуют на уровне CIL. На самом деле CIL является единственным языком .NET, который позволяет получать доступ ко всем аспектам CTS. Например, за счет использования низкоуровневого кода CIL появляется возможность определения членов и полей глобального уровня (которые не разрешены в C#).

Ради полной ясности нужно еще раз подчеркнуть, что овладеть мастерством работы с языком C# и библиотеками базовых классов .NET можно и *без* изучения деталей кода CIL. Во многих отношениях знание CIL аналогично знанию языка ассемблера программистом на C (C++). Те, кто разбирается в низкоуровневых деталях, способны создавать более совершенные решения поставленных задач и глубже понимают лежащую в основе среду программирования (и выполнения). Таким образом, если вы готовы принять вызов, давайте приступим к исследованию внутренних деталей CIL.

На заметку! Имейте в виду, что настоящая глава не планировалась быть всеобъемлющим руководством по синтаксису и семантике CIL. Если вам требуется полное изложение этой темы, рекомендуется загрузить официальную спецификацию ECMA (`ecma-335.pdf`) из веб-сайта ECMA International (`www.ecma-international.org`).

Директивы, атрибуты и коды операций CIL

Когда вы начинаете изучение низкоуровневых языков, таких как CIL, то гарантированно встретите новые (и часто пугающие) названия для знакомых концепций. Например, к этому моменту приведенный ниже набор элементов вы почти наверняка посчитаете ключевыми словами языка C# (и это правильно):

```
{new, public, this, base, get, set, explicit, unsafe, enum, operator, partial}
```

Тем не менее, внимательнее присмотревшись к элементам этого набора, вы сможете заметить, что в то время как каждый из них действительно является ключевым словом C#, он имеет радикально отличающуюся семантику. Скажем, ключевое слово `enum` определяет производный от `System.Enum` тип, а ключевые слова `this` и `base` позволяют ссылаться на текущий объект и его родительский класс. Ключевое слово `unsafe` применяется для установления блока кода, который не может напрямую отслеживаться средой CLR, а ключевое слово `operator` дает возможность создать скрытый (специально именованный) метод, который будет вызываться, когда используется специфическая операция C# (такая как знак "плюс").

По разительному контрасту с высокоуровневым языком вроде C# в CIL не просто определяется общий набор ключевых слов сам по себе. Взамен набор лексем, распозна-

ваемых компилятором CIL, на основе их семантики подразделяется на следующие три обширные категории:

- директивы CIL;
- атрибуты CIL;
- коды операций CIL.

Лексемы CIL каждой категории выражаются с применением отдельного синтаксиса и комбинируются для построения допустимой сборки .NET.

Роль директив CIL

Прежде всего, существует набор хорошо известных лексем CIL, которые используются для описания общей структуры сборки .NET. Эти лексемы называются *директивами*. Директивы CIL позволяют информировать компилятор CIL о том, каким образом определять пространства имен, типы и члены, которые будут заполнять сборку.

Синтаксически директивы представляются с применением префикса в виде точки (.), например, `.namespace`, `.class`, `.publickeytoken`, `.override`, `.method`, `.assembly` и т.д. Таким образом, если в файле с расширением `*.il` (общепринятое расширение для файлов кода CIL) указана одна директива `.namespace` и три директивы `.class`, то компилятор CIL сгенерирует сборку, в которой определено единственное пространство имен, содержащее три класса .NET.

Роль атрибутов CIL

Во многих случаях директивы CIL сами по себе недостаточно описательны для того, чтобы полностью выразить определение заданного типа .NET или члена типа. С учетом этого факта многие директивы CIL могут сопровождаться разнообразными *атрибутами* CIL, которые уточняют способ обработки директивы. Например, директива `.class` может быть снабжена атрибутом `public` (для установления видимости типа), атрибутом `extends` (для явного указания базового класса типа) и атрибутом `implements` (для перечисления набора интерфейсов, поддерживаемых данным типом).

На заметку! Не путайте атрибут .NET (глава 15) и атрибут CIL, которые представляют собой два совершенно разных понятия.

Роль кодов операций CIL

После того как сборка .NET, пространство имен и набор типов определены в терминах языка CIL с использованием различных директив и связанных атрибутов, остается только предоставить логику реализации для типов. Это работа *кодов операций*. В традициях других низкоуровневых языков программирования многие коды операций CIL обычно имеют непонятный и совершенно нечитабельный вид. Например, для загрузки в память переменной `string` применяется код операции, который имеет не дружественное имя наподобие `LoadString`, а выглядит как `ldstr`.

Справедливости ради следует отметить, что некоторые коды операций CIL довольно естественно отображаются на свои аналоги в C# (например, `box`, `unbox`, `throw` и `sizeof`). Вы увидите, что коды операций CIL всегда используются внутри области реализации члена и в отличие от директив никогда не записываются с префиксом-точкой.

Разница между кодами операций и их мнемоническими эквивалентами в CIL

Как только что объяснялось, для реализации членов отдельно взятого типа применяются коды операций наподобие `ldstr`. Однако на самом деле такие лексемы, как `ldstr`, являются *мнемоническими эквивалентами CIL* действительных *двоичных кодов операций CIL*. Чтобы прояснить разницу, напишем следующий метод C#:

```
static int Add(int x, int y)
{
  return x + y;
}
```

Действие сложения двух чисел в терминах CIL выражается посредством кода операции `0X58`. В том же духе вычитание двух чисел выражается с помощью кода операции `0X59`, а действие по размещению нового объекта в управляемой куче записывается с использованием кода операции `0X73`. С учетом такой реальности "код CIL", обрабатываемый JIT-компилятором, в действительности представляет собой не более чем порцию двоичных данных.

К счастью, для каждого двоичного кода операции CIL имеется соответствующий мнемонический эквивалент. Например, вместо кода `0X58` может применяться мнемонический эквивалент `add`, вместо `0X59` — эквивалент `sub`, а вместо `0X73` — `newobj`. При такой разнице между кодами операций и их мнемоническими эквивалентами декомпиляторы CIL, такие как `ildasm.exe`, транслируют двоичные коды операций сборки в соответствующие им мнемонические эквиваленты CIL. Вот как `ildasm.exe` представит в CIL предыдущий метод `Add()`, написанный на языке C# (в зависимости от версии .NET вывод может отличаться):

```
.method private hidebysig static int32 Add(int32 x,
  int32 y) cil managed
{
  // Code size 9 (0x9)
  .maxstack 2
  .locals init ([0] int32 CS$1$0000)
  IL_0000: nop
  IL_0001: ldarg.0
  IL_0002: ldarg.1
  IL_0003: add
  IL_0004: stloc.0
  IL_0005: br.s IL_0007
  IL_0007: ldloc.0
  IL_0008: ret
}
```

Если вы не занимаетесь разработкой исключительно низкоуровневого программного обеспечения .NET (вроде специального управляемого компилятора), то иметь дело с числовыми двоичными кодами операций CIL никогда не придется. В практическом смысле, когда программисты .NET говорят о "кодах операций CIL", они имеют в виду набор дружественных строковых мнемонических эквивалентов (что и делается в настоящей книге), а не лежащие в основе числовые значения.

Заталкивание и выталкивание: основанная на стеке природа CIL

В языках .NET высокого уровня (таких как C#) предпринимается попытка скрыть из виду низкоуровневые детали CIL настолько, насколько это возможно. Один из особенно хорошо скрываемых аспектов — тот факт, что CIL является языком программирования, основанным на использовании стека. Вспомните из исследования пространств имен коллекций (см. главу 9), что класс `Stack<T>` может применяться для помещения значения в стек, а также для извлечения самого верхнего значения из стека с целью последующего использования. Разумеется, разработчики на CIL не работают с объектом типа `Stack<T>` для загрузки и выгрузки вычисляемых значений, но применяют образ действий, похожий на заталкивание и выталкивание.

Формально сущность, используемая для хранения набора вычисляемых значений, называется *виртуальным стеком выполнения*. Вы увидите, что CIL предоставляет несколько кодов операций, которые служат для помещения значения в стек; такой процесс именуется *загрузкой*. Кроме того, в CIL определен набор дополнительных кодов операций, которые перемещают самое верхнее значение из стека в память (скажем, в локальную переменную), применяя процесс под названием *сохранение*.

В мире CIL невозможно напрямую получать доступ к элементам данных, включая локально определенные переменные, входные аргументы методов и данные полей типа. Вместо этого элемент данных должен быть явно загружен в стек и затем извлекаться оттуда для использования в более позднее время (запомните упомянутое требование, поскольку это поможет понять, почему блок кода CIL может выглядеть несколько избыточным).

На заметку! Вспомните, что код CIL не выполняется напрямую, а компилируется по требованию. Во время компиляции кода CIL многие избыточные аспекты реализации оптимизируются. Более того, если для текущего проекта включена оптимизация кода (на вкладке Build (Сборка) окна свойств проекта в Visual Studio), то компилятор будет также удалять разнообразные избыточные детали CIL.

Чтобы понять, каким образом CIL задействует модель обработки на основе стека, создадим простой метод C# по имени `PrintMessage()`, который не принимает аргументов и возвращает `void`. Внутри его реализации будет просто выводиться в стандартный выходной поток значение локальной переменной:

```
public void PrintMessage()
{
  string myMessage = "Hello.";
  Console.WriteLine(myMessage);
}
```

Если просмотреть код CIL, который получился в результате трансляции этого метода компилятором C#, то первым делом обнаружится, что метод `PrintMessage()` определяет ячейку памяти для локальной переменной с помощью директивы `.locals`. Затем локальная строка загружается и сохраняется в этой локальной переменной с применением кодов операций `ldstr` (загрузить строку) и `stloc.0` (сохранить текущее значение в локальной переменной, находящейся в ячейке 0).

Далее с помощью кода операции `ldloc.0` (загрузить локальный аргумент по индексу 0) значение (по индексу 0) загружается в память для использования в вызове метода `System.Console.WriteLine()` (представленного кодом операции `call`). Наконец, посредством кода операции `ret` производится возвращение из функции. Ниже показан

(прокомментированный) код CIL для метода `PrintMessage()` (ради краткости из листинга были удалены коды операций `nop`):

```
.method public hidebysig instance void PrintMessage() cil managed
{
  .maxstack 1
  // Определить локальную переменную типа string (по индексу 0).
  .locals init ([0] string myMessage)

  // Загрузить в стек строку со значением "Hello.".
  ldstr "Hello."

  // Сохранить строковое значение из стека в локальной переменной.
  stloc.0

  // Загрузить значение по индексу 0.
  ldloc.0

  // Вызвать метод с текущим значением.
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

На заметку! Как видите, язык CIL поддерживает синтаксис комментариев в виде двойной косой черты (а также синтаксис `/*...*/`, если уж на то пошло). Как и в C#, комментарии в коде компилятор CIL игнорирует.

Теперь, когда вы знаете основы директив, атрибутов и кодов операций CIL, давайте приступим к практическому программированию на CIL, начав с рассмотрения темы возвратного проектирования.

Возвратное проектирование

В главе 1 было показано, как применять утилиту `ildasm.exe` для просмотра кода CIL, сгенерированного компилятором C#. Тем не менее, вы можете даже не подозревать, что эта утилита позволяет сбрасывать код CIL, содержащийся внутри загруженной в нее сборки, во внешний файл. Полученный подобным образом код CIL можно редактировать и компилировать заново с помощью компилятора CIL (`ilasm.exe`).

Выражаясь формально, такой прием называется *возвратным проектированием* и может быть полезен в избранных обстоятельствах вроде перечисленных ниже.

- Вам необходимо модифицировать сборку, исходный код которой больше не доступен.
- Вы работаете с далеким от идеала компилятором языка .NET, который генерирует неэффективный (или явно некорректный) код CIL, поэтому нужно изменить кодовую базу.
- Вы конструируете библиотеку взаимодействия с COM и хотите учесть ряд атрибутов COM IDL, которые были утеряны во время процесса преобразования (такие как COM-атрибут `[helpstring]`).

Чтобы продемонстрировать процесс возвратного проектирования, создадим в простом текстовом редакторе новый файл кода C# (`HelloProgram.cs`) и определим в нем следующий тип класса (при желании можете создать новый проект консольного приложения в Visual Studio, но только удалите файл `AssemblyInfo.cs` для уменьшения объема генерируемого кода CIL):

```
// Простое консольное приложение C#.
using System;

// Обратите внимание, что с целью упрощения генерируемого
// кода CIL класс не помещается в пространство имен.
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Hello CIL code!");
    Console.ReadLine();
  }
}
```

Сохраним этот файл в удобном месте (например, `C:\RoundTrip`) и скомпилируем программу, используя `csc.exe`:

```
csc HelloProgram.cs
```

Теперь откроем файл `HelloProgram.exe` в `ildasm.exe` и путем выбора пункта меню File⇨Dump (Файл⇨Сбросить) сохраним низкоуровневый код CIL в новый файл `*.il` (`HelloProgram.il`) внутри той же папки, в которой находится скомпилированная сборка (оставив стандартные настройки в диалоговом окне сохранения неизменными).

На заметку! Во время сбрасывания содержимого сборки в файл утилита `ildasm.exe` также генерирует файл `*.res`. Такие ресурсные файлы можно игнорировать (и удалять), поскольку в этой главе они не применяются. В них содержится низкоуровневая информация, касающаяся безопасности CLR (помимо прочих данных).

Теперь можно просмотреть файл `HelloProgram.il` в любом текстовом редакторе. Вот его содержимое (для удобства оно слегка переформатировано и снабжено комментариями):

```
// Ссылаемые сборки.
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}

// Наша сборка.
.assembly HelloProgram
{
  /**** Ради ясности данные TargetFrameworkAttribute удалены! ****/

  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module HelloProgram.exe
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003
.corflags 0x00000003

// Определение класса Program.
.class private auto ansi beforefieldinit Program
  extends [mscorlib]System.Object
{
```

```
  .method private hidebysig static void Main(string[] args) cil managed
  {
    // Помечает этот метод как точку входа исполняемой сборки.
    .entrypoint
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr "Hello CIL code!"
    IL_0006: call void [mscorlib]System.Console::WriteLine(string)
    IL_000b: nop
    IL_000c: call string [mscorlib]System.Console::ReadLine()
    IL_0011: pop
    IL_0012: ret
  }
  // Стандартный конструктор.
  .method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
  {
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call instance void [mscorlib]System.Object::.ctor()
    IL_0006: ret
  }
}
```

Обратите внимание, что файл `*.il` начинается с объявления всех внешних сборок, на которые ссылается текущая скомпилированная сборка. Здесь имеется единственная лексема `.assembly extern`, установленная для постоянно присутствующей сборки `mscorlib.dll`. Конечно, если бы в библиотеке классов использовались типы из других сборок, то были бы определены дополнительные директивы `.assembly extern`.

Далее обнаруживается формальное определение сборки `HelloProgram.exe`, которой был назначен стандартный номер версии `0.0.0.0` (потому что не было указано какое-либо значение с помощью атрибута `[AssemblyVersion]`). Вдобавок сборка описана с применением разнообразных директив CIL (таких как `.module`, `.imagebase` и т.д.).

После документирования внешних сборок и определения текущей сборки находится определение типа `Program`. Обратите внимание, что директива `.class` имеет различные атрибуты (многие из которых на самом деле необязательны) вроде показанного ниже атрибута `extends`, который указывает базовый класс для типа:

```
.class private auto ansi beforefieldinit Program
  extends [mscorlib]System.Object
{ ... }
```

Основной объем кода CIL представляет реализацию стандартного конструктора класса и метода `Main()`, которые оба определены (частично) посредством директивы `.method`. После того, как эти члены были определены с использованием корректных директив и атрибутов, они реализуются с применением разнообразных кодов операций.

Важно понимать, что при взаимодействии с типами .NET (такими как `System.Console`) в CIL *всегда* необходимо использовать полностью заданное имя типа. Более того, полностью заданное имя типа *всегда* должно предваряться префиксом в форме дружественного имени сборки, в которой тип определен (в квадратных скобках). Взгляните на следующую реализацию метода `Main()` в CIL:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
```

```
  .maxstack 8
  IL_0000: nop
  IL_0001: ldstr "Hello CIL code!"
  IL_0006: call void [mscorlib]System.Console::WriteLine(string)
  IL_000b: nop
  IL_000c: call string [mscorlib]System.Console::ReadLine()
  IL_0011: pop
  IL_0012: ret
}
```

В реализации стандартного конструктора на языке CIL применяется еще одна инструкция, связанная с загрузкой — `ldarg.0`. В этом случае значение, загружаемое в стек, представляет собой не специальную переменную, указанную вами, а ссылку на текущий объект (более подробно об этом речь пойдет позже). Также обратите внимание, что в стандартном конструкторе явно производится вызов конструктора базового класса, которым в данном случае является хорошо знакомый класс `System.Object`:

```
.method public hidebysig specialname rtspecialname
  instance void .ctor() cil managed
{
  .maxstack 8
  IL_0000: ldarg.0
  IL_0001: call instance void [mscorlib]System.Object::.ctor()
  IL_0006: ret
}
```

Роль меток в коде CIL

Вы определенно заметили, что каждая строка в коде реализации предваряется лексемой в форме `IL_XXX:` (например, `IL_0000:`, `IL_0001:` и т.д.). Такие лексемы называются *метками кода* и могут именоваться в любой выбранной вами манере (при условии, что они не дублируются внутри области действия члена). При сбросе содержимого сборки в файл утилита `ildasm.exe` автоматически генерирует метки кода, которые следуют соглашению об именовании вида `IL_XXX:`. Однако их можно заменить более описательными маркерами, например:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack 8
  Nothing_1: nop
  Load_String: ldstr "Hello CIL code!"
  PrintToConsole: call void [mscorlib]System.Console::WriteLine(string)
  Nothing_2: nop
  WaitFor_KeyPress: call string [mscorlib]System.Console::ReadLine()
  RemoveValueFromStack: pop
  Leave_Function: ret
}
```

Говоря по существу, большая часть меток кода совершенно не обязательна. Единственный случай, когда метки кода по-настоящему необходимы, связан с написанием кода CIL, в котором используются разнообразные конструкции ветвления или зацикливания, т.к. с помощью меток можно указывать, куда должен быть направлен поток логики. В текущем примере все автоматически сгенерированные метки кода можно удалить безо всяких последствий:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack 8
  nop
  ldstr "Hello CIL code!"
  call void [mscorlib]System.Console::WriteLine(string)
  nop
  call string [mscorlib]System.Console::ReadLine()
  pop
  ret
}
```

Взаимодействие с CIL: модификация файла *.il

Теперь, когда вы имеете представление о том, из чего состоит базовый файл CIL, давайте завершим эксперимент с возвратным проектированием. Наша целью является модификация кода CIL в существующем файле *.il следующим образом:

1. Добавление ссылки на сборку System.Windows.Forms.dll.
2. Загрузка локальной строки внутри метода Main().
3. Вызов метода System.Windows.Forms.MessageBox.Show() с передачей локальной строковой переменной в качестве аргумента.

Первым делом понадобится добавить новую директиву .assembly (уточненную атрибутом extern), которая указывает, что нашей сборке требуется сборка System.Windows.Forms.dll. Для этого непосредственно после ссылки на внешнюю сборку mscorlib в файле *.il необходимо поместить такой код:

```
.assembly extern System.Windows.Forms
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
  .ver 4:0:0:0
}
```

Имейте в виду, что значение, назначаемое директиве .ver, может отличаться в зависимости от версии платформы .NET, установленной на машине разработки. Здесь вы видите, что применяется сборка System.Windows.Forms.dll версии 4.0.0.0, которая имеет маркер открытого ключа B77A5C561934E089. Открыв GAC (см. главу 14) и отыскав версию сборки System.Windows.Forms.dll, можно просто скопировать оттуда корректный номер версии и маркер открытого ключа.

Теперь нужно изменить текущую реализацию метода Main(). Для этого следует найти этот метод внутри файла *.il и удалить текущий код реализации (оставив незатронутыми только директивы .maxstack и .entrypoint):

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack 8
  // Написать новый код CIL!
}
```

Цель в том, чтобы поместить новую строку в стек и вызвать метод MessageBox.Show() (вместо Console.WriteLine()). Вспомните, что при ссылке на внешний тип должно указываться его полностью заданное имя (в сочетании с дружественным именем сборки). Также обратите внимание, что в терминах CIL каждый вызов метода снабжается

полностью заданным возвращаемым типом. Учитывая это, приведем метод `Main()` к следующему виду:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack 8
  ldstr "CIL is way cool"
  call valuetype [System.Windows.Forms]
    System.Windows.Forms.DialogResult
    [System.Windows.Forms]
  System.Windows.Forms.MessageBox::Show(string)
  pop
  ret
}
```

В действительности код CIL был модифицирован для соответствия такому определению класса C#:

```
class Program
{
  static void Main(string[] args)
  {
    System.Windows.Forms.MessageBox.Show("CIL is way cool");
  }
}
```

Компиляция кода CIL с помощью `ilasm.exe`

После сохранения обновленного файла `*.il` можно скомпилировать новую сборку .NET, используя утилиту `ilasm.exe` (компилятор CIL). Компилятор CIL поддерживает многочисленные параметры командной строки (их можно вывести, указав опцию `-?`), наиболее интересные из которых описаны в табл. 18.1.

Таблица 18.1. Распространенные параметры командной строки утилиты `ilasm.exe`

Параметр	Описание
`/debug`	Позволяет включить отладочную информацию (такую как имена локальных переменных и аргументов, а также номера строк)
`/dll`	Позволяет генерировать в качестве вывода файл `*.dll`
`/exe`	Позволяет генерировать в качестве вывода файл `*.exe`. Это стандартная опция, которую можно не указывать
`/key`	Позволяет компилировать сборку со строгим именем, применяя заданный файл `*.snk`
`/output`	Позволяет указать имя и расширение выходного файла. Если флаг `/output` не используется, то результирующее имя файла (без расширения) будет таким же, как имя первого исходного файла

Чтобы скомпилировать модифицированный файл `HelloProgram.il` в новую .NET-сборку `*.exe`, необходимо ввести в окне командной строки следующую команду:

```
ilasm /exe HelloProgram.il /output=NewAssembly.exe
```

В случае успешной компиляции на экране появится такой отчет:

```
Microsoft (R) .NET Framework IL Assembler. Version 4.0.30319.33440
Copyright (c) Microsoft Corporation. All rights reserved.
Assembling 'HelloProgram.il' to EXE --> 'NewAssembly.exe'
Source file is UTF-8

Assembled method Program::Main
Assembled method Program::.ctor
Creating PE file

Emitting classes:
Class 1: Program

Emitting fields and methods:
Global
Class 1 Methods: 2;

Emitting events and properties:
Global
Class 1
Writing PE file
Operation completed successfully
```

На этом этапе новое приложение можно запустить. Естественно, сообщение теперь отображается в отдельном окне сообщения, а не в окне консоли. Хотя этот простой пример не является особенно впечатляющим, он иллюстрирует один из сценариев применения возвратного проектирования на CIL.

Роль инструмента `peverify.exe`

При построении либо изменении сборок с использованием кода CIL рекомендуется всегда проверять, является ли скомпилированный двоичный образ правильно оформленным образом .NET, с помощью утилиты командной строки `peverify.exe`:

```
peverify NewAssembly.exe
```

Эта утилита проверяет достоверность всех кодов операций CIL внутри указанной сборки. Например, CIL требует, чтобы перед выходом из функции стек вычислений был пустым. Если вы забудете извлечь любые оставшиеся значения, то компилятор `ilasm.exe` все равно сгенерирует сборку (поскольку компилятор заботит только *синтаксис*). С другой стороны, утилита `peverify.exe` контролирует правильность семантики, и если стек не был опустошен перед выходом из функции, то она уведомит об этом до запуска кодовой базы.

Исходный код. Пример `RoundTrip` доступен в подкаталоге `Chapter_18`.

Директивы и атрибуты CIL

После демонстрации применения утилит `ildasm.exe` и `ilasm.exe` при возвратном проектировании можно переходить к более детальному исследованию синтаксиса и семантики CIL. В последующих разделах будет поэтапно рассматриваться процесс создания специального пространства имен, содержащего набор типов. Тем не менее, для простоты типы пока не будут иметь логики реализации своих членов. Разобравшись с созданием простых типов, внимание можно будет переключить на процесс определения "реальных" членов с использованием кодов операций CIL.

Указание ссылок на внешние сборки в CIL

Создадим в текстовом редакторе новый файл по имени CILTypes.il. Первой задачей в проекте CIL является перечисление внешних сборок, которые будут задействованы текущей сборкой. В рассматриваемом примере применяются только типы, находящиеся внутри сборки mscorlib.dll. Для этого понадобится указать в новом файле директиву .assembly с уточняющим атрибутом external. При добавлении ссылки на сборку со строгим именем, такую как mscorlib.dll, должны быть также указаны директивы .publickeytoken и .ver:

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
```

На заметку! Строго говоря, явно добавлять ссылку на внешнюю сборку mscorlib.dll не обязательно, потому что компилятор ilasm.exe сделает это автоматически. Однако для всех остальных внешних библиотек .NET, требующихся в проекте CIL, должны быть предусмотрены соответствующие директивы .assembly extern.

Определение текущей сборки в CIL

Следующее действие заключается в определении создаваемой сборки с использованием директивы .assembly. В простейшем случае сборка может быть определена за счет указания дружественного имени двоичного файла:

```
// Наша сборка.
.assembly CILTypes { }
```

В то время как это действительно определяет новую сборку .NET, обычно внутрь такого объявления будут помещаться дополнительные директивы. В рассматриваемом примере определение сборки необходимо снабдить номером версии 1.0.0.0 посредством директивы .ver (обратите внимание, что числа в номере версии отделяются друг от друга *двоеточиями*, а не точками, как принято в C#):

```
// Наша сборка.
.assembly CILTypes
{
  .ver 1:0:0:0
}
```

Из-за того, что сборка CILTypes является однофайловой (см. главу 14), ее определение завершается с применением следующей директивы .module, которая обозначает официальное имя двоичного файла .NET, CILTypes.dll:

```
// Наша сборка.
.assembly CILTypes
{
  .ver 1:0:0:0
}
// Модуль нашей однофайловой сборки.
.module CILTypes.dll
```

Кроме .assembly и .module существуют директивы CIL, которые позволяют дополнительно уточнять общую структуру создаваемого двоичного файла .NET. В табл. 18.2 перечислены некоторые распространенные директивы такого рода.

Таблица 18.2. Дополнительные директивы, связанные со сборками

Директива	Описание
.mresources	Если сборка использует внутренние ресурсы (такие как растровые изображения или таблицы строк), то данная директива применяется для указания имени файла, в котором содержатся ресурсы, подлежащие встраиванию в сборку
.subsystem	Эта директива CIL служит для указания предпочитаемого пользовательского интерфейса, в рамках которого желательно запускать сборку. Например, значение 2 указывает, что сборка должна выполняться в приложении с графическим пользовательским интерфейсом, а значение 3 — в консольном приложении

Определение пространств имен в CIL

После определения внешнего вида и поведения сборки (а также обязательных внешних ссылок) можно создать пространство имен .NET (MyNamespace), используя директиву .namespace:

```
// Наша сборка имеет единственное пространство имен.
.namespace MyNamespace {}
```

Подобно C# определения пространств имен CIL могут быть вложены в другие пространства имен. Хотя здесь нет нужды определять корневое пространство имен, ради интереса посмотрим, как создать корневое пространство имен MyCompany:

```
.namespace MyCompany
{
  .namespace MyNamespace {}
}
```

Как и C#, язык CIL позволяет определить вложенное пространство имен следующим образом:

```
// Определение вложенного пространства имен.
.namespace MyCompany.MyNamespace {}
```

Определение типов классов в CIL

Пустые пространства имен не особенно интересны, поэтому давайте рассмотрим процесс определения типов классов в CIL. Для определения нового типа класса предназначена директива .class. Тем не менее, эта простая директива может быть декорирована многочисленными дополнительными атрибутами, уточняющими природу типа. В целях иллюстрации добавим в наше пространство имен открытый класс под названием MyBaseClass. Как и в C#, если базовый класс явно не указан, то тип автоматически становится производным от System.Object:

```
.namespace MyNamespace
{
  // Предполагается базовый класс System.Object.
  .class public MyBaseClass {}
}
```

При построении типа, производного не от класса System.Object, применяется атрибут extends. Для ссылки на тип, определенный внутри той же самой сборки, язык CIL требует использования полностью заданного имени (однако если базовый тип находится внутри той же самой сборки, то префикс в виде дружественного имени сборки можно не указывать). Следовательно, показанная ниже попытка расширения MyBaseClass в результате дает ошибку на этапе компиляции:

```
// Этот код не скомпилируется!
.namespace MyNamespace
{
  .class public MyBaseClass {}
  .class public MyDerivedClass
    extends MyBaseClass {}
}
```

Чтобы корректно определить родительский класс для `MyDerivedClass`, потребуется указать полностью заданное имя `MyBaseClass`:

```
// Уже лучше!
.namespace MyNamespace
{
  .class public MyBaseClass {}
  .class public MyDerivedClass
    extends MyNamespace.MyBaseClass {}
}
```

В дополнение к атрибутам `public` и `extends` определение класса CIL может иметь множество добавочных квалификаторов, которые управляют видимостью типа, компоновкой полей и т.д. В табл. 18.3 описаны избранные атрибуты, которые могут применяться в сочетании с директивой `.class`.

Таблица 18.3. Избранные атрибуты, используемые вместе с директивой `.class`

Атрибут	Описание
`public`, `private`, `nested assembly`, `nested famandassem`, `nested family`, `nested famorassem`, `nested public`, `nested private`	Эти атрибуты применяются для указания видимости заданного типа. Как нетрудно заметить, язык CIL предлагает много других возможностей помимо тех, что доступны в C#. За дополнительными сведениями обращайтесь в документ ECMA 335
`abstract`, `sealed`	Эти два атрибута могут быть присоединены к директиве `.class` для определения, соответственно, абстрактного или запечатанного класса
`auto`, `sequential`, `explicit`	Эти атрибуты инструктируют среду CLR о том, как размещать данные полей в памяти. Для типов классов подходит стандартный флаг `auto`. Изменение стандартной установки может быть удобно в случае использования P/Invoke для обращения к неуправляемому коду C
`extends`, `implements`	Эти атрибуты позволяют определять базовый класс для типа (посредством `extends`) и реализовывать интерфейс в типе (с помощью `implements`)

Определение и реализация интерфейсов в CIL

Хотя это может показаться странным, но типы интерфейсов в CIL определяются с применением директивы `.class`. Тем не менее, когда директива `.class` декорирована атрибутом `interface`, тип трактуется как интерфейсный тип CTS. После определения интерфейс можно привязывать к типу класса или структуры с использованием атрибута `implements`:

```
.namespace MyNamespace
{
  // Определение интерфейса.
  .class public interface IMyInterface {}
  // Простой базовый класс.
  .class public MyBaseClass {}
  // Теперь MyDerivedClass реализует IMyInterface
  // и расширяет MyBaseClass.
  .class public MyDerivedClass
    extends MyNamespace.MyBaseClass
    implements MyNamespace.IMyInterface {}
}
```

На заметку! Конструкция `extends` должна предшествовать конструкции `implements`. Кроме того, в конструкции `implements` может содержаться список интерфейсов, разделенных запятыми.

Вспомните из главы 8, что интерфейсы могут выступать в роли базовых для других типов интерфейсов, позволяя строить иерархии интерфейсов. Однако вопреки возможным ожиданиям применять атрибут `extends` для порождения интерфейса А от интерфейса Б в CIL нельзя. Этот атрибут используется только для указания базового класса типа. Когда интерфейс необходимо расширить, еще раз будет применяться атрибут `implements`, например:

```
// Расширение интерфейсов в CIL.
.class public interface IMyInterface {}
.class public interface IMyOtherInterface
  implements MyNamespace.IMyInterface {}
```

Определение структур в CIL

Директива `.class` может использоваться для определения любой структуры CTS, если тип расширяет `System.ValueType`. Кроме того, такая директива `.class` должна уточняться атрибутом `sealed` (учитывая, что структуры никогда не могут выступать в роли базовых для других типов значений). Если попытаться поступить иначе, компилятор `ilasm.exe` выдаст сообщение об ошибке.

```
// Определение структуры всегда является запечатанным.
.class public sealed MyStruct
  extends [mscorlib]System.ValueType{}
```

Имейте в виду, что в CIL предусмотрен сокращенный синтаксис для определения типа структуры. В случае применения атрибута `value` новый тип автоматически становится производным от `[mscorlib]System.ValueType`. Следовательно, тип `MyStruct` можно было бы определить и так:

```
// Сокращенный синтаксис объявления структуры.
.class public sealed value MyStruct{}
```

Определение перечислений в CIL

Перечисления .NET порождены от класса `System.Enum`, который является `System.ValueType` (и потому также должен быть запечатанным). Чтобы определить перечисление в CIL, необходимо просто расширить `[mscorlib]System.Enum`:

```
// Перечисление.
.class public sealed MyEnum
  extends [mscorlib]System.Enum{}
```

Подобно структурам перечисления могут быть определены с помощью сокращенного синтаксиса, используя атрибут enum:

```
// Сокращенный синтаксис определения перечисления.
.class public sealed enum MyEnum{}
```

Вскоре будет показано, как указывать пары "имя-значение" перечисления.

На заметку! Еще один фундаментальный тип .NET — делегат — также имеет специфическое представление в CIL. Подробности приведены в главе 10.

Определение обобщений в CIL

Обобщенные типы также имеют собственное представление в синтаксисе CIL. Вспомните из главы 9, что обобщенный тип или член может иметь один и более параметров типа. Например, в типе List<T> определен один параметр типа, а в Dictionary<TKey, TValue> — два. В CIL количество параметров типа указывается с применением символа обратной одиночной кавычки (`), за которым следует число, представляющее количество параметров типа. Как и в C#, действительные значения параметров типа заключаются в угловые скобки.

На заметку! На большинстве клавиатур символ ` находится на клавише, расположенной над клавишей <Tab> (и слева от клавиши <1>).

Например, предположим, что требуется создать переменную List<T>, где T — тип System.Int32. В C# пришлось бы написать такой код:

```
void SomeMethod()
{
  List<int> myInts = new List<int>();
}
```

В CIL необходимо поступить следующим образом (этот код может находиться внутри любого метода CIL):

```
// В C#: List<int> myInts = new List<int>();
newobj instance void class [mscorlib]
  System.Collections.Generic.List`1<int32>::.ctor()
```

Обратите внимание, что этот обобщенный класс определен как List`1<int32>, поскольку List<T> имеет единственный параметр типа. А вот как определить тип Dictionary<string, int>:

```
// В C#: Dictionary<string, int> d = new Dictionary<string, int>();
newobj instance void class [mscorlib]
  System.Collections.Generic.Dictionary`2<string,int32>::.ctor()
```

Чтобы рассмотреть еще один пример, предположим, что имеется обобщенный тип, использующий в качестве параметра типа другой обобщенный тип. Код CIL выглядит следующим образом:

```
// В C#: List<List<int>> myInts = new List<List<int>>();
newobj instance void class [mscorlib]
  System.Collections.Generic.List`1<class
    [mscorlib]System.Collections.Generic.List`1<int32>>::.ctor()
```

Компиляция файла CILTypes.il

Несмотря на то что к определенным ранее типам пока не были добавлены члены или код реализации, этот файл *.il можно скомпилировать в .NET-сборку *.dll (так и нужно поступать ввиду отсутствия метода Main()). Для этого необходимо открыть окно командной строки и ввести показанную ниже команду для запуска ilasm.exe:

```
ilasm /dll CilTypes.il
```

Затем можно открыть скомпилированную сборку в ildasm.exe, чтобы удостовериться в создании каждого типа. После проверки содержимого сборки понадобится запустить в ее отношении утилиту peverify.exe:

```
peverify CilTypes.dll
```

Обратите внимание на сообщения об ошибках, т.к. все типы пусты. Вот частичный вывод:

```
Microsoft (R) .NET Framework PE Verifier. Version 4.0.30319.33440
Copyright (c) Microsoft Corporation. All rights reserved.
[MD]: Error: Value class has neither fields nor size parameter. [token:0x02000005]
     Ошибка: Класс значения не имеет ни полей, ни параметра размера.
[MD]: Error: Enum has no instance field. [token:0x02000006]
     Ошибка: Перечисление не имеет полей экземпляра.
...
```

Чтобы понять, каким образом заполнить тип содержимым, сначала необходимо ознакомиться с фундаментальными типами данных CIL.

Соответствия между типами данных в библиотеке базовых классов .NET, C# и CIL

В табл. 18.4 показано, как базовые классы .NET отображаются на соответствующие ключевые слова C#, а ключевые слова C# — на их представления в CIL. Кроме того, для каждого типа CIL приведено сокращенное константное обозначение. Как вы вскоре увидите, на эти константы часто ссылаются многие коды операций CIL.

Таблица 18.4. Отображение базовых классов .NET на ключевые слова C# и ключевых слов C# на CIL

Базовый класс .NET	Ключевое слово в C#	Представление CIL	Константное обозначение CIL
System.SByte	sbyte	int8	I1
System.Byte	byte	unsigned int8	U1
System.Int16	short	int16	I2
System.UInt16	ushort	unsigned int16	U2
System.Int32	int	int32	I4
System.UInt32	uint	unsigned int32	U4
System.Int64	long	int64	I8
System.UInt64	ulong	unsigned int64	U8
System.Char	char	char	CHAR
System.Single	float	float32	R4

Окончание табл. 18.4

Базовый класс .NET	Ключевое слово в C#	Представление CIL	Константное обозначение CIL
System.Double	double	float64	R8
System.Boolean	bool	bool	BOOLEAN
System.String	string	string	-
System.Object	object	object	-
System.Void	void	void	VOID

На заметку! Типы System.IntPtr и System.UIntPtr отображаются на собственные типы int и unsigned int в CIL (это полезно знать, т.к. они интенсивно применяются во многих сценариях взаимодействия с COM и P/Invoke).

Определение членов типов в CIL

Как вам уже известно, типы .NET могут поддерживать разнообразные члены. Перечисления содержат набор пар "имя-значение". Структуры и классы могут иметь конструкторы, поля, методы, свойства, статические члены и т.д. В предшествующих семнадцати главах книги вы уже видели частичные определения в CIL упомянутых элементов, но давайте еще раз кратко повторим, каким образом различные члены отображаются на примитивы CIL.

Определение полей данных в CIL

Перечисления, структуры и классы могут поддерживать поля данных. Во всех случаях для их определения будет использоваться директива .field. Например, давайте добавим к перечислению MyEnum следующие три пары "имя-значение" (обратите внимание, что значения указаны в круглых скобках):

```
.class public sealed enum MyEnum
{
  .field public static literal valuetype
   MyNamespace.MyEnum A = int32(0)
  .field public static literal valuetype
   MyNamespace.MyEnum B = int32(1)
  .field public static literal valuetype
   MyNamespace.MyEnum C = int32(2)
}
```

Поля, находящиеся внутри области действия производного от System.Enum типа .NET, уточняются с применением атрибутов static и literal. Как не трудно догадаться, эти атрибуты указывают, что данные полей должны быть фиксированными значениями, доступными только из самого типа (например, MyEnum.A).

На заметку! Значения, присваиваемые полям в перечислении, могут быть также представлены в шестнадцатеричном формате с помощью префикса 0x.

Конечно, когда нужно определить элемент поля данных внутри класса или структуры, вы не ограничены только открытыми статическими литеральными данными.

Например, класс MyBaseClass можно было бы модифицировать для поддержки двух закрытых полей данных уровня экземпляра со стандартными значениями:

```
.class public MyBaseClass
{
  .field private string stringField = "hello!"
  .field private int32 intField = int32(42)
}
```

Как и в C#, поля данных класса будут автоматически инициализироваться подходящими стандартными значениями. Чтобы предоставить пользователю объекта возможность указывать собственные значения во время создания закрытых полей данных, потребуется создать специальные конструкторы.

Определение конструкторов типа в CIL

Спецификация CTS поддерживает создание конструкторов как уровня экземпляра, так и уровня класса (статических). В CIL конструкторы уровня экземпляра представляются с использованием лексемы .ctor, тогда как конструкторы уровня класса — посредством лексемы .cctor (class constructor — конструктор класса). Обе лексемы CIL должны сопровождаться атрибутами rtspecialname (return type special name — специальное имя возвращаемого типа) и specialname. В двух словах, эти атрибуты применяются для обозначения специфической лексемы CIL, которая может трактоваться уникальным образом в любом отдельно взятом языке .NET. Например, в языке C# конструкторы не определяют возвращаемый тип, но в CIL возвращаемым значением конструктора на самом деле является void:

```
.class public MyBaseClass
{
  .field private string stringField
  .field private int32 intField
  .method public hidebysig specialname rtspecialname
    instance void .ctor(string s, int32 i) cil managed
  {
    // Добавить код реализации...
  }
}
```

Обратите внимание, что директива .ctor снабжена атрибутом instance (поскольку это не статический конструктор). Атрибуты cil managed указывают на то, что внутри данного метода содержится код CIL, а не неуправляемый код, который может использоваться при выполнении запросов P/Invoke.

Определение свойств в CIL

Свойства и методы также имеют специфические представления в CIL. В качестве примера модифицируем класс MyBaseClass с целью поддержки открытого свойства по имени TheString, написав следующий код CIL (обратите внимание на применение атрибута specialname):

```
.class public MyBaseClass
{
  ...
  .method public hidebysig specialname
    instance string get_TheString() cil managed
  {
    // Добавить код реализации...
  }
```

```
  .method public hidebysig specialname
    instance void set_TheString(string 'value') cil managed
  {
    // Добавить код реализации...
  }
  .property instance string TheString()
  {
    .get instance string
      MyNamespace.MyBaseClass::get_TheString()
    .set instance void
      MyNamespace.MyBaseClass::set_TheString(string)
  }
}
```

Внутри CIL свойство отображается на пару методов, имеющих префиксы `get_` и `set_`. В директиве `.property` используются связанные директивы `.get` и `.set` для отображения синтаксиса свойств на подходящие "специально именованные" методы.

На заметку! Обратите внимание, что входной параметр метода `set_` в свойстве помещен в одинарные кавычки и представляет имя лексемы, которая должна применяться в правой части операции присваивания внутри области определения метода.

Определение параметров членов

Коротко говоря, параметры в CIL указываются (более или менее) идентично тому, как это делается в C#. Например, каждый параметр определяется путем указания его типа данных, за которым следует имя параметра. Более того, подобно C# язык CIL позволяет определять входные, выходные и передаваемые по ссылке параметры. Вдобавок в CIL допускается определять массив параметров (соответствует ключевому слову `params` в C#), а также необязательные параметры.

Чтобы проиллюстрировать процесс определения параметров в низкоуровневом коде CIL, предположим, что необходимо построить метод, который принимает параметр `int32` (по значению), параметр `int32` (по ссылке), параметр `[mscorlib]System.Collection.ArrayList` и один выходной параметр (типа `int32`). В C# этот метод выглядел бы примерно так:

```
public static void MyMethod(int inputInt,
  ref int refInt, ArrayList ar, out int outputInt)
{
  outputInt = 0;  // Просто чтобы удовлетворить компилятор C#...
}
```

После отображения этого метода на код CIL вы обнаружите, что ссылочные параметры C# помечаются символом амперсанда (`&`), который дополняет лежащий в основе тип данных (`int32&`).

Выходные параметры также снабжаются суффиксом `&`, но при этом дополнительно уточняются лексемой `[out]` языка CIL. Вдобавок, если параметр относится к ссылочному типу (в этом случае `[mscorlib]System.Collections.ArrayList`), то перед типом данных указывается лексема `class` (не путайте ее с директивой `.class`):

```
.method public hidebysig static void MyMethod(int32 inputInt,
  int32& refInt,
  class [mscorlib]System.Collections.ArrayList ar,
  [out] int32& outputInt) cil managed
{
  ...
}
```

Исследование кодов операций CIL

Последний аспект кода CIL, который будет здесь рассматриваться, связан с ролью разнообразных кодов операций. Вспомните, что код операции — это просто лексема CIL, используемая при построении логики реализации для заданного члена. Все коды операций CIL (которых довольно много) могут быть разделены на три обширных категории:

- коды операций, которые управляют потоком выполнения программы;
- коды операций, которые вычисляют выражения;
- коды операций, которые получают доступ к значениям в памяти (через параметры, локальные переменные и т.д.).

В табл. 18.5 описаны наиболее полезные коды операций, имеющие прямое отношение к логике реализации членов; они сгруппированы по функциональности.

Таблица 18.5. Различные коды операций CIL, связанные с реализацией членов

Коды операций	Описание
`add`, `sub`, `mul`, `div`, `rem`	Позволяют выполнять сложение, вычитание, умножение и деление двух значений (`rem` возвращает остаток от деления)
`and`, `or`, `not`, `xor`	Позволяют выполнять побитовые операции над двумя значениями
`ceq`, `cgt`, `clt`	Позволяют сравнивать два значения в стеке разными способами, например: `ceq` — сравнение на равенство `cgt` — сравнение "больше чем" `clt` — сравнение "меньше чем"
`box`, `unbox`	Применяются для преобразования между ссылочными типами и типами значений
`ret`	Используется для выхода из метода и возврата значения вызывающему коду (при необходимости)
`beq`, `bgt`, `ble`, `blt`, `switch`	Применяются для управления логикой ветвления внутри метода (вдобавок ко многим другим связанным кодам операций), например: `beq` — переход к метке в коде, если равно `bgt` — переход к метке в коде, если больше чем `ble` — переход к метке в коде, если меньше или равно `blt` — переход к метке в коде, если меньше чем Все коды операций, связанные с ветвлением, требуют указания в коде CIL метки для перехода в случае, если результат проверки оказывается истинным
`call`	Используется для вызова члена заданного типа
`newarr`, `newobj`	Позволяют размещать в памяти новый массив или новый объект (соответственно)

Коды операций из следующей обширной категории (подмножество которых описано в табл. 18.6) применяются для загрузки (заталкивания) аргументов в виртуальный стек выполнения. Обратите внимание, что все эти ориентированные на загрузку коды операций имеют префикс `ld` (load — загрузить).

Таблица 18.6. Основные коды операций CIL, связанные с загрузкой в стек

Код операции	Описание
`ldarg` (и многочисленные вариации)	Загружает в стек аргумент метода. Помимо общей формы `ldarg` (которая работает с индексом, идентифицирующим аргумент), существует множество других вариаций. Например, коды операций `ldarg`, которые имеют числовой суффикс (`ldarg_0`), жестко кодируют загружаемый аргумент. Кроме того, вариации `ldarg` позволяют жестко кодировать тип данных, используя константное обозначение CIL из табл. 18.4 (скажем, `ldarg_I4` для `int32`), а также тип данных и значение (к примеру, `ldarg_I4_5` для загрузки `int32` со значением 5)
`ldc` (и многочисленные вариации)	Загружает в стек константное значение
`ldfld` (и многочисленные вариации)	Загружает в стек значение поля уровня экземпляра
`ldloc` (и многочисленные вариации)	Загружает в стек значение локальной переменной
`ldobj`	Получает все значения, собранные размещенным в куче объектом, и помещает их в стек
`ldstr`	Загружает в стек строковое значение

В дополнение к набору кодов операций, связанных с загрузкой, CIL предоставляет многочисленные коды операций, которые явно извлекают из стека самое верхнее значение. Как было показано в нескольких начальных примерах, извлечение значения из стека обычно предусматривает его сохранение во временном локальном хранилище для дальнейшего использования (наподобие параметра для предстоящего вызова метода). Многие коды операций, извлекающие текущее значение из виртуального стека выполнения, снабжены префиксом `st` (store — сохранить). В табл. 18.7 описаны некоторые распространенные коды операций.

Таблица 18.7. Разнообразные коды операций CIL, связанные с извлечением из стека

Код операции	Описание
`pop`	Удаляет значение, которое в текущий момент находится на верхушке стека вычислений, но не заботится о его сохранении
`starg`	Сохраняет значение из верхушки стека в аргументе метода с определенным индексом
`stloc` (и многочисленные вариации)	Извлекает текущее значение из верхушки стека вычислений и сохраняет его в списке локальных переменных по указанному индексу
`stobj`	Копирует значение указанного типа из стека вычислений по заданному адресу в памяти
`stsfld`	Заменяет значение статического поля значением из стека вычислений

Имейте в виду, что различные коды операций CIL будут неявно извлекать значения из стека во время выполнения своих задач. Например, при вычитании одного числа из другого с применением кода операции `sub` должно быть очевидным, что перед самим вычислением операция `sub` должна извлечь из стека два следующих доступных значения. Результат вычисления снова помещается в стек.

Директива .maxstack

При написании кода реализации методов на низкоуровневом языке CIL необходимо помнить о специальной директиве под названием .maxstack. С ее помощью устанавливается максимальное количество переменных, которые могут находиться внутри стека в любой заданный момент времени на протяжении периода выполнения метода. Хорошая новость в том, что директива .maxstack имеет стандартное значение (8), которое должно подходить для подавляющего большинства создаваемых методов. Тем не менее, если вы хотите указывать все явно, то можете вручную подсчитать количество локальных переменных в стеке и определить это значение в .maxstack:

```
.method public hidebysig instance void
  Speak() cil managed
{
  // Внутри области действия этого метода в стеке находится
  // в точности одно значение (строковый литерал).
  .maxstack 1
  ldstr "Hello there..."
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

Объявление локальных переменных в CIL

Теперь давайте посмотрим, как объявлять локальные переменные. Предположим, что необходимо построить в CIL метод по имени MyLocalVariables(), который не принимает аргументов и возвращает void, и определить в нем три локальные переменные с типами System.String, System.Int32 и System.Object. В C# такой метод выглядел бы следующим образом (вспомните, что локальные переменные не получают стандартные значения и потому перед использованием должны быть инициализированы):

```
public static void MyLocalVariables()
{
  string myStr = "CIL code is fun!";
  int myInt = 33;
  object myObj = new object();
}
```

А вот как реализовать метод MyLocalVariables() на языке CIL:

```
.method public hidebysig static void
  MyLocalVariables() cil managed
{
  .maxstack 8
  // Определить три локальные переменные.
  .locals init ([0] string myStr, [1] int32 myInt, [2] object myObj)
  // Загрузить строку в виртуальный стек выполнения.
  ldstr "CIL code is fun!"
  // Извлечь текущее значение и сохранить его в локальной переменной [0].
  stloc.0
  // Загрузить константу типа i4 (сокращение для int32) со значением 33.
  ldc.i4 33
  // Извлечь текущее значение и сохранить его в локальной переменной [1].
  stloc.1
  // Создать новый объект и поместить его в стек.
  newobj instance void [mscorlib]System.Object::.ctor()
```

```
  // Извлечь текущее значение и сохранить его в локальной переменной [2].
  stloc.2
  ret
}
```

Первым шагом при размещении локальных переменных с помощью CIL является применение директивы `.locals` в паре с атрибутом `init`. Внутри связанных с каждой переменной квадратных скобок понадобится указать определенный числовой индекс (`[0]`, `[1]` и `[2]`). Каждый индекс идентифицируется типом данных и необязательным именем переменной. После определения локальных переменных значения загружаются в стек (с использованием различных кодов операций загрузки) и сохраняются в этих локальных переменных (с помощью кодов операций сохранения).

Отображение параметров на локальные переменные в CIL

Вы уже видели, каким образом объявляются локальные переменные в CIL с применением директивы `.local init`; однако осталось еще взглянуть на то, как входные параметры отображаются на локальные переменные. Рассмотрим показанный ниже статический метод C#:

```
public static int Add(int a, int b)
{
  return a + b;
}
```

Этот невинно выглядящий метод требует немалого объема кодирования на языке CIL. Во-первых, входные аргументы (`a` и `b`) должны быть помещены в виртуальный стек выполнения с использованием кода операции `ldarg` (load argument — загрузить аргумент). Во-вторых, с помощью кода операции `add` из стека будут извлечены следующие два значения и просуммированы с сохранением результата обратно в стек. В-третьих, сумма будет извлечена из стека и возвращена вызывающему коду посредством кода операции `ret`. Дизассемблировав этот метод C# с применением `ildasm.exe`, вы обнаружите множество дополнительных лексем, которые вставил компилятор `csc.exe`, но основная часть кода CIL довольно проста:

```
.method public hidebysig static int32 Add(int32 a,
  int32 b) cil managed
{
  .maxstack 2
  ldarg.0       // Загрузить a в стек.
  ldarg.1       // Загрузить b в стек.
  add           // Сложить оба значения.
  ret
}
```

Скрытая ссылка `this`

Обратите внимание, что ссылка на два входных аргумента (`a` и `b`) в коде CIL производится с использованием их индексных позиций (`0` и `1`), т.к. индексация в виртуальном стеке выполнения начинается с нуля.

Во время исследования или написания кода CIL нужно помнить о том, что каждый нестатический метод, принимающий входные аргументы, автоматически получает неявный дополнительный параметр, который представляет собой ссылку на текущий объект (аналогичный ключевому слову `this` в C#). Скажем, если бы метод `Add()` был определен как *нестатический*:

```
// Больше не является статическим!
public int Add(int a, int b)
{
  return a + b;
}
```

то входные аргументы a и b загружались бы с применением кодов операций ldarg.1 и ldarg.2 (а не ожидаемых ldarg.0 и ldarg.1). Причина в том, что ячейка 0 в действительности содержит неявную ссылку this. Взгляните на следующий псевдокод:

```
// Это ТОЛЬКО псевдокод!
.method public hidebysig static int32 AddTwoIntParams(
  MyClass_HiddenThisPointer this, int32 a, int32 b) cil managed
{
  ldarg.0  // Загрузить MyClass_HiddenThisPointer в стек.
  ldarg.1  // Загрузить a в стек.
  ldarg.2  // Загрузить b в стек.
  ...
}
```

Представление итерационных конструкций в CIL

Итерационные конструкции в языке программирования C# реализуются посредством ключевых слов for, foreach, while и do, каждое из которых имеет специальное представление в CIL. Для примера рассмотрим следующий классический цикл for:

```
public static void CountToTen()
{
  for(int i = 0; i < 10; i++)
    ;
}
```

Вспомните, что для управления прекращением потока выполнения, когда удовлетворено некоторое условие, используются коды операций br (br, blt и т.д.). В приведенном примере указано условие, согласно которому выполнение цикла for должно прекращаться, когда значение локальной переменной i становится больше или равно 10. С каждым проходом к значению i добавляется 1, после чего проверяемое условие оценивается заново.

Также вспомните, что в случае применения любого кода операции CIL, предназначенного для ветвления, должна быть определена специфичная метка кода (или две), обозначающая место, куда будет произведен переход при истинном значении условия. С учетом всего сказанного рассмотрим показанный ниже (дополненный комментариями) код CIL, который сгенерирован утилитой ildasm.exe (вместе с автоматически созданными метками):

```
.method public hidebysig static void CountToTen() cil managed
{
  .maxstack 2
  .locals init ([0] int32 i)  // Инициализировать локальную целочисленную
                              // переменную i.
  IL_0000: ldc.i4.0       // Загрузить это значение в стек.
  IL_0001: stloc.0        // Сохранить это значение по индексу 0.
  IL_0002: br.s IL_0008   // Перейти на метку IL_0008.
  IL_0004: ldloc.0        // Загрузить значение переменной по индексу 0.
  IL_0005: ldc.i4.1       // Загрузить значение 1 в стек.
  IL_0006: add            // Добавить текущее значение в стеке по индексу 0.
  IL_0007: stloc.0
  IL_0008: ldloc.0        // Загрузить значение по индексу 0.
```

```
  IL_0009: ldc.i4.s 10    // Загрузить значение 10 в стек.
  IL_000b: blt.s IL_0004  // Меньше чем? Если да, то перейти на метку IL_0004.
  IL_000d: ret
}
```

Код CIL начинается с определения локальной переменной типа `int32` и ее загрузки в стек. Затем производятся переходы туда и обратно между метками `IL_0008` и `IL_0004`, во время каждого из которых значение `i` увеличивается на 1 и проверяется на предмет того, что оно все еще меньше 10. Как только условие будет нарушено, осуществляется выход из метода.

Исходный код. Пример `CilTypes` доступен в подкаталоге `Chapter_18`.

Построение сборки .NET на CIL

После ознакомления с синтаксисом и семантикой языка CIL самое время закрепить изученный материал, построив приложение .NET с использованием утилиты `ilasm.exe` и текстового редактора по выбору. Приложение будет состоять из закрытой однофайловой сборки `*.dll`, содержащей определения двух типов классов, и консольной программы `*.exe`, которая взаимодействует с этими типами.

Построение сборки `CILCars.dll`

В первую очередь понадобится построить сборку `*.dll`, которую будет потреблять клиент. Создадим в текстовом редакторе файл `*.il` по имени `CILCars.il`. В этой однофайловой сборке задействуются две внешние сборки .NET. Модифицируем код следующим образом:

```
// Ссылки на mscorlib.dll и System.Windows.Forms.dll.
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
.assembly extern System.Windows.Forms
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
// Определить однофайловую сборку.
.assembly CILCars
{
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module CILCars.dll
```

Сборка будет содержать два типа класса. Первый тип, `CILCar`, определяет два поля данных (для простоты открытых) и специальный конструктор. Второй тип, `CarInfoHelper`, определяет единственный статический метод по имени `DisplayCarInfo()`, который принимает параметр типа `CILCar` и возвращает `void`. Оба типа находятся в пространстве имен `CILCars`. Вот как может быть реализован тип `CILCar` на языке CIL:

```
// Реализация типа CILCars.CILCar.
.namespace CILCars
{
  .class public auto ansi beforefieldinit CILCar
    extends [mscorlib]System.Object
  {
    // Поля данных CILCar.
    .field public string petName
    .field public int32 currSpeed

    // Специальный конструктор просто позволяет
    // вызывающему коду присваивать поля данных.
    .method public hidebysig specialname rtspecialname
    instance void .ctor(int32 c, string p) cil managed
    {
      .maxstack 8

      // Загрузить первый аргумент в стек и вызвать конструктор базового класса.
      ldarg.0 // объект this, не значение int32!
      call instance void [mscorlib]System.Object::.ctor()

      // Загрузить первый и второй аргументы в стек.
      ldarg.0 // объект this
      ldarg.1 // аргумент int32

      // Сохранить самый верхний элемент стека (int32) в поле currSpeed.
      stfld int32 CILCars.CILCar::currSpeed

      // Загрузить строковый аргумент и сохранить в поле petName.
      ldarg.0 // объект this
      ldarg.2 // аргумент string
      stfld string CILCars.CILCar::petName
      ret
    }
  }
}
```

Учитывая то, что реальным первым аргументом для любого нестатического члена является ссылка на текущий объект, в первом блоке CIL просто загружается ссылка на текущий объект и вызывается конструктор базового класса. Затем входные аргументы конструктора помещаются в стек и сохраняются в соответствующих полях данных с помощью кода операции `stfld` (store in field — сохранить в поле).

Теперь давайте реализуем второй тип из этого пространства имен — `CILCarInfo`. Главным в нем будет статический метод `Display()`. Роль метода заключается в том, чтобы принять входной параметр `CILCar`, извлечь значения его полей данных и отобразить их в окне сообщений Windows Forms. Ниже приведена полная реализация типа `CILCarInfo`, а после нее — необходимые пояснения:

```
.class public auto ansi beforefieldinit CILCarInfo
  extends [mscorlib]System.Object
{
  .method public hidebysig static void
    Display(class CILCars.CILCar c) cil managed
  {
    .maxstack 8

    // Нам нужна локальная строковая переменная.
    .locals init ([0] string caption)
```

```
    // Загрузить строку и входной параметр CILCar в стек.
    ldstr "{0}'s speed is:"
    ldarg.0

    // Поместить в стек значение поля petName из CILCar
    // и вызвать статический метод String.Format().
    ldfld string CILCars.CILCar::petName
    call string [mscorlib]System.String::Format(string, object)
    stloc.0

    // Загрузить значение поля currSpeed и получить его строковое
    // представление (обратите внимание на вызов ToString()).
    ldarg.0
    ldflda int32 CILCars.CILCar::currSpeed
    call instance string [mscorlib]System.Int32::ToString()
    ldloc.0

    // Вызвать метод MessageBox.Show() с загруженными значениями.
    call valuetype [System.Windows.Forms]
       System.Windows.Forms.DialogResult
       [System.Windows.Forms]
       System.Windows.Forms.MessageBox::Show(string, string)
    pop
    ret
  }
}
```

Хотя объем кода CIL здесь немного больше, чем в реализации `CILCar`, он по-прежнему прямолинеен. Поскольку определяется статический метод, беспокоиться о скрытой ссылке на текущий объект больше не требуется (таким образом, код операции `ldarg.0` действительно загружает входной аргумент `CILCar`).

Метод начинается с загрузки в стек строки `"{0}'s speed is"` и следом за ней аргумента `CILCar`. Затем загружается значение `petName` и вызывается статический метод `System.String.Format()` для замены заполнителя в фигурных скобках дружественным именем `CILCar`.

С помощью той же самой общей процедуры обрабатывается поле `currSpeed`, но на этот раз применяется код операции `ldflda`, который приводит к загрузке в стек адреса аргумента. Далее вызывается метод `System.Int32.ToString()` с целью преобразования находящегося по указанному адресу значения в строковый тип. И, наконец, после того, как обе строки сформатированы должным образом, вызывается метод `MessageBox.Show()`.

Теперь можно скомпилировать новую сборку `*.dll` с помощью `ilasm.exe`:

```
ilasm /dll CILCars.il
```

и проверить содержащийся внутри нее код CIL посредством утилиты `peverify.exe`:

```
peverify CILCars.dll
```

Построение сборки `CILCarClient.exe`

Теперь можно построить простую сборку `*.exe` с методом `Main()`, который будет:

- создавать объект `CILCar`;
- передавать объект статическому методу `CILCarInfo.Display()`.

Создадим новый файл `CarClient.il`, добавим в него ссылки на внешние сборки `mscorlib.dll` и `CILCars.dll` (не забыв поместить копию `CILCars.dll` в каталог клиентского приложения) и определим в нем единственный тип (`Program`), который будет манипулировать сборкой `CILCars.dll`.

Вот полный код:

```
// Ссылки на внешние сборки.
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
  .ver 4:0:0:0
}
.assembly extern CILCars
{
  .ver 1:0:0:0
}
// Наша исполняемая сборка.
.assembly CarClient
{
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module CarClient.exe
// Реализация типа Program.
.namespace CarClient
{
  .class private auto ansi beforefieldinit Program
  extends [mscorlib]System.Object
  {
    .method private hidebysig static void
    Main(string[] args) cil managed
    {
      // Пометить точку входа в сборке *.exe.
      .entrypoint
      .maxstack 8
      // Объявить локальную переменную типа CILCar и поместить
      // значения в стек для вызова конструктора.
      .locals init ([0] class
      [CILCars]CILCars.CILCar myCilCar)
      ldc.i4 55
      ldstr "Junior"
      // Создать новый объект CILCar; сохранить и загрузить ссылку на него.
      newobj instance void
        [CILCars]CILCars.CILCar::.ctor(int32, string)
      stloc.0
      ldloc.0
      // Вызвать метод Display(), передав ему самое верхнее значение из стека.
      call void [CILCars]
        CILCars.CILCarInfo::Display(
          class [CILCars]CILCars.CILCar)
      ret
    }
  }
}
```

Здесь важно отметить код операции `.entrypoint`. Как упоминалось ранее в главе, этот код используется для пометки метода в сборке `*.exe`, который должен служить точкой входа. Учитывая, что посредством `.entrypoint` среда CLR идентифицирует начальный метод для выполнения, этот метод на самом деле может иметь любое имя, хотя

для него было выбрано стандартное имя `Main()`. В оставшемся коде CIL метода `Main()` производятся типичные операции по помещению и извлечению значений из стека.

Тем не менее, для создания объекта `CILCar` применяется код операции `.newobj`. В связи с этим вспомните, что для вызова члена типа в коде CIL используется синтаксис в виде двойного двоеточия и, как обычно, указывается полностью заданное имя типа. Далее можно скомпилировать новый файл с помощью `ilasm.exe`, проверить сборку с применением `peverify.exe` и запустить программу. Введите следующие команды в окне командной строки:

```
ilasm CarClient.il
peverify CarClient.exe
CarClient.exe
```

Исходный код. Пример `CilCars` доступен в подкаталоге `Chapter_18`.

Динамические сборки

Естественно, процесс построения сложных приложений .NET на языке CIL окажется довольно-таки неблагодарным трудом. С одной стороны, CIL является чрезвычайно выразительным языком программирования, который позволяет взаимодействовать со всеми программными конструкциями, разрешенными CTS. С другой стороны, написание низкоуровневого кода CIL утомительно, сопряжено с большими затратами времени и подвержено ошибкам. Хотя и правда, что знание — сила, вас может интересовать, насколько важно держать все правила синтаксиса CIL в голове. Ответ: зависит от ситуации. Разумеется, в большинстве случаев при программировании приложений .NET просматривать, редактировать или писать код CIL не потребуется. Однако знание основ языка CIL означает готовность перейти к исследованию мира динамических сборок (как противоположности статическим сборкам) и роли пространства имен `System.Reflection.Emit`.

Первым может возникнуть вопрос: чем отличаются статические и динамические сборки? По определению *статической сборкой* называется двоичная сборка .NET, которая загружается прямо из дискового хранилища, т.е. на момент запроса средой CLR она находится где-то на жестком диске в физическом файле (или в наборе файлов, если сборка многофайловая). Как и можно было предположить, при каждой компиляции исходного кода C# в результате получается статическая сборка.

С другой стороны, *динамическая сборка* создается в памяти на лету с использованием типов из пространства имен `System.Reflection.Emit`. Это пространство имен делает возможным построение сборки и ее модулей, определений типов и логики реализации на CIL *во время выполнения*. Затем сборку, расположенную в памяти, можно сохранить на диск, получив в результате новую статическую сборку. Ясно, что процесс создания динамических сборок с помощью пространства имен `System.Reflection.Emit` требует понимания природы кодов операций CIL.

Несмотря на то что создание динамических сборок является довольно сложной (и редкой) задачей программирования, оно может быть удобным в разнообразных обстоятельствах. Ниже перечислены примеры.

- Вы строите инструмент программирования .NET, который должен быть способен генерировать сборки по требованию на основе пользовательского ввода.
- Вы создаете приложение, которое нуждается в генерации прокси для удаленных типов на лету, основываясь на полученных метаданных.
- Вам необходима возможность загрузки статической сборки и динамической вставки в двоичный образ новых типов.

Итак, давайте посмотрим, какие типы доступны в пространстве имен `System.Reflection.Emit`.

Исследование пространства имен `System.Reflection.Emit`

Создание динамической сборки требует некоторых знаний кодов операций CIL, но типы из пространства имен `System.Reflection.Emit` максимально возможно скрывают сложность языка CIL. Скажем, вместо непосредственного указания необходимых директив и атрибутов CIL для определения типа класса можно просто применять класс `TypeBuilder`. Аналогично, если нужно определить новый конструктор уровня экземпляра, то не придется задавать лексему `specialname`, `rtspecialname` или `.ctor`; взамен можно использовать класс `ConstructorBuilder`. Основные члены пространства имен `System.Reflection.Emit` описаны в табл. 18.8.

Таблица 18.8. Избранные члены пространства имен `System.Reflection.Emit`

Член	Описание
`AssemblyBuilder`	Применяется для создания сборки (`*.dll` или `*.exe`) во время выполнения. В сборках `*.exe` должен вызываться метод `ModuleBuilder.SetEntryPoint()` для установки метода, который является точкой входа в модуль. Если ни одной точки входа не указано, то будет генерироваться файл `*.dll`
`ModuleBuilder`	Используется для определения набора модулей внутри текущей сборки
`EnumBuilder`	Применяется для создания типа перечисления .NET
`TypeBuilder`	Может использоваться для создания классов, интерфейсов, структур и делегатов внутри модуля во время выполнения

`MethodBuilder`
`LocalBuilder`
`PropertyBuilder`
`FieldBuilder`
`ConstructorBuilder`
`CustomAttributeBuilder`
`ParameterBuilder`
`EventBuilder`
	Применяются для создания членов типов (таких как методы, локальные переменные, свойства, конструкторы и атрибуты) во время выполнения

`ILGenerator`	Выпускает коды операций CIL для указанного члена типа
`OpCodes`	Предоставляет многочисленные поля, которые отображаются на коды операций CIL. Используется вместе с разнообразными членами типа `System.Reflection.Emit.ILGenerator`

В целом типы из пространства имен `System.Reflection.Emit` позволяют представлять низкоуровневые лексемы CIL программным образом во время построения динамической сборки. Вы увидите многие из них в рассматриваемом далее примере; тем не менее, тип `ILGenerator` заслуживает специального внимания.

Роль типа `System.Reflection.Emit.ILGenerator`

Роль типа `ILGenerator` заключается во вставке кодов операций CIL внутрь заданного члена типа. Однако создавать объекты `ILGenerator` напрямую невозможно, т.к. этот тип не имеет открытых конструкторов. Взамен объекты `ILGenerator` должны получаться путем вызова специфических методов типов, связанных с построителями (таких как `MethodBuilder` и `ConstructorBuilder`). Вот пример:

```
// Получить объект ILGenerator из объекта ConstructorBuilder по имени myCtorBuilder.
ConstructorBuilder myCtorBuilder =
  new ConstructorBuilder(/* ...разнообразные аргументы... */);
ILGenerator myCILGen = myCtorBuilder.GetILGenerator();
```

Имея объект `ILGenerator`, можно выпускать низкоуровневые коды операций CIL с помощью любых его методов. Некоторые (но не все) методы `ILGenerator` кратко описаны в табл. 18.9.

Таблица 18.9. Избранные методы класса `ILGenerator`

Метод	Описание
`BeginCatchBlock()`	Начинает блок `catch`
`BeginExceptionBlock()`	Начинает блок исключения
`BeginFinallyBlock()`	Начинает блок `finally`
`BeginScope()`	Начинает лексическую область
`DeclareLocal()`	Объявляет локальную переменную
`DefineLabel()`	Объявляет новую метку
`Emit()`	Многократно перегружен, чтобы позволить выпускать коды операций CIL
`EmitCall()`	Помещает в поток CIL код операции `call` или `callvirt`
`EmitWriteLine()`	Выпускает вызов `Console.WriteLine()` со значениями разных типов
`EndExceptionBlock()`	Завершает блок исключения
`EndScope()`	Завершает лексическую область
`ThrowException()`	Выпускает инструкцию для генерации исключения
`UsingNamespace()`	Указывает пространство имен, которое должно применяться при оценке локальных и наблюдаемых значений в текущей активной лексической области

Основным методом класса `ILGenerator` является `Emit()`, который работает в сочетании с типом `System.Reflection.Emit.OpCodes`. Как упоминалось ранее в главе, данный тип открывает доступ к множеству полей только для чтения, которые отображаются на низкоуровневые коды операций CIL. Полный набор этих членов документирован в онлайновой справочной системе, и далее в главе вы неоднократно встретите примеры их использования.

Выпуск динамической сборки

Чтобы проиллюстрировать процесс определения сборки .NET во время выполнения, давайте рассмотрим процесс создания однофайловой динамической сборки по имени `MyAssembly.dll`. Внутри модуля находится класс `HelloWorld`. Класс `HelloWorld` поддерживает стандартный конструктор и специальный конструктор, который применяется для присваивания значения закрытой переменной-члена (`theMessage`) типа `string`. Вдобавок в классе `HelloWorld` имеется открытый метод экземпляра под названием `SayHello()`, который выводит приветственное сообщение в стандартный поток ввода-вывода, и еще один метод экземпляра по имени `GetMsg()`, возвращающий внутреннюю

закрытую строку. По существу мы собираемся программно сгенерировать следующий тип класса:

```
// Этот класс будет создаваться во время выполнения с использованием
// пространства имен System.Reflection.Emit.
public class HelloWorld
{
  private string theMessage;
  HelloWorld() {}
  HelloWorld(string s) {theMessage = s;}
  public string GetMsg() {return theMessage;}
  public void SayHello()
  {
    System.Console.WriteLine("Hello from the HelloWorld class!");
  }
}
```

Создадим в Visual Studio новый проект консольного приложения по имени MyAsmBuilder, импортируем в него пространства имен System.Reflection, System.Reflection.Emit и System.Threading, после чего определим в классе Program статический метод по имени CreateMyAsm(). Этот единственный метод будет отвечать за решение следующих задач:

- определение характеристик динамической сборки (имя, версия и т.п.);
- реализация типа HelloClass;
- сохранение находящейся в памяти сборки в физическом файле.

Кроме того, обратите внимание, что метод CreateMyAsm() принимает единственный параметр типа System.AppDomain, который будет использоваться для получения доступа к объекту типа AssemblyBuilder, ассоциированному с текущим доменом приложения (домены приложений обсуждались в главе 17). Ниже показан полный код.

```
// Объект AppDomain отправляется вызывающим кодом.
public static void CreateMyAsm(AppDomain curAppDomain)
{
  // Установить общие характеристики сборки.
  AssemblyName assemblyName = new AssemblyName();
  assemblyName.Name = "MyAssembly";
  assemblyName.Version = new Version("1.0.0.0");

  // Создать новую сборку внутри текущего домена приложения.
  AssemblyBuilder assembly =
    curAppDomain.DefineDynamicAssembly(assemblyName,
    AssemblyBuilderAccess.Save);

  // Поскольку строится однофайловая сборка, имя модуля
  // будет таким же, как у сборки.
  ModuleBuilder module =
    assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");

  // Определить открытый класс по имени HelloWorld.
  TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld",
    TypeAttributes.Public);

  // Определить закрытую переменную-член типа String по имени theMessage.
  FieldBuilder msgField =
    helloWorldClass.DefineField("theMessage", Type.GetType("System.String"),
    FieldAttributes.Private);
```

```
// Создать специальный конструктор.
Type[] constructorArgs = new Type[1];
constructorArgs[0] = typeof(string);
ConstructorBuilder constructor =
  helloWorldClass.DefineConstructor(MethodAttributes.Public,
  CallingConventions.Standard,
  constructorArgs);
ILGenerator constructorIL = constructor.GetILGenerator();
constructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = typeof(object);
ConstructorInfo superConstructor =
  objectClass.GetConstructor(new Type[0]);
constructorIL.Emit(OpCodes.Call, superConstructor);
constructorIL.Emit(OpCodes.Ldarg_0);
constructorIL.Emit(OpCodes.Ldarg_1);
constructorIL.Emit(OpCodes.Stfld, msgField);
constructorIL.Emit(OpCodes.Ret);

// Создать стандартный конструктор.
helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);

// Создать метод GetMsg().
MethodBuilder getMsgMethod =
  helloWorldClass.DefineMethod("GetMsg", MethodAttributes.Public,
  typeof(string), null);
ILGenerator methodIL = getMsgMethod.GetILGenerator();
methodIL.Emit(OpCodes.Ldarg_0);
methodIL.Emit(OpCodes.Ldfld, msgField);
methodIL.Emit(OpCodes.Ret);

// Создать метод SayHello().
MethodBuilder sayHiMethod =
  helloWorldClass.DefineMethod("SayHello",
  MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();
methodIL.EmitWriteLine("Hello from the HelloWorld class!");
methodIL.Emit(OpCodes.Ret);

// Выпустить класс HelloWorld.
helloWorldClass.CreateType();

// (Дополнительно) сохранить сборку в файле.
assembly.Save("MyAssembly.dll");
}
```

Выпуск сборки и набора модулей

Тело метода начинается с установления минимального набора характеристик сборки с применением типов `AssemblyName` и `Version` (определенных в пространстве имен `System.Reflection`). Затем производится получение объекта типа `AssemblyBuilder` посредством метода `AppDomain.DefineDynamicAssembly()` уровня экземпляра (вспомните, что вызывающий код будет передавать ссылку на `AppDomain` методу `CreateMyAsm()`):

```
// Установить общие характеристики сборки.
// и получить доступ к объекту AssemblyBuilder.
public static void CreateMyAsm(AppDomain curAppDomain)
{
  AssemblyName assemblyName = new AssemblyName();
  assemblyName.Name = "MyAssembly";
  assemblyName.Version = new Version("1.0.0.0");
```

```
    // Создать новую сборку внутри текущего домена приложения.
    AssemblyBuilder assembly =
      curAppDomain.DefineDynamicAssembly(assemblyName,
      AssemblyBuilderAccess.Save);
    ...
}
```

Как видите, при вызове метода `AppDomain.DefineDynamicAssembly()` должен быть указан режим доступа к определяемой сборке; его наиболее распространенные значения представлены в табл. 18.10.

Таблица 18.10. Часто используемые значения перечисления `AssemblyBuilderAccess`

Значение	Описание
`ReflectionOnly`	Указывает, что динамическая сборка предназначена только для рефлексии
`Run`	Указывает, что динамическая сборка может выполняться в памяти, но не сохраняться на диске
`RunAndSave`	Указывает, что динамическая сборка может выполняться в памяти и сохраняться на диске
`Save`	Указывает, что динамическая сборка может сохраняться на диске, но не выполняться в памяти

Следующей задачей будет определение набора модулей для новой сборки. С учетом того, что сборка является однофайловой единицей, необходимо определить только один модуль. В случае построения многофайловой сборки с применением метода `DefineDynamicModule()` пришлось бы указывать необязательный второй параметр, который представляет имя заданного модуля (например, `myMod.dotnetmodule`). Тем не менее, когда создается однофайловая сборка, имя модуля идентично имени самой сборки. Метод `DefineDynamicModule()` возвращает ссылку на действительный объект типа `ModuleBuilder`:

```
// Однофайловая сборка.
ModuleBuilder module =
  assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");
```

Роль типа `ModuleBuilder`

Тип `ModuleBuilder` играет ключевую роль во время разработки динамических сборок. Как и можно было ожидать, `ModuleBuilder` поддерживает несколько членов, которые позволяют определять набор типов, содержащихся внутри модуля (классы, интерфейсы, структуры и т.д.), а также набор встроенных ресурсов (таблицы строк, изображения и т.п.). В табл. 18.11 кратко описаны некоторые методы, связанные с созданием. (Обратите внимание, что каждый метод возвращает объект связанного типа, который представляет подлежащий созданию тип.)

Таблица 18.11. Избранные методы типа `ModuleBuilder`

Метод	Описание
`DefineEnum()`	Используется для выпуска определения перечисления .NET
`DefineResource()`	Определяет управляемый встроенный ресурс, который должен храниться в данном модуле
`DefineType()`	Конструирует объект `TypeBuilder`, который позволяет определять типы значений, интерфейсы и типы классов (включая делегаты)

Основным членом класса ModuleBuilder является метод DefineType(). Кроме указания имени типа (в виде простой строки) с помощью перечисления System.Reflection.TypeAttributes можно описывать формат этого типа. В табл. 18.12 приведены избранные члены перечисления TypeAttributes.

Таблица 18.12. Избранные члены перечисления TypeAttributes

Член	Описание
Abstract	Указывает, что тип является абстрактным
Class	Указывает, что тип является классом
Interface	Указывает, что тип является интерфейсом
NestedAssembly	Указывает, что класс находится в области видимости сборки, поэтому доступен только методам внутри его сборки
NestedFamANDAssem	Указывает, что класс находится в области видимости сборки и семейства, поэтому доступен только методам, которые относятся к пересечению семейства и сборки
NestedFamily	Указывает, что класс находится в области видимости семейства, поэтому доступен только методам, которые содержатся внутри собственного типа и любых подтипов
NestedFamORAssem	Указывает, что класс находится в области видимости семейства или сборки, поэтому доступен только методам, которые относятся к объединению семейства и сборки
NestedPrivate	Указывает, что класс является вложенным и закрытым
NestedPublic	Указывает, класс является вложенным и открытым
NotPublic	Указывает класс, что класс не является открытым
Public	Указывает, что класс является открытым
Sealed	Указывает, что класс является конкретным и не может быть расширен
Serializable	Указывает, что класс может быть сериализирован

Выпуск типа HelloClass и строковой переменной-члена

Теперь, когда вы лучше понимаете роль метода ModuleBuilder.CreateType(), давайте посмотрим, как можно выпустить открытый тип класса HelloWorld и закрытую строковую переменную:

```
// Определить открытый класс по имени MyAssembly.HelloWorld.
TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld",
  TypeAttributes.Public);

// Определить закрытую переменную-член типа String по имени theMessage.
FieldBuilder msgField =
  helloWorldClass.DefineField("theMessage",
  Type.GetType("System.String"),
  FieldAttributes.Private);
```

Обратите внимание, что метод TypeBuilder.DefineField() предоставляет доступ к объекту типа FieldBuilder. В классе TypeBuilder также определены дополнительные методы, которые обеспечивают доступ к другим типам "построителей". Например, метод DefineConstructor() возвращает объект типа ConstructorBuilder, метод DefineProperty() — объект типа PropertyBuilder и т.д.

Выпуск конструкторов

Как упоминалось ранее, для определения конструктора текущего типа можно применять метод `TypeBuilder.DefineConstructor()`. Однако когда дело доходит до реализации конструктора `HelloClass`, в тело конструктора необходимо вставить низкоуровневый код CIL, который будет отвечать за присваивание входного параметра внутренней закрытой строке. Чтобы получить объект типа `ILGenerator`, понадобится вызвать метод `GetILGenerator()` из соответствующего типа "построителя" (в данном случае `ConstructorBuilder`).

Помещение кода CIL в реализацию членов осуществляется с помощью метода `Emit()` класса `ILGenerator`. В самом методе `Emit()` часто используется тип класса `OpCodes`, который открывает доступ к набору кодов операций CIL через свойства только для чтения. Например, свойство `OpCodes.Ret` обозначает возвращение из вызова метода, `OpCodes.Stfld` создает присваивание значения переменной-члену, а `OpCodes.Call` применяется для вызова заданного метода (конструктора базового класса в этом случае). Итак, логика для реализации конструктора будет выглядеть следующим образом:

```
// Создать специальный конструктор, принимающий
// единственный аргумент типа System.String.
Type[] constructorArgs = new Type[1];
constructorArgs[0] = typeof(string);
ConstructorBuilder constructor =
  helloWorldClass.DefineConstructor(MethodAttributes.Public,
  CallingConventions.Standard, constructorArgs);

// Выпустить необходимый код CIL для конструктора.
ILGenerator constructorIL = constructor.GetILGenerator();
constructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = typeof(object);
ConstructorInfo superConstructor = objectClass.GetConstructor(new Type[0]);
constructorIL.Emit(OpCodes.Call, superConstructor); // Вызвать конструктор
                                                    // базового класса.
// Загрузить в стек указатель this объекта.
constructorIL.Emit(OpCodes.Ldarg_0);

// Загрузить входной аргумент в стек и сохранить его в msgField.
constructorIL.Emit(OpCodes.Ldarg_1);
constructorIL.Emit(OpCodes.Stfld, msgField); //Присвоить значение полю msgField.
constructorIL.Emit(OpCodes.Ret);  // Возвращение.
```

Как теперь уже хорошо известно, в результате определения специального конструктора для типа стандартный конструктор молча удаляется. Чтобы снова определить конструктор без аргументов, нужно просто вызвать метод `DefineDefaultConstructor()` типа `TypeBuilder`:

```
// Вставить заново стандартный конструктор.
helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);
```

Этот единственный вызов выпускает стандартный код CIL, используемый для определения стандартного конструктора:

```
.method public hidebysig specialname rtspecialname
  instance void .ctor() cil managed
{
  .maxstack 1
  ldarg.0
  call instance void [mscorlib]System.Object::.ctor()
  ret
}
```

Выпуск метода SayHello()

Наконец, давайте исследуем процесс выпуска метода SayHello(). Первая задача связана с получением объекта типа MethodBuilder из переменной helloWorldClass. После этого можно определить сам метод и получить внутренний объект типа ILGenerator для вставки необходимых инструкций CIL:

```
// Создать метод SayHello.
MethodBuilder sayHiMethod =
  helloWorldClass.DefineMethod("SayHello",
  MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();

// Вывести строку на консоль.
methodIL.EmitWriteLine("Hello from the HelloWorld class!");
methodIL.Emit(OpCodes.Ret);
```

Здесь был определен открытый метод (т.к. указано значение MethodAttributes.Public), который не принимает параметров и ничего не возвращает (на что указывают значения null в вызове DefineMethod()). Также обратите внимание на вызов EmitWriteLine(). Посредством этого вспомогательного метода класса ILGenerator можно записать строку в стандартный поток вывода, приложив минимальные усилия.

Использование динамически сгенерированной сборки

Имея готовую логику для создания и сохранения сборки, осталось только разработать класс, который запустит эту логику. Определим в текущем проекте класс по имени AsmReader. Внутри Main() посредством метода Thread.GetDoMain() необходимо получить ссылку на текущий домен приложения, который будет применяться для размещения динамически созданной сборки. Благодаря такой ссылке появится возможность вызывать метод CreateMyAsm().

Чтобы сделать пример немного интереснее, после вызова CreateMyAsm() мы задействуем позднее связывание (см. главу 15) для загрузки созданной сборки в память и взаимодействия с членами класса HelloWorld. Модифицируем метод Main(), как показано ниже:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Amazing Dynamic Assembly Builder App
*****");

  // Получить домен приложения для текущего потока.
  AppDomain curAppDomain = Thread.GetDomain();

  // Создать динамическую сборку с помощью нашего вспомогательного метода.
  CreateMyAsm(curAppDomain);
  Console.WriteLine("-> Finished creating MyAssembly.dll.");

  // Загрузить новую сборку из файла.
  Console.WriteLine("-> Loading MyAssembly.dll from file.");
  Assembly a = Assembly.Load("MyAssembly");

  // Получить тип HelloWorld.
  Type hello = a.GetType("MyAssembly.HelloWorld");

  // Создать объект HelloWorld и вызвать корректный конструктор.
  Console.Write("-> Enter message to pass HelloWorld class: ");
  string msg = Console.ReadLine();
  object[] ctorArgs = new object[1];
  ctorArgs[0] = msg;
  object obj = Activator.CreateInstance(hello, ctorArgs);
```

```
    // Вызвать SayHello и отобразить возвращенную строку.
    Console.WriteLine("-> Calling SayHello() via late binding.");
    MethodInfo mi = hello.GetMethod("SayHello");
    mi.Invoke(obj, null);

    // Вызвать метод.
    mi = hello.GetMethod("GetMsg");
    Console.WriteLine(mi.Invoke(obj, null));
  }
```

В сущности, только что была построена сборка .NET, которая способна создавать и запускать другие сборки .NET во время выполнения. На этом исследование языка CIL и роли динамических сборок завершено. Данная глава должна была помочь углубить знания системы типов .NET, синтаксиса и семантики языка CIL, а также способа обработки кода компилятором C# в процессе его компиляции.

Исходный код. Проект `DynamicAsmBuilder` доступен в подкаталоге `Chapter_18`.

Резюме

В главе был представлен обзор синтаксиса и семантики языка CIL. В отличие от управляемых языков более высокого уровня, таких как C#, в CIL не просто определяется набор ключевых слов, а предоставляются директивы (используемые для определения конструкции сборки и ее типов), атрибуты (дополнительно уточняющие данные директивы) и коды операций (которые применяются для реализации членов типов).

Вы ознакомились с несколькими инструментами, связанными с программированием на CIL, и узнали, как изменять содержимое сборки .NET за счет добавления новых инструкций CIL, используя возвратное проектирование. Кроме того, вы изучили способы установления текущей (и ссылаемой сборки), пространств имен, типов и членов. Был рассмотрен простой пример построения библиотеки кода .NET и исполняемого файла с применением CIL и соответствующих инструментов командной строки.

Наконец, вы получили начальное представление о процессе создания *динамической сборки*. Используя пространство имен `System.Reflection.Emit`, сборку .NET можно определять в памяти во время выполнения. Вы видели, что работа с этим пространством имен требует знания семантики кода CIL. Хотя построение динамических сборок не является распространенной задачей при разработке большинства приложений .NET, это может быть полезно в случае создания инструментов поддержки и различных утилит для программирования.

ЧАСТЬ VI

Введение в библиотеки базовых классов .NET

В этой части

ГЛАВА 19

Многопоточное, параллельное и асинхронное программирование

Вряд ли многим нравится работать с приложением, которое притормаживает во время выполнения. Аналогично, никто не будет в восторге из-за того, что запуск какой-то задачи в приложении (возможно, по щелчку на элементе в панели инструментов) снижает отзывчивость других частей приложения. До выхода платформы .NET построение приложений, способных выполнять сразу несколько задач, обычно требовало написания сложного кода на языке C++, в котором использовались API-интерфейсы многопоточности Windows. К счастью, платформа .NET предложила несколько способов построения программного обеспечения, которое может делать сложные операции по уникальным путям выполнения, с намного меньшими сложностями.

Глава начинается с определения общей природы "многопоточного приложения". Затем мы снова обратимся к типу делегата .NET, чтобы исследовать его внутреннюю поддержку *асинхронных вызовов методов*. Как вы увидите, такой прием позволяет вызывать метод во вторичном потоке выполнения без необходимости в ручном создании или конфигурировании самого потока.

После этого будет представлено первоначальное пространство имен для многопоточности, которое поставляется, начиная с версии .NET 1.0, и называется `System.Threading`. Вы ознакомитесь с многочисленными типами (`Thread`, `ThreadStart` и т.д.), которые позволяют явно создавать дополнительные потоки выполнения и синхронизировать разделяемые ресурсы, обеспечивая совместное использование данных несколькими потоками в неизменчивой манере.

В оставшихся разделах главы будут рассматриваться последние три технологии, которые разработчики приложений .NET могут применять для построения многопоточного программного обеспечения, в частности библиотека Task Parallel Library (TPL), технология PLINQ (Parallel LINQ) и новые ключевые слова языка C#, связанные с асинхронной обработкой (`async` и `await`). Вы увидите, что эти средства помогают существенно упростить процесс создания отзывчивых многопоточных программных приложений.

Отношения между процессом, доменом приложения, контекстом и потоком

В главе 17 *поток* был определен как путь выполнения внутри исполняемого приложения. Хотя многие приложения .NET могут успешно и продуктивно работать, будучи однопоточными, первичный поток сборки (создаваемый средой CLR при выполнении метода `Main()`) в любое время может порождать вторичные потоки для выполнения дополнительных единиц работы. За счет создания дополнительных потоков можно строить более отзывчивые (но не обязательно быстрее выполняющиеся на одноядерных машинах) приложения.

Пространство имен `System.Threading` появилось в версии .NET 1.0 и предлагает один из подходов к построению многопоточных приложений. Главным типом в этом пространстве имен, пожалуй, можно назвать класс `Thread`, поскольку он представляет отдельный поток. Если необходимо программно получить ссылку на поток, который в текущий момент выполняет заданный член, нужно просто обратиться к статическому свойству `Thread.CurrentThread`:

```
static void ExtractExecutingThread()
{
  // Получить поток, который в настоящий момент выполняет этот метод.
  Thread currThread = Thread.CurrentThread;
}
```

Внутри платформы .NET *не существует* прямого соответствия "один к одному" между доменами приложений и потоками. В действительности заданный домен приложения может иметь многочисленные потоки, выполняющиеся в каждый конкретный момент времени. Более того, отдельный поток на протяжении своего времени жизни не ограничен единственным доменом приложения. Потоки могут пересекать границы доменов приложений, когда планировщик потоков Windows и среда CLR сочтут это подходящим.

Несмотря на то что активные потоки могут перемещаться между границами доменов приложений, каждый поток в любой момент времени может выполняться только внутри одного домена приложения (другими словами, одиночный поток не может выполнять работу в более чем одном домене приложения одновременно). Чтобы программно получить доступ к домену приложения, который обслуживает текущий поток, понадобится вызвать статический метод `Thread.GetDomain()`:

```
static void ExtractAppDomainHostingThread()
{
  // Получить домен приложения, обслуживающий текущий поток.
  AppDomain ad = Thread.GetDomain();
}
```

Одиночный поток в любой момент может быть также перенесен в определенный контекст, и он может перемещаться внутри нового контекста по прихоти среды CLR. Для получения текущего контекста, в котором выполняется поток, используется статическое свойство `Thread.CurrentContext` (которое возвращает объект `System.Runtime.Remoting.Contexts.Context`):

```
static void ExtractCurrentThreadContext()
{
  // Получить контекст, в котором работает текущий поток.
  Context ctx = Thread.CurrentContext;
}
```

Еще раз: за перемещение потоков между доменами приложений и контекстами отвечает среда CLR. Как разработчик приложений .NET, вы всегда остаетесь в блаженном неведении относительно того, где завершается каждый конкретный поток (или когда он в точности помещается внутрь новых границ). Тем не менее, вы должны быть осведомлены о разнообразных способах получения лежащих в основе примитивов.

Проблема параллелизма

Один из многих болезненных аспектов многопоточного программирования связан с ограниченным контролем над тем, как операционная система или среда CLR задействует потоки. Например, написав блок кода, который создает новый поток выполнения, нельзя гарантировать, что этот поток запустится немедленно. Взамен такой код только инструктирует операционную систему или CLR о необходимости запуска потока как можно скорее (что обычно происходит, когда планировщик потоков доберется до него).

Более того, учитывая, что потоки могут перемещаться между границами приложений и контекстов, как требуется среде CLR, вы должны представлять, какие аспекты приложения являются *изменчивыми в потоках* (например, подвергаются многопоточному доступу), а какие операции считаются *атомарными* (изменчивые в потоках операции опасны).

Чтобы проиллюстрировать проблему, давайте предположим, что поток вызывает метод специфичного объекта. Теперь представим, что этот поток приостановлен планировщиком потока, чтобы позволить другому потоку обратиться к тому же методу того же самого объекта.

Если исходный поток еще не полностью завершил свою операцию, то второй входящий поток может увидеть объект в частично модифицированном состоянии. В этот момент второй поток по существу читает фиктивные данные, что определенно может привести к очень странным (и трудно обнаруживаемым) ошибкам, которые еще труднее воспроизвести и отладить.

С другой стороны, атомарные операции в многопоточной среде всегда безопасны. К сожалению, в библиотеках базовых классов .NET есть лишь несколько гарантированно атомарных операций. Даже действие по присваиванию значения переменной-члену не является атомарным! Если только в документации .NET Framework 4.6 SDK специально не сказано об атомарности операции, то вы обязаны считать ее изменчивой в потоках и предпринимать меры предосторожности.

Роль синхронизации потоков

К этому моменту должно быть ясно, что многопоточные программы сами по себе довольно изменчивы, т.к. множество потоков могут оперировать разделяемыми ресурсами (более или менее) одновременно. Чтобы защитить ресурсы приложений от возможного повреждения, разработчики приложений .NET должны применять любое количество потоковых примитивов (таких как блокировки, мониторы, атрибут `[Synchronization]` или поддержка языковых ключевых слов) для управления доступом между выполняющимися потоками.

Хотя платформа .NET не может полностью скрыть сложности, связанные с построением надежных многопоточных приложений, сам процесс был значительно упрощен. Используя типы из пространства имен `System.Threading`, библиотеку Task Parallel Library (TPL) и ключевые слова `async` и `await` языка C#, можно работать с множеством потоков, прикладывая минимальные усилия.

Прежде чем погрузиться в детали пространства имен `System.Threading`, библиотеки TPL и ключевых слов `async` и `await` языка C#, мы начнем с выяснения того, каким образом можно применять тип делегата .NET для вызова метода в асинхронной манере.

В то время как, несомненно, верно то, что начиная с версии .NET 4.6 ключевые слова async и await предлагают более простую альтернативу асинхронным делегатам, по-прежнему важно знать способы взаимодействия с кодом, использующим этот подход (в производственной среде имеется масса кода, в котором применяются асинхронные делегаты).

Краткий обзор делегатов .NET

Вспомните, что делегат .NET по существу представляет собой безопасный в отношении типов объектно-ориентированный указатель на функцию. На объявление типа делегата .NET компилятор C# отвечает построением запечатанного класса, производного от System.MulticastDelegate (который, в свою очередь, унаследован от System.Delegate). Эти базовые классы предоставляют каждому делегату возможность поддерживать список адресов методов, которые могут быть вызваны в более позднее время. Рассмотрим следующий делегат BinaryOp, впервые определенный в главе 10:

```
// Тип делегата C#.
public delegate int BinaryOp(int x, int y);
```

Исходя из определения, тип BinaryOp может указывать на любой метод, который принимает два целых числа (по значению) в качестве аргументов и возвращает целое число. После компиляции сборка с определением делегата содержит полноценное определение класса, сгенерированного динамически на основе объявления делегата при компиляции проекта. В случае BinaryOp этот класс похож на показанный ниже класс, записанный посредством псевдокода:

```
public sealed class BinaryOp : System.MulticastDelegate
{
  public BinaryOp(object target, uint functionAddress);
  public int Invoke(int x, int y);
  public IAsyncResult BeginInvoke(int x, int y,
    AsyncCallback cb, object state);
  public int EndInvoke(IAsyncResult result);
}
```

Вспомните, что сгенерированный метод Invoke() используется для вызова методов, поддерживаемых объектом делегата, в *синхронной манере*. Следовательно, вызывающий поток (подобный первичному потоку приложения) вынужден ждать до тех пор, пока не завершится вызов делегата. Также вспомните, что в языке C# метод Invoke() не должен вызываться в коде напрямую, а может быть запущен косвенно, "за кулисами", когда применяется "нормальный" синтаксис вызова методов.

Рассмотрим следующий проект консольного приложения (SyncDelegateReview), в котором статический метод Add() вызывается в синхронном (т.е. блокирующем) режиме (не забудьте импортировать в файл кода C# пространство имен System.Threading, т.к. будет вызываться метод Thread.Sleep()):

```
namespace SyncDelegateReview
{
  public delegate int BinaryOp(int x, int y);

  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Synch Delegate Review *****");
```

```
    // Вывести идентификатор выполняющегося потока.
    Console.WriteLine("Main() invoked on thread {0}.",
      Thread.CurrentThread.ManagedThreadId);
    // Вызвать Add() в синхронном режиме.
    BinaryOp b = new BinaryOp(Add);
    // Можно было бы также написать b.Invoke(10, 10);
    int answer = b(10, 10);
    // Эти строки кода не выполнятся до тех пор,
    // пока не завершится метод Add().
    Console.WriteLine("Doing more work in Main()!");
    Console.WriteLine("10 + 10 is {0}.", answer);
    Console.ReadLine();
  }
  static int Add(int x, int y)
  {
    // Вывести идентификатор выполняющегося потока.
    Console.WriteLine("Add() invoked on thread {0}.",
      Thread.CurrentThread.ManagedThreadId);
    // Сделать паузу для моделирования длительной операции.
    Thread.Sleep(5000);
    return x + y;
  }
 }
}
```

Внутри метода `Add()` вызывается статический метод `Thread.Sleep()`, чтобы приостановить вызывающий поток приблизительно на 5 секунд для моделирования длительно выполняющейся задачи. Поскольку метод `Add()` вызывается в *синхронной* манере, метод `Main()` не будет выводить результат операции до тех пор, пока не завершится метод `Add()`.

Обратите внимание, что метод `Main()` получает доступ к текущему потоку (через свойство `Thread.CurrentThread`) и выводит идентификатор потока посредством свойства `ManagedThreadId`. Та же самая логика повторяется в статическом методе `Add()`. Как и можно было ожидать, учитывая, что вся работа в этом приложении выполняется исключительно в первичном потоке, на консоль выводится одно и то же значение идентификатора:

```
***** Synch Delegate Review *****
Main() invoked on thread 1.
Add() invoked on thread 1.
Doing more work in Main()!
10 + 10 is 20.
Press any key to continue . . .
```

Запустив программу, вы должны заметить пятисекундную задержку перед тем, как выполнится последний вызов `Console.WriteLine()` в методе `Main()`. Хотя многие (если не большинство) методов могут вызываться синхронно без болезненных последствий, при необходимости делегаты .NET можно инструктировать на вызов их методов асинхронным образом.

Исходный код. Проект `SyncDelegateReview` доступен в подкаталоге `Chapter_19`.

Асинхронная природа делегатов

Если тема многопоточности для вас в новинку, может возникнуть вопрос: что собой представляет *асинхронный* вызов метода? Как вам без сомнения известно, некоторые программные операции требуют времени. Несмотря на то что предыдущий метод `Add()` был исключительно иллюстративным, представьте себе, что вы строите однопоточное приложение, вызывающее метод удаленной веб-службы, который выполняет длительный запрос к базе данных, загружает крупный документ либо выводит 500 строк текста во внешний файл. Пока эти операции выполняются, приложение может выглядеть зависшим в течение некоторого периода времени. До завершения задачи все другие поведенческие аспекты программы (вроде выбора пунктов меню, щелчков в панели инструментов или вывода на консоль) приостанавливаются (что может раздражать пользователей).

По этой причине возникает вопрос: как сообщить делегату о вызове его метода в отдельном потоке выполнения, чтобы эмулировать "одновременное" выполнение многочисленных задач? К счастью, каждый тип делегата .NET автоматически оснащен такой возможностью. А еще лучше то, что для этого вы *не* обязаны углубляться в детали типов пространства имен `System.Threading` (хотя эти сущности могут вполне естественно работать рука об руку).

Методы BeginInvoke() и EndInvoke()

Когда компилятор C# обрабатывает ключевое слово `delegate`, он динамически генерирует класс, в котором определены два метода с именами `BeginInvoke()` и `EndInvoke()`. Учитывая определение делегата `BinaryOp`, эти методы будут прототипированы следующим образом:

```
public sealed class BinaryOp : System.MulticastDelegate
{
  ...
  // Используется для асинхронного вызова метода.
  public IAsyncResult BeginInvoke(int x, int y,
    AsyncCallback cb, object state);

  // Используется для извлечения возвращаемого значения вызванного метода.
  public int EndInvoke(IAsyncResult result);
}
```

Первый набор параметров, передаваемый методу `BeginInvoke()`, будет основан на формате делегата C# (два целых числа в случае `BinaryOp`). Последними двумя аргументами всегда будут `System.AsyncCallback` и `System.Object`. Вскоре вы узнаете о роли этих параметров, а пока передадим в каждом из них значение `null`. Также обратите внимание, что возвращаемое значение `EndInvoke()` является целочисленным, согласно возвращаемому типу `BinaryOp`, тогда как единственный параметр этого метода всегда имеет тип `IAsyncResult`.

Интерфейс System.IAsyncResult

Метод `BeginInvoke()` всегда возвращает объект, реализующий интерфейс `IAsyncResult`, а метод `EndInvoke()` требует единственный параметр совместимого с `IAsyncResult` типа. Совместимый с `IAsyncResult` объект, возвращаемый из `BeginInvoke()` — это по существу соединительный механизм, который позволяет вызывающему потоку получать результат вызова асинхронного метода в более позднее время посредством `EndInvoke()`. Интерфейс `IAsyncResult` (находящийся в пространстве имен `System`) определен следующим образом:

```
public interface IAsyncResult
{
  object AsyncState { get; }
  WaitHandle AsyncWaitHandle { get; }
  bool CompletedSynchronously { get; }
  bool IsCompleted { get; }
}
```

В простейшем случае прямого обращения к этим членам можно избежать. Все, что потребуется сделать — это сохранить совместимый с `IAsyncResult` объект, возвращенный методом `BeginInvoke()`, и передать его методу `EndInvoke()` при готовности к получению результата вызова метода. Как будет показано, члены совместимого с `IAsyncResult` объекта можно вызывать, когда требуется "более высокая вовлеченность" в процесс извлечения возвращаемого методом значения.

На заметку! Метод, возвращающий `void`, можно просто вызвать асинхронно и забыть. В таких случаях нет необходимости сохранять совместимый с `IAsyncResult` объект или вызывать `EndInvoke()`, т.к. нет возвращаемого значения, которое требуется получить.

Асинхронный вызов метода

Чтобы проинструктировать делегат `BinaryOp` о вызове метода `Add()` асинхронным образом, модифицируем логику предыдущего проекта (можно добавить код к имеющемуся проекту, но в коде примеров для главы доступен новый проект консольного приложения по имени `AsyncDelegate`). Обновите предыдущий метод `Main()`, как показано ниже:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Async Delegate Invocation *****");

  // Вывести идентификатор выполняющегося потока.
  Console.WriteLine("Main() invoked on thread {0}.",
    Thread.CurrentThread.ManagedThreadId);
  // Вызвать Add() во вторичном потоке.
  BinaryOp b = new BinaryOp(Add);
  IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);
  // Выполнить другую работу в первичном потоке...
  Console.WriteLine("Doing more work in Main()!");
  // По готовности получить результат выполнения метода Add().
  int answer = b.EndInvoke(iftAR);
  Console.WriteLine("10 + 10 is {0}.", answer);
  Console.ReadLine();
}
```

Запуск этого приложения приводит к выводу на консоль двух уникальных идентификатора потоков, поскольку в текущем домене приложения функционирует множество потоков:

```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Doing more work in Main()!
Add() invoked on thread 3.
10 + 10 is 20.
```

В дополнение к уникальным идентификаторам при выполнении этого приложения вы заметите, что сообщение `Doing more work in Main()!` выводится практически мгновенно, в то время как вторичный поток продолжает свою работу.

Синхронизация вызывающего потока

Внимательно проанализировав текущую реализацию Main(), можно обнаружить, что промежуток времени между вызовами BeginInvoke() и EndInvoke() однозначно меньше пяти секунд. Следовательно, как только сообщение Doing more work in Main()! выводится на консоль, вызывающий поток блокируется и ожидает завершения вторичного потока, чтобы получить результат вызова метода Add(). Таким образом, в действительности происходит еще один *синхронный вызов*.

```
static void Main(string[] args)
{
  ...
  BinaryOp b = new BinaryOp(Add);

  // После обработки следующего оператора вызывающий поток
  // блокируется, пока не будет завершен BeginInvoke().
  IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);

  // Этот вызов занимает намного меньше пяти секунд!
  Console.WriteLine("Doing more work in Main()!");

  // Снова происходит ожидание завершения другого потока!
  int answer = b.EndInvoke(iftAR);
  ...
}
```

Очевидно, что асинхронные делегаты утратили бы свою привлекательность, если бы вызывающий поток при определенных обстоятельствах мог блокироваться. Чтобы позволить вызывающему потоку выяснять, завершил ли работу асинхронно вызванный метод, в интерфейсе IAsyncResult предусмотрено свойство IsCompleted. С его помощью вызывающий поток может определять, действительно ли асинхронный вызов был завершен, прежде чем обращаться к методу EndInvoke().

Если метод еще не завершился, то свойство IsCompleted возвращает false, и вызывающий поток может продолжить заниматься своей работой. Когда IsCompleted возвращает true, вызывающий поток может получить результат в "наименее блокирующей манере". Взгляните на следующую модификацию метода Main():

```
static void Main(string[] args)
{
  ...
  BinaryOp b = new BinaryOp(Add);
  IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);

  // Это сообщение продолжит выводиться до тех пор,
  // пока не будет завершен метод Add().
  while(!iftAR.IsCompleted)
  {
    Console.WriteLine("Doing more work in Main()!");
    Thread.Sleep(1000);
  }

  // Теперь известно, что метод Add() завершен.
  int answer = b.EndInvoke(iftAR);
  ...
}
```

Здесь организован цикл, который продолжает выполнять оператор Console.WriteLine() до тех пор, пока не завершится вторичный поток. Когда это произойдет, можно получить результат выполнения метода Add(), точно зная, что он закончил ра-

боту. Вызов `Thread.Sleep(1000)` не обязателен для корректного функционирования этого конкретного приложения; однако, вынуждая первичный поток ожидать приблизительно секунду на каждой итерации, мы предотвращаем вывод на консоль слишком большого количества одного и того же сообщения. Ниже показан вывод (он может слегка отличаться, в зависимости от быстродействия машины и времени запуска потоков):

```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Doing more work in Main()!
Add() invoked on thread 3.
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
10 + 10 is 20.
```

Помимо свойства `IsCompleted` интерфейс `IAsyncResult` предлагает свойство `AsyncWaitHandle`, предназначенное для реализации более гибкой логики ожидания. Это свойство возвращает экземпляр типа `WaitHandle`, который открывает доступ к методу по имени `WaitOne()`. Преимущество использования `WaitHandle.WaitOne()` заключается в том, что можно указывать максимальное время ожидания. По истечении заданного времени метод `WaitOne()` возвратит `false`. Взгляните на следующий измененный цикл `while`, в котором больше не применяется вызов `Thread.Sleep()`:

```
while (!iftAR.AsyncWaitHandle.WaitOne(1000, true))
{
  Console.WriteLine("Doing more work in Main()!");
}
```

Хотя эти свойства интерфейса `IAsyncResult` и предоставляют способ синхронизации вызывающего потока, все же они не обеспечивают самый эффективный подход. Во многих отношениях свойство `IsCompleted` похоже на надоедливого менеджера (или коллегу), который постоянно спрашивает: "Вы уже сделали это?". К счастью, делегаты предлагают несколько дополнительных (и более элегантных) приемов получения результата из метода, который был вызван асинхронным образом.

Исходный код. Проект `AsyncDelegate` доступен в подкаталоге `Chapter_19`.

Роль делегата `AsyncCallback`

Вместо опроса делегата с целью определения, завершился ли асинхронно вызванный метод, было бы более эффективно заставить вторичный поток информировать вызывающий поток о завершении выполнения задачи. Чтобы сделать такое поведение возможным, понадобится передать методу `BeginInvoke()` в качестве параметра экземпляр делегата `System.AsyncCallback`; до сих пор для этого параметра указывалось значение `null`. Тем не менее, когда предоставляется объект `AsyncCallback`, делегат будет автоматически вызывать указанный метод по завершении асинхронного вызова.

На заметку! Метод обратного вызова будет вызван во вторичном потоке, а не в первичном. Это имеет важные последствия для потоков внутри графического пользовательского интерфейса (WPF или Windows Forms), т.к. элементы управления привязаны к потоку, в котором они были созданы, и могут обрабатываться только в нем. При рассмотрении библиотеки TPL и новых ключевых слов `async` и `await` языка C# в .NET 4.5 далее в главе будут представлены некоторые примеры работы потоков из графического пользовательского интерфейса.

Как и любой делегат, AsyncCallback может вызывать только методы, соответствующие определенному шаблону, которым в данном случае является метод, принимающий единственный параметр IAsyncResult и ничего не возвращающий:

```
// Целевые методы AsyncCallback должны соответствовать следующему шаблону.
void MyAsyncCallbackMethod(IAsyncResult itfAR)
```

Предположим, что есть еще один проект консольного приложения (AsyncCallbackDelegate), в котором используется делегат BinaryOp. Однако на этот раз мы не будем опрашивать делегат с целью выяснения, завершился ли метод Add(). Взамен мы определим статический метод по имени AddComplete() для получения уведомления о том, что асинхронный вызов завершен. Также в примере применяется булевское статическое поле уровня класса, которое служит для удержания в активном состоянии первичного потока Main(), пока не завершится вторичный поток.

На заметку! Использование булевской переменной в данном примере, строго говоря, не является безопасным в отношении потоков, поскольку к ее значению имеют доступ два разных потока. В рассматриваемом примере это допустимо; тем не менее, запомните в качестве *очень* важного эмпирического правила: вы должны обеспечивать блокировку данных, разделяемых между несколькими потоками. Вы увидите, как это делается, далее в главе.

```
namespace AsyncCallbackDelegate
{
  public delegate int BinaryOp(int x, int y);

  class Program
  {
    private static bool isDone = false;

    static void Main(string[] args)
    {
      Console.WriteLine("***** AsyncCallbackDelegate Example *****");
      Console.WriteLine("Main() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
      BinaryOp b = new BinaryOp(Add);
      IAsyncResult iftAR = b.BeginInvoke(10, 10,
        new AsyncCallback(AddComplete), null);

      // Предположим, что здесь делается какая-то другая работа...
      while (!isDone)
      {
        Thread.Sleep(1000);
        Console.WriteLine("Working....");
      }
      Console.ReadLine();
    }
    static int Add(int x, int y)
    {
      Console.WriteLine("Add() invoked on thread {0}.",
      Thread.CurrentThread.ManagedThreadId);
      Thread.Sleep(5000);
      return x + y;
    }
    static void AddComplete(IAsyncResult itfAR)
    {
      Console.WriteLine("AddComplete() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
```

```
      Console.WriteLine("Your addition is complete");
      isDone = true;
    }
  }
}
```

И снова статический метод `AddComplete()` будет вызван делегатом `AsyncCallback` по завершении метода `Add()`. Запустив программу, можно убедиться, что именно вторичный поток вызывает `AddComplete()`:

```
***** AsyncCallbackDelegate Example *****
Main() invoked on thread 1.
Add() invoked on thread 3.
Working....
Working....
Working....
Working....
Working....
AddComplete() invoked on thread 3.
Your addition is complete
```

Подобно другим примерам, приведенным в настоящей главе, вывод может несколько отличаться. На самом деле после завершения работы метода `AddComplete()` может появиться только одно финальное сообщение `Working....` Это просто побочный эффект односекундной задержки в `Main()`.

Роль класса `AsyncResult`

В настоящий момент метод `AddComplete()` не выводит на консоль действительный результат операции (сложение двух чисел). Причина с том, что целевой метод делегата `AsycnCallback` (`AsyncResult()` в этом примере) не имеет доступа к исходному делегату `BinaryOp`, созданному в контексте `Main()`, а потому вызывать `EndInvoke()` из `AddComplete()` нельзя!

В то время как переменную `BinaryOp` можно было бы просто объявить как статический член класса, чтобы позволить обоим методам обращаться к одному и тому же объекту, более элегантное решение предусматривает применение входного параметра `IAsyncResult`. Входной параметр `IAsyncResult`, передаваемый целевому методу делегата `AsyncCallback` — это в действительности экземпляр класса `AsyncResult` (обратите внимание на отсутствие префикса `I` в имени), определенного в пространстве имен `System.Runtime.Remoting.Messaging`. Свойство `AsyncDelegate` возвращает ссылку на исходный асинхронный делегат, который был создан где-то в другом месте.

Следовательно, чтобы получить ссылку на объект делегата `BinaryOp`, размещенный внутри `Main()`, нужно просто привести экземпляр `System.Object`, возвращенный свойством `AsyncDelegate`, к типу `BinaryOp`. В этот момент можно запустить `EndInvoke()`, как и ожидалось:

```
// Не забудьте импортировать пространство имен System.Runtime.Remoting.Messaging!
static void AddComplete(IAsyncResult itfAR)
{
  Console.WriteLine("AddComplete() invoked on thread {0}.",
    Thread.CurrentThread.ManagedThreadId);
  Console.WriteLine("Your addition is complete");
  // Теперь получить результат.
  AsyncResult ar = (AsyncResult)itfAR;
  BinaryOp b = (BinaryOp)ar.AsyncDelegate;
  Console.WriteLine("10 + 10 is {0}.", b.EndInvoke(itfAR));
  isDone = true;
}
```

Передача и получение специальных данных состояния

Финальным аспектом асинхронных делегатов, который необходимо обсудить, является последний аргумент метода `BeginInvoke()` (который до сих пор был `null`). Этот параметр позволяет передавать дополнительную информацию о состоянии методу обратного вызова из первичного потока. Поскольку упомянутый аргумент прототипирован как `System.Object`, в нем можно передавать данные любого типа при условии, что методу обратного вызова известно, чего ожидать. В целях демонстрации предположим, что первичный поток желает передать методу `AddComplete()` специальное текстовое сообщение:

```
static void Main(string[] args)
{
  ...
  IAsyncResult iftAR = b.BeginInvoke(10, 10,
    new AsyncCallback(AddComplete),
    "Main() thanks you for adding these numbers.");
  ...
}
```

Для получения этих данных внутри метода `AddComplete()` используется свойство `AsyncState` входного параметра `IAsyncResult`. Обратите внимание, что здесь потребуется явное приведение; следовательно, первичный и вторичный потоки должны согласовать между собой тип, возвращаемый `AsyncState`:

```
static void AddComplete(IAsyncResult itfAR)
{
  ...
  // Получить информационный объект и привести его к типу string.
  string msg = (string)itfAR.AsyncState;
  Console.WriteLine(msg);
  isDone = true;
}
```

Ниже показан вывод последней модификации примера:

```
***** AsyncCallbackDelegate Example *****
Main() invoked on thread 1.
Add() invoked on thread 3.
Working....
Working....
Working....
Working....
Working....
AddComplete() invoked on thread 3.
Your addition is complete
10 + 10 is 20.
Main() thanks you for adding these numbers.
```

Теперь, когда вы понимаете, каким образом применять делегат .NET для автоматического запуска вторичного потока выполнения с целью обработки асинхронного вызова метода, внимание можно переключить непосредственно на взаимодействие с потоками, используя пространство имен `System.Threading`. Вспомните, что данное пространство имен было первоначальным API-интерфейсом для реализации многопоточности .NET, который поставлялся еще в версии 1.0.

Исходный код. Проект `AsyncCallbackDelegate` доступен в подкаталоге `Chapter_19`.

Пространство имен System.Threading

В рамках платформы .NET пространство имен `System.Threading` предлагает несколько типов, которые дают возможность напрямую конструировать многопоточные приложения. В дополнение к типам, позволяющим взаимодействовать с отдельным потоком CLR, в этом пространстве имен определены типы, которые открывают доступ к пулу потоков, обслуживаемому CLR, простому (не связанному с графическим пользовательским интерфейсом) классу `Timer` и многочисленным типам, применяемым для предоставления синхронизированного доступа к разделяемым ресурсам. В табл. 19.1 перечислены некоторые важные члены пространства имен `System.Threading`. (За полными сведениями обращайтесь в документацию .NET Framework 4.6 SDK.)

Таблица 19.1. Основные типы пространства имен System.Threading

Тип	Назначение
`Interlocked`	Этот тип предоставляет атомарные операции для переменных, разделяемых между несколькими потоками
`Monitor`	Этот тип обеспечивает синхронизацию потоковых объектов, используя блокировки и ожидания/сигналы. Ключевое слово `lock` языка C# "за кулисами" применяет объект `Monitor`
`Mutex`	Этот примитив синхронизации может использоваться для синхронизации между границами доменов приложений
`ParameterizedThreadStart`	Этот делегат позволяет потоку вызывать методы, принимающие произвольное количество аргументов
`Semaphore`	Этот тип позволяет ограничивать количество потоков, которые могут иметь доступ к ресурсу или к определенному типу ресурсов одновременно
`Thread`	Этот тип представляет поток, выполняемый в среде CLR. С применением этого типа можно порождать дополнительные потоки в исходном домене приложения
`ThreadPool`	Этот тип позволяет взаимодействовать с поддерживаемым средой CLR пулом потоков внутри заданного процесса
`ThreadPriority`	Это перечисление представляет уровень приоритета потока (`Highest`, `Normal` и т.д.)
`ThreadStart`	Этот делегат позволяет указывать метод для вызова в заданном потоке. В отличие от делегата `ParametrizedThreadStart`, целевые методы `ThreadStart` всегда должны иметь один и тот же прототип
`ThreadState`	Это перечисление задает допустимые состояния потока (`Running`, `Aborted` и т.д.)
`Timer`	Этот тип предоставляет механизм выполнения метода через указанные интервалы
`TimerCallback`	Этот тип делегата используется в сочетании с типами `Timer`

Класс System.Threading.Thread

Класс `Thread` является самым элементарным из всех типов в пространстве имен `System.Threading`. Он представляет объектно-ориентированную оболочку вокруг заданного пути выполнения внутри отдельного домена приложения. В этом классе опре-

делено несколько методов (статических и уровня экземпляра), которые позволяют создавать новые потоки внутри текущего домена приложения, а также приостанавливать, останавливать и уничтожать указанный поток. Список основных статических членов приведен в табл. 19.2.

Таблица 19.2. Основные статические члены типа `Thread`

Статический член	Назначение
`CurrentContext`	Это свойство только для чтения возвращает контекст, в котором в текущий момент выполняется поток
`CurrentThread`	Это свойство только для чтения возвращает ссылку на текущий выполняемый поток
`GetDomain()` `GetDomainID()`	Эти методы возвращают ссылку на текущий домен приложения либо идентификатор домена, в котором выполняется текущий поток
`Sleep()`	Этот метод приостанавливает текущий поток на указанное время

Класс `Thread` также поддерживает члены уровня экземпляра, часть которых описана в табл. 19.3.

Таблица 19.3. Избранные члены уровня экземпляра типа `Thread`

Член уровня экземпляра	Назначение
`IsAlive`	Возвращает булевское значение, указывающее на то, запущен ли поток (и пока еще не прекращен или не отменен)
`IsBackground`	Получает или устанавливает значение, которое указывает, является ли данный поток фоновым (более подробно это объясняется далее в главе)
`Name`	Позволяет установить дружественное текстовое имя потока
`Priority`	Получает или устанавливает приоритет потока, который может принимать значение из перечисления `ThreadPriority`
`ThreadState`	Получает состояние данного потока, которое может принимать значение из перечисления `ThreadState`
`Abort()`	Указывает среде CLR на необходимость как можно более скорого прекращения работы потока
`Interrupt()`	Прерывает (например, приостанавливает) текущий поток на подходящий период ожидания
`Join()`	Блокирует вызывающий поток до тех пор, пока указанный поток (тот, на котором вызван метод `Join()`) не завершится
`Resume()`	Возобновляет выполнение ранее приостановленного потока
`Start()`	Указывает среде CLR на необходимость как можно более скорого запуска потока
`Suspend()`	Приостанавливает поток. Если поток уже приостановлен, то вызов `Suspend()` не оказывает никакого действия

На заметку! Прерывание или приостановка активного потока обычно считается плохой идеей. В таком случае есть шанс (хотя и небольшой), что поток может допустить "утечку" своей рабочей нагрузки.

Получение статистики о текущем потоке выполнения

Вспомните, что точка входа исполняемой сборки (т.е. метод `Main()`) запускается в первичном потоке выполнения. Чтобы проиллюстрировать базовое применение типа `Thread`, предположим, что имеется новый проект консольного приложения по имени `ThreadStats`. Как вам известно, статическое свойство `Thread.CurrentThread` извлекает объект `Thread`, который представляет поток, выполняющийся в текущий момент. Получив текущий поток, можно вывести разнообразные статистические сведения о нем:

```
// Не забудьте импортировать пространство имен System.Threading.
static void Main(string[] args)
{
  Console.WriteLine("***** Primary Thread stats *****\n");

  // Получить имя текущего потока.
  Thread primaryThread = Thread.CurrentThread;
  primaryThread.Name = "ThePrimaryThread";

  // Показать детали обслуживающего домена приложения и контекста.
  Console.WriteLine("Name of current AppDomain: {0}",
    Thread.GetDomain().FriendlyName);  // имя текущего домена приложения
  Console.WriteLine("ID of current Context: {0}",
    Thread.CurrentContext.ContextID);  // идентификатор текущего контекста

  // Вывести некоторую статистику о текущем потоке.
  Console.WriteLine("Thread Name: {0}",
    primaryThread.Name);               // имя потока
  Console.WriteLine("Has thread started?: {0}",
    primaryThread.IsAlive);            // запущен ли поток
  Console.WriteLine("Priority Level: {0}",
    primaryThread.Priority);           // приоритет потока
  Console.WriteLine("Thread State: {0}",
    primaryThread.ThreadState);        // состояние потока
  Console.ReadLine();
}
```

Вот как выглядит вывод:

```
***** Primary Thread stats *****
Name of current AppDomain: ThreadStats.exe
ID of current Context: 0
Thread Name: ThePrimaryThread
Has thread started?: True
Priority Level: Normal
Thread State: Running
```

Свойство Name

Хотя показанный выше код более или менее самоочевиден, обратите внимание, что класс `Thread` поддерживает свойство по имени `Name`. Если значение `Name` не было установлено, то будет возвращаться пустая строка. Однако назначение конкретному объекту `Thread` дружественного имени может значительно упростить отладку. Во время сеанса отладки в Visual Studio можно открыть окно Threads (Потоки), выбрав пункт меню Debug⇨Windows⇨Threads (Отладка⇨Окна⇨Потоки). На рис. 19.1 можно заметить, что это окно позволяет быстро идентифицировать поток, который нужно диагностировать.

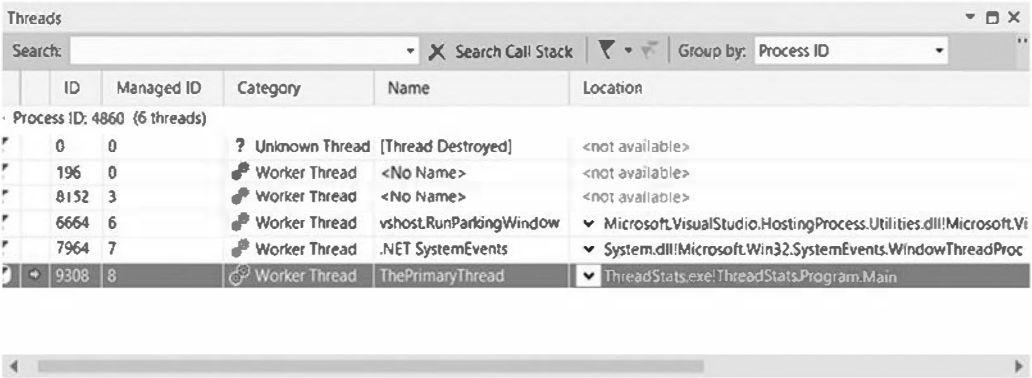


Рис. 19.1. Отладка потока в Visual Studio

Свойство `Priority`

Далее обратите внимание, что в типе `Thread` определено свойство по имени `Priority`. По умолчанию все потоки имеют уровень приоритета `Normal`. Тем не менее, в любой момент жизненного цикла потока это можно изменить, используя свойство `Thread.Priority` и связанное с ним перечисление `System.Threading.ThreadPriority`:

```
public enum ThreadPriority
{
  Lowest,
  BelowNormal,
  Normal,  // Стандартное значение.
  AboveNormal,
  Highest
}
```

В случае присваивания уровню приоритета потока значения, отличного от стандартного (`ThreadPriority.Normal`), помните об отсутствии прямого контроля над тем, когда планировщик потоков будет переключать потоки между собой. В действительности уровень приоритета потока предоставляет среде CLR подсказку относительно важности действия потока. Таким образом, поток с уровнем приоритета `ThreadPriority.Highest` не обязательно гарантированно получит наивысший приоритет.

Опять-таки, если планировщик потоков занят решением определенной задачи (например, синхронизацией объекта, переключением потоков либо их перемещением), то уровень приоритета, скорее всего, будет соответствующим образом изменен. Однако при прочих равных условиях среда CLR прочитает эти значения и проинструктирует планировщик потоков о том, как лучше выделять кванты времени. Потоки с идентичными уровнями приоритета должны получать одинаковое количество времени на выполнение своей работы.

В большинстве случаев редко (если вообще когда-либо) возникает необходимость напрямую изменять уровень приоритета потока. Теоретически можно так повысить уровень приоритета набора потоков, что вообще предотвратить выполнение низкоприоритетных потоков с их запрошенными уровнями (поэтому соблюдайте осторожность).

Исходный код. Проект `ThreadStats` доступен в подкаталоге `Chapter_19`.

Ручное создание вторичных потоков

Когда вы хотите программно создать дополнительные потоки для выполнения какой-то единицы работы, во время применения типов из пространства имен `System.Threading` следуйте представленному ниже предсказуемому процессу.

1. Создайте метод, который будет служить точкой входа для нового потока.
2. Создайте новый делегат `ParametrizedThreadStart` (или `ThreadStart`), передав его конструктору адрес метода, который был определен на шаге 1.
3. Создайте объект `Thread`, передав конструктору делегат `ParametrizedThreadStart`/`ThreadStart` в качестве аргумента.
4. Установите начальные характеристики потока (имя, приоритет и т.д.).
5. Вызовите метод `Thread.Start()`. Это приведет к как можно более быстрому запуску потока для метода, на который ссылается делегат, созданный на шаге 2.

Согласно шагу 2 для указания на метод, который будет выполняться во вторичном потоке, можно использовать два разных типа делегата. Делегат `ThreadStart` может указывать на любой метод, который не принимает аргументов и ничего не возвращает. Этот делегат может быть полезен, когда метод предназначен просто для запуска в фоновом режиме без дальнейшего взаимодействия с ним.

Очевидное ограничение `ThreadStart` связано с невозможностью передавать ему параметры для обработки. Тем не менее, тип делегата `ParametrizedThreadStart` позволяет передать единственный параметр типа `System.Object`. Учитывая, что с помощью `System.Object` представляется все, что угодно, посредством специального класса или структуры можно передавать любое количество параметров. Однако имейте в виду, что делегат `ParametrizedThreadStart` может указывать только на методы, возвращающие `void`.

Работа с делегатом `ThreadStart`

Чтобы проиллюстрировать процесс построения многопоточного приложения (а также его полезность), создадим проект консольного приложения по имени `SimpleMultiThreadApp`, которое позволяет конечному пользователю выбирать, будет приложение выполнять свою работу в единственном первичном потоке или же разделит рабочую нагрузку с применением двух отдельных потоков выполнения.

После импортирования пространства имен `System.Threading` определяется метод для выполнения работы (возможного) вторичного потока. Чтобы сосредоточиться на механизме построения многопоточных программ, этот метод будет просто выводить на консоль последовательность чисел, делая на каждом шаге паузу примерно в 2 секунды. Ниже показано полное определение класса `Printer`:

```
public class Printer
{
  public void PrintNumbers()
  {
    // Вывести информацию о потоке.
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);

    // Вывести числа.
    Console.Write("Your numbers: ");
    for(int i = 0; i < 10; i++)
    {
      Console.Write("{0}, ", i);
```

```
      Thread.Sleep(2000);
    }
    Console.WriteLine();
  }
}
```

Внутри метода `Main()` пользователю сначала предлагается решить, сколько потоков будет использоваться для выполнения работы приложения: один или два. Если пользователь запрашивает один поток, то нужно просто вызвать метод `PrintNumbers()` в первичном потоке. Тем не менее, когда пользователь выбирает два потока, понадобится создать делегат `ThreadStart`, указывающий на `PrintNumbers()`, передать объект делегата конструктору нового объекта `Thread` и вызвать метод `Start()` для информирования среды CLR о том, что данный поток готов к обработке.

Первым делом установим ссылку на сборку `System.Windows.Forms.dll` (а также импортируем пространство имен `System.Windows.Forms`) и отобразим сообщение в `Main()` с применением метода `MessageBox.Show()` (причина такого решения станет ясной после запуска программы). Вот полная реализация `Main()`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Amazing Thread App *****\n");
  Console.Write("Do you want [1] or [2] threads? ");
  string threadCount = Console.ReadLine();  // Запрос количества потоков
  // Назначить имя текущему потоку.
  Thread primaryThread = Thread.CurrentThread;
  primaryThread.Name = "Primary";
  // Вывести информацию о потоке.
  Console.WriteLine("-> {0} is executing Main()",
  Thread.CurrentThread.Name);
  // Создать рабочий класс.
  Printer p = new Printer();
  switch(threadCount)
  {
    case "2":
      // Создать поток.
      Thread backgroundThread =
        new Thread(new ThreadStart(p.PrintNumbers));
      backgroundThread.Name = "Secondary";
      backgroundThread.Start();
    break;
    case "1":
      p.PrintNumbers();
    break;
    default:
      Console.WriteLine("I don't know what you want...you get 1 thread.");
      goto case "1";  // Переход к варианту с одним потоком
  }
  // Выполнить некоторую дополнительную работу.
  MessageBox.Show("I'm busy!", "Work on main thread...");
  Console.ReadLine();
}
```

Если теперь запустить программу с одним потоком, то обнаружится, что финальное окно сообщения не будет отображать сообщение, пока вся последовательность чисел не выведется на консоль. Поскольку после вывода каждого числа установлена пауза около 2 секунд, это создаст не особенно приятное впечатление у конечного пользователя.

Однако в случае выбора двух потоков окно сообщения отображается немедленно, потому что для вывода чисел на консоль выделен отдельный объект Thread.

Исходный код. Проект SimpleMultiThreadApp доступен в подкаталоге Chapter_19.

Работа с делегатом ParametrizedThreadStart

Вспомните, что делегат ThreadStart может указывать только на методы, которые возвращают void и не принимают аргументов. В некоторых случаях это подходит, но если нужно передать данные методу, выполняющемуся во вторичном потоке, то придется использовать тип делегата ParametrizedThreadStart. В целях иллюстрации давайте воссоздадим логику проекта AsyncCallbackDelegate, разработанного ранее в главе, но на этот раз применим тип делегата ParameterizedThreadStart.

Для начала создадим новый проект консольного приложения по имени AddWithThreads и импортируем пространство имен System.Threading. С учетом того, что делегат ParametrizedThreadStart может указывать на любой метод, принимающий параметр типа System.Object, построим специальный тип, который содержит числа, подлежащие сложению:

```
class AddParams
{
  public int a, b;
  public AddParams(int numb1, int numb2)
  {
    a = numb1;
    b = numb2;
  }
}
```

Далее создадим в классе Program статический метод, который принимает параметр AddParams и выводит на консоль сумму двух чисел:

```
static void Add(object data)
{
  if (data is AddParams)
  {
    Console.WriteLine("ID of thread in Add(): {0}",
      Thread.CurrentThread.ManagedThreadId);
    AddParams ap = (AddParams)data;
    Console.WriteLine("{0} + {1} is {2}",
      ap.a, ap.b, ap.a + ap.b);
  }
}
```

Код внутри Main() прямолинеен. Просто вместо типа ThreadStart используется ParametrizedThreadStart:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Adding with Thread objects *****");
  Console.WriteLine("ID of thread in Main(): {0}",
    Thread.CurrentThread.ManagedThreadId);

  // Создать объект AddParams для передачи вторичному потоку.
  AddParams ap = new AddParams(10, 10);
  Thread t = new Thread(new ParameterizedThreadStart(Add));
  t.Start(ap);
```

```
  // Подождать, пока другой поток завершится.
  Thread.Sleep(5);
  Console.ReadLine();
}
```

Класс AutoResetEvent

В нескольких первых примерах для информирования первичного потока о необходимости подождать, пока вторичный поток не завершится, применялся ряд грубых способов. Во время исследования асинхронных делегатов в качестве переключателя использовалась простая булевская переменная; тем не менее, такое решение не является рекомендуемым, т.к. оба потока имеют доступ к одному и тому же элементу данных, что может привести к его повреждению. Более безопасной, хотя все еще неудобной альтернативой будет вызов метода `Thread.Sleep()` с фиксированным периодом времени. Проблема здесь в том, что нет желания ожидать дольше, чем необходимо.

Простой и безопасный к потокам способ заставить один поток ожидать, пока не завершится другой поток, предусматривает применение класса `AutoResetEvent`. В потоке, который должен ожидать (таком как метод `Main()`), создадим экземпляр `AutoResetEvent` и передадим его конструктору значение `false`, указав, что уведомления пока не было. Затем в точке, где требуется ожидать, вызовем метод `WaitOne()`. Ниже приведен модифицированный класс `Program`, который делает все это с использованием статической переменной-члена `AutoResetEvent`:

```
class Program
{
  private static AutoResetEvent waitHandle = new AutoResetEvent(false);
  static void Main(string[] args)
  {
    Console.WriteLine("***** Adding with Thread objects *****");
    Console.WriteLine("ID of thread in Main(): {0}",
      Thread.CurrentThread.ManagedThreadId);
    AddParams ap = new AddParams(10, 10);
    Thread t = new Thread(new ParameterizedThreadStart(Add));
    t.Start(ap);
    // Ожидать, пока не поступит уведомление!
    waitHandle.WaitOne();
    Console.WriteLine("Other thread is done!");
    Console.ReadLine();
  }
  ...
}
```

Когда другой поток завершит свою работу, он вызовет метод `Set()` на том же экземпляре типа `AutoResetEvent`:

```
static void Add(object data)
{
  if (data is AddParams)
  {
    Console.WriteLine("ID of thread in Add(): {0}",
      Thread.CurrentThread.ManagedThreadId);
    AddParams ap = (AddParams)data;
    Console.WriteLine("{0} + {1} is {2}",
      ap.a, ap.b, ap.a + ap.b);
    // Сообщить другому потоку о том, что работа завершена.
    waitHandle.Set();
  }
}
```

Исходный код. Проект `AddWithThreads` доступен в подкаталоге `Chapter_19`.

Потоки переднего плана и фоновые потоки

Теперь, когда вы знаете, как программно создавать новые потоки выполнения с применением типов из пространства имен `System.Threading`, давайте формализуем разницу между *потоками переднего плана* и *фоновыми потоками*.

- *Потоки переднего плана* имеют возможность предохранять текущее приложение от завершения. Среда CLR не будет прекращать работу приложения (скажем, выгружая текущий домен приложения) до тех пор, пока не будут завершены все потоки переднего плана.
- *Фоновые потоки* (иногда называемые *потоками-демонами*) воспринимаются средой CLR как расширяемые пути выполнения, которые в любой момент времени могут быть проигнорированы (даже если они заняты выполнением некоторой части работы). Таким образом, если при выгрузке домена приложения все потоки переднего плана завершены, то все фоновые потоки автоматически уничтожаются.

Важно отметить, что потоки переднего плана и фоновые потоки — это *не* синонимы первичных и рабочих потоков. По умолчанию каждый поток, создаваемый посредством метода `Thread.Start()`, автоматически становится потоком переднего плана. В итоге домен приложения не выгрузится до тех пор, пока все потоки выполнения не завершат свои единицы работы. В большинстве случаев именно такое поведение и требуется.

Ради доказательства сделанных утверждений предположим, что метод `Printer.PrintNumbers()` необходимо вызвать во вторичном потоке, который должен вести себя как фоновый. Это означает, что метод, указываемый типом `Thread` (через делегат `ThreadStart` или `ParametrizedThreadStart`), должен обладать возможностью безопасного останова, как только все потоки переднего плана закончат свою работу. Конфигурирование такого потока сводится просто к установке свойства `IsBackground` в `true`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Background Threads *****\n");
  Printer p = new Printer();
  Thread bgroundThread =
    new Thread(new ThreadStart(p.PrintNumbers));

  // Теперь это фоновый поток.
  bgroundThread.IsBackground = true;
  bgroundThread.Start();
}
```

Обратите внимание, что в этом методе `Main()` *не* делается вызов `Console.ReadLine()`, чтобы заставить окно консоли оставаться видимым, пока не будет нажата клавиша <Enter>. Таким образом, после запуска приложение немедленно прекращается, потому что объект `Thread` сконфигурирован как фоновый поток. Учитывая, что метод `Main()` инициирует создание первичного потока *переднего плана*, как только логика метода `Main()` завершится, домен приложения будет выгружен, прежде чем вторичный поток сможет закончить свою работу.

Однако если закомментировать строку, которая устанавливает свойство `IsBackground` в `true`, то обнаружится, что на консоль выводятся все числа, т.к. все потоки переднего плана должны завершить свою работу перед тем, как домен приложения будет выгружен из обслуживающего процесса.

По большей части конфигурировать поток для функционирования в фоновом режиме может быть удобно, когда интересующий рабочий поток выполняет некритичную задачу, потребность в которой исчезает после завершения главной задачи программы. Например, можно было бы построить приложение, которое проверяет сервер электронной почты каждые несколько минут на предмет поступления новых сообщений, обновляет текущий прогноз погоды или решает какие-то другие некритичные задачи.

Проблемы параллелизма

При построении многопоточных приложений необходимо гарантировать, что любой фрагмент разделяемых данных защищен от возможности изменения со стороны сразу нескольких потоков. Поскольку все потоки в домене приложения имеют параллельный доступ к разделяемым данным приложения, вообразите, что может произойти, если множество потоков одновременно обратятся к одному и тому же элементу данных. Так как планировщик потоков случайным образом приостанавливает их работу, что если поток А будет вытеснен до завершения своей работы? Тогда поток Б прочитает нестабильные данные.

Чтобы проиллюстрировать проблему, связанную с параллелизмом, давайте создадим еще один проект консольного приложения под названием `MultiThreadedPrinting`. В приложении снова будет использоваться построенный ранее класс `Printer`, но на этот раз метод `PrintNumbers()` приостановит текущий поток на случайно сгенерированный период времени.

```
public class Printer
{
  public void PrintNumbers()
  {
    ...
    for (int i = 0; i < 10; i++)
    {
      // Приостановить поток на случайный период времени.
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(5));
      Console.Write("{0}, ", i);
    }
    Console.WriteLine();
  }
}
```

Метод `Main()` отвечает за создание массива из десяти (уникально именованных) объектов `Thread`, каждый из которых производит вызов *одного и того же экземпляра* класса `Printer`:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("*****Synchronizing Threads *****\n");

    Printer p = new Printer();

    // Создать 10 потоков, которые указывают на один
    // и тот же метод того же самого объекта.
    Thread[] threads = new Thread[10];
    for (int i = 0; i < 10; i++)
    {
```

```
      threads[i] =
        new Thread(new ThreadStart(p.PrintNumbers));
      threads[i].Name = string.Format("Worker thread #{0}", i);
    }
    // Теперь запустить их все.
    foreach (Thread t in threads)
      t.Start();
    Console.ReadLine();
  }
}
```

Прежде чем взглянуть на тестовые запуски, кратко повторим проблему. Первичный поток внутри этого домена приложения начинает свое существование с порождения десяти вторичных рабочих потоков. Каждому рабочему потоку указывается на необходимость вызова метода `PrintNumbers()` *того же самого* экземпляра `Printer`. Поскольку никаких мер для блокировки разделяемых ресурсов этого объекта (консоли) не предпринималось, есть неплохой шанс, что текущий поток будет вытеснен до того, как метод `PrintNumbers()` выведет полные результаты. Из-за того, что в точности не известно, когда подобное может произойти (если вообще произойдет), будут получены непредсказуемые результаты. Например, вывод может выглядеть так:

```
*****Synchronizing Threads *****

-> Worker thread #1 is executing PrintNumbers()
Your numbers: -> Worker thread #0 is executing PrintNumbers()
-> Worker thread #2 is executing PrintNumbers()
Your numbers: -> Worker thread #3 is executing PrintNumbers()
Your numbers: -> Worker thread #4 is executing PrintNumbers()
Your numbers: -> Worker thread #6 is executing PrintNumbers()
Your numbers: -> Worker thread #7 is executing PrintNumbers()
Your numbers: -> Worker thread #8 is executing PrintNumbers()
Your numbers: -> Worker thread #9 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #5 is executing PrintNumbers()
Your numbers: 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 2, 1, 0, 0, 4,
3,
4, 1, 2, 4, 5, 5, 5, 6, 6, 6, 2, 7, 7, 7, 3, 4, 0, 8, 4, 5, 1, 5, 8, 8, 9,
2, 6, 1, 0, 9, 1,
6, 2, 7, 9,
2, 1, 7, 8, 3, 2, 3, 3, 9,
8, 4, 4, 5, 9,
4, 3, 5, 5, 6, 3, 6, 7, 4, 7, 6, 8, 7, 4, 8, 5, 5, 6, 6, 8, 7, 7, 9,
8, 9,
8, 9,
9,
9,
```

Запустим приложение еще несколько раз. Вот еще один вариант вывода:

```
*****Synchronizing Threads *****
-> Worker thread #0 is executing PrintNumbers()
-> Worker thread #1 is executing PrintNumbers()
-> Worker thread #2 is executing PrintNumbers()
Your numbers: -> Worker thread #4 is executing PrintNumbers()
Your numbers: -> Worker thread #5 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #6 is executing PrintNumbers()
Your numbers: -> Worker thread #7 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #8 is executing PrintNumbers()
```

```
Your numbers: -> Worker thread #9 is executing PrintNumbers()
Your numbers: -> Worker thread #3 is executing PrintNumbers()
Your numbers: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
2,
2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4,
4,
 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7,
7
, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9,
9,
9,
9,
9,
9,
9,
9,
9,
9,
```

На заметку! Если получить непредсказуемый вывод не удается, увеличьте количество потоков с 10 до 100 (к примеру) или добавьте в код еще один вызов `Thread.Sleep()`. В конце концов, вы столкнетесь с проблемой параллелизма.

Должно быть совершенно ясно, что здесь присутствуют проблемы. В то время как каждый поток сообщает экземпляру `Printer` о необходимости вывода числовых данных, планировщик потоков благополучно переключает потоки в фоновом режиме. В итоге получается несогласованный вывод. Нужен способ программной реализации синхронизированного доступа к разделяемым ресурсам. Как и можно было предположить, пространство имен `System.Threading` предлагает несколько типов, связанных с синхронизацией. В языке C# также предусмотрено специальное ключевое слово для синхронизации разделяемых данных в многопоточных приложениях.

Синхронизация с использованием ключевого слова `lock` языка C#

Первый прием, который можно применять для синхронизации доступа к разделяемым ресурсам, предполагает использование ключевого слова `lock` языка C#. Оно позволяет определять блок операторов, которые должны быть синхронизованными между потоками. В результате входящие потоки не могут прерывать текущий поток, мешая ему завершить свою работу. Ключевое слово `lock` требует указания *маркера* (объектной ссылки), который должен быть получен потоком для входа в область действия блокировки. Чтобы попытаться заблокировать *закрытый* метод уровня экземпляра, необходимо просто передать ссылку на текущий тип:

```
private void SomePrivateMethod()
{
  // Использовать текущий объект как маркер потока.
  lock(this)
  {
    // Весь код внутри этого блока является безопасным к потокам.
  }
}
```

Тем не менее, если блокируется область кода внутри *открытого* члена, безопаснее (да и рекомендуется) объявить закрытую переменную-член типа `object` для применения в качестве маркера блокировки:

```
public class Printer
{
  // Маркер блокировки.
  private object threadLock = new object();

  public void PrintNumbers()
  {
    // Использовать маркер блокировки.
    lock (threadLock)
    {
      ...
    }
  }
}
```

Если взглянуть на метод `PrintNumbers()`, то можно заметить, что разделяемым ресурсом, за доступ к которому соперничают потоки, является окно консоли. Следовательно, если поместить весь код взаимодействия с типом `Console` внутрь области `lock`, как показано ниже, то будет построен метод, который позволит текущему потоку завершить свою задачу:

```
public void PrintNumbers()
{
  // Использовать в качестве маркера блокировки закрытый член object.
  lock (threadLock)
  {
    // Вывести информацию о потоке.
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);

    // Вывести числа.
    Console.Write("Your numbers: ");
    for (int i = 0; i < 10; i++)
    {
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(5));
      Console.Write("{0}, ", i);
    }
    Console.WriteLine();
  }
}
```

Как только поток входит в область `lock`, маркер блокировки (в данном случае ссылка на текущий объект) становится недоступным другим потокам до тех пор, пока блокировка не будет освобождена после выхода из области `lock`. Таким образом, если поток А получил маркер блокировки, то другие потоки не смогут войти *ни в одну из областей*, которые используют тот же самый маркер, до тех пор, пока поток А не освободит его.

На заметку! Если необходимо блокировать код в статическом методе, то следует просто объявить закрытую статическую переменную-член типа `object`, которая и будет служить маркером блокировки.

Запустив приложение, можно заметить, что каждый поток получил возможность выполнить свою работу до конца:

```
*****Synchronizing Threads *****
-> Worker thread #0 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

```
-> Worker thread #1 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #3 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #2 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #4 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #5 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #7 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #6 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #8 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #9 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Исходный код. Проект `MultiThreadedPrinting` доступен в подкаталоге `Chapter_19`.

Синхронизация с использованием типа `System.Threading.Monitor`

Оператор `lock` языка C# на самом деле представляет собой сокращение для работы с классом `System.Threading.Monitor`. При обработке компилятором C# область `lock` преобразуется в следующую конструкцию (в чем легко убедиться с помощью утилиты `ldasm.exe`):

```
public void PrintNumbers()
{
  Monitor.Enter(threadLock);
  try
  {
    // Вывести информацию о потоке.
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);
    // Вывести числа.
    Console.Write("Your numbers: ");
    for (int i = 0; i < 10; i++)
    {
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(5));
      Console.Write("{0}, ", i);
    }
    Console.WriteLine();
  }
  finally
  {
    Monitor.Exit(threadLock);
  }
}
```

Первым делом обратите внимание, что конечным получателем маркера потока, который указывается как аргумент ключевого слова `lock`, является метод `Monitor`.

Enter(). Весь код внутри области lock помещен внутрь блока try. Соответствующая конструкция finally гарантирует освобождение маркера блокировки (посредством метода Monitor.Exit()), даже если возникнут любые исключения времени выполнения. Модифицировав программу MultiThreadShareData с целью прямого применения типа Monitor (как только что было показано), вы обнаружите, что вывод идентичен.

С учетом того, что ключевое слово lock требует написания меньшего объема кода, чем при явной работе с типом System.Threading.Monitor, может возникнуть вопрос о преимуществах использования этого типа напрямую. Выражаясь кратко, тип Monitor обеспечивает большую степень контроля. Применяя тип Monitor, можно заставить активный поток ожидать в течение некоторого периода времени (с помощью статического метода Monitor.Wait()), информировать ожидающие потоки о том, что текущий поток завершен (через статические методы Monitor.Pulse() и Monitor.PulseAll()), и т.д.

Как и можно было ожидать, в значительном числе случаев ключевого слова lock будет достаточно. Если вас интересуют дополнительные члены класса Monitor, обращайтесь в документацию .NET Framework 4.6 SDK.

Синхронизация с использованием типа System.Threading.Interlocked

Не заглядывая в код CIL, трудно поверить, что присваивание и простые арифметические операции *не являются атомарными*. По этой причине в пространстве имен System.Threading предоставляется тип, который позволяет атомарно оперировать одиночным элементом данных с меньшими накладными расходами, чем тип Monitor. В классе Interlocked определены статические члены, часть которых описана в табл. 19.4.

Таблица 19.4. Избранные статические члены типа System.Threading.Interlocked

Член	Назначение
CompareExchange()	Безопасно проверяет два значения на равенство и, если они равны, то заменяет одно из значений третьим
Decrement()	Безопасно уменьшает значение на 1
Exchange()	Безопасно меняет два значения местами
Increment()	Безопасно увеличивает значение на 1

Несмотря на то что это не сразу видно, процесс атомарного изменения одиночного значения довольно часто применяется в многопоточной среде. Предположим, что есть метод AddOne(), который инкрементирует целочисленную переменную-член по имени intVal. Вместо написания кода синхронизации вроде показанного ниже:

```
public void AddOne()
{
  lock(myLockToken)
  {
    intVal++;
  }
}
```

код можно упростить, используя статический метод Interlocked.Increment(). Методу потребуется передать инкрементируемую переменную по ссылке. Обратите внимание, что метод Increment() не только изменяет значение входного параметра, но также возвращает полученное новое значение:

```
public void AddOne()
{
  int newVal = Interlocked.Increment(ref intVal);
}
```

В дополнение к методам `Increment()` и `Decrement()` тип `Interlocked` позволяет атомарно присваивать числовые и объектные данные. Например, чтобы присвоить переменной-члену значение `83`, можно обойтись без явного оператора `lock` (или явной логики `Monitor`) и применить метод `Interlock.Exchange()`:

```
public void SafeAssignment()
{
  Interlocked.Exchange(ref myInt, 83);
}
```

Наконец, если необходимо проверить два значения на равенство и изменить элемент сравнения в безопасной к потокам манере, можно использовать метод `Interlocked.CompareExchange()`:

```
public void CompareAndExchange()
{
  // Если значение i равно 83, то изменить его на 99.
  Interlocked.CompareExchange(ref i, 99, 83);
}
```

Синхронизация с использованием атрибута [Synchronization]

Последний из примитивов синхронизации, который мы здесь рассмотрим — атрибут `[Synchronization]`, который определен в пространстве имен `System.Runtime.Remoting.Contexts`. По существу этот атрибут уровня класса блокирует весь код членов экземпляра класса для обеспечения безопасности к потокам. Когда среда CLR размещает в памяти объекты, снабженные атрибутами `[Synchronization]`, она помещает объект внутрь контекста синхронизации. Как было показано в главе 17, объекты, которые не должны выходить за границы контекста, являются производными от `ContextBoundObject`. Следовательно, чтобы сделать класс `Printer` безопасным к потокам (без явного написания соответствующего кода внутри членов класса), его определение можно обновить, как показано ниже:

```
using System.Runtime.Remoting.Contexts;
...
// Все методы класса Printer теперь безопасны к потокам!
[Synchronization]
public class Printer : ContextBoundObject
{
  public void PrintNumbers()
  {
    ...
  }
}
```

В некоторых отношениях этот подход выглядит как "ленивый" способ написания безопасного к потокам кода, т.к. не приходится углубляться в детали того, какие именно аспекты типа действительно манипулируют чувствительными к потокам данными. Однако главный недостаток подхода в том, что даже если определенный метод не работает с чувствительными к потокам данными, среда CLR будет *по-прежнему* блокировать вызовы этого метода. Очевидно, что это может привести к снижению общей функциональности типа, потому применяйте описанный прием с осторожностью.

Программирование с использованием обратных вызовов Timer

Многие приложения нуждаются в вызове специфического метода через регулярные интервалы времени. Например, в приложении может существовать необходимость в отображении текущего времени в панели состояния с помощью определенной вспомогательной функции. Или, скажем, нужно, чтобы приложение эпизодически вызывало вспомогательную функцию, выполняющую некритичные фоновые задачи, такие как проверка поступления новых сообщений электронной почты. В ситуациях подобного рода можно применять тип `System.Threading.Timer` в сочетании со связанным делегатом по имени `TimerCallback`.

В целях иллюстрации предположим, что имеется проект консольного приложения (`TimerApp`), которое будет выводить текущее время каждую секунду до тех пор, пока пользователь не нажмет клавишу <Enter> для прекращения работы приложения. Первый очевидный шаг — написание метода, который будет вызываться типом `Timer` (не забудьте импортировать в файл кода пространство имен `System.Threading`):

```
class Program
{
  static void PrintTime(object state)
  {
    Console.WriteLine("Time is: {0}",
      DateTime.Now.ToLongTimeString());
  }
  static void Main(string[] args)
  {
  }
}
```

Обратите внимание, что метод `PrintTime()` принимает единственный параметр типа `System.Object` и возвращает `void`. Это обязательно, потому что делегат `TimerCallback` может вызывать только методы, которые соответствуют такой сигнатуре. Значение, передаваемое целевому методу делегата `TimerCallback`, может быть объектом любого типа (в случае примера с электронной почтой параметр может представлять имя сервера Microsoft Exchange для взаимодействия в течение процесса). Также обратите внимание, что поскольку этот параметр на самом деле является экземпляром типа `System.Object`, в нем можно передавать несколько аргументов, используя `System.Array` или специальный класс либо структуру.

Следующий шаг связан с конфигурированием экземпляра делегата `TimerCallback` и передачей его объекту `Timer`. В дополнение к настройке делегата `TimerCallback` конструктор `Timer` позволяет указывать необязательный информационный параметр для передачи целевому методу делегата (определенный как `System.Object`), интервал вызова метода и период ожидания (в миллисекундах), который должен истечь перед первым вызовом. Вот пример:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Working with Timer type *****\n");

  // Создать делегат для типа Timer.
  TimerCallback timeCB = new TimerCallback(PrintTime);

  // Установить параметры таймера.
  Timer t = new Timer(
```

```
    timeCB,          // Объект делегата TimerCallback.
    null,            // Информация для передачи в вызванный метод
                     // (null, если информация отсутствует).
    0,               // Период ожидания перед запуском (в миллисекундах).
    1000);           // Интервал между вызовами (в миллисекундах).
  Console.WriteLine("Hit key to terminate...");
  Console.ReadLine();
}
```

В этом случае метод `PrintTime()` вызывается приблизительно каждую секунду и не получает никакой дополнительной информации. Ниже показан вывод примера:

```
***** Working with Timer type *****
Hit key to terminate...
Time is: 6:51:48 PM
Time is: 6:51:49 PM
Time is: 6:51:50 PM
Time is: 6:51:51 PM
Time is: 6:51:52 PM
Press any key to continue . . .
```

Чтобы передать целевому методу делегата какую-то информацию, необходимо просто заменить значение `null` во втором параметре конструктора подходящей информацией, например:

```
// Установить параметры таймера.
Timer t = new Timer(timeCB, "Hello From Main", 0, 1000);
```

Получить входные данные можно следующим образом:

```
static void PrintTime(object state)
{
  Console.WriteLine("Time is: {0}, Param is: {1}",
    DateTime.Now.ToLongTimeString(), state.ToString());
}
```

Исходный код. Проект `TimerApp` доступен в подкаталоге `Chapter_19`.

Пул потоков CLR

Следующей темой, связанной с потоками, которую мы рассмотрим в главе, будет роль пула потоков CLR. При асинхронном вызове метода с применением типов делегатов (посредством метода `BeginInvoke()`) среда CLR не создает новый поток в прямом смысле слова. В целях эффективности метод `BeginInvoke()` делегата задействует пул рабочих потоков, который поддерживается исполняющей средой. Для взаимодействия с этим пулом ожидающих потоков в пространстве имен `System.Threading` предлагается класс `ThreadPool`.

Чтобы запросить поток из пула для обработки вызова метода, можно использовать метод `ThreadPool.QueueUserWorkItem()`. Этот метод имеет перегруженную версию, которая позволяет в дополнение к экземпляру делегата `WaitCallback` указывать необязательный параметр `System.Object` для передачи специальных данных состояния:

```
public static class ThreadPool
{
  ...
  public static bool QueueUserWorkItem(WaitCallback callBack);
  public static bool QueueUserWorkItem(WaitCallback callBack, object state);
}
```

Делегат `WaitCallback` может указывать на любой метод, который принимает в качестве единственного параметра экземпляр `System.Object` (представляющий необязательные данные состояния) и ничего не возвращает. Обратите внимание, что если при вызове `QueueUserWorkItem()` не задается экземпляр `System.Object`, то среда CLR автоматически передает значение `null`. Чтобы продемонстрировать работу методов очередей, работающих с пулом потоков CLR, рассмотрим еще раз программу, в которой применяется тип `Printer`. На этот раз массив объектов `Thread` не создается вручную, а метод `PrintNumbers()` будет назначаться членам пула потоков:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with the CLR Thread Pool *****\n");
    Console.WriteLine("Main thread started. ThreadID = {0}",
      Thread.CurrentThread.ManagedThreadId);
    Printer p = new Printer();
    WaitCallback workItem = new WaitCallback(PrintTheNumbers);
    // Поставить в очередь метод десять раз.
    for (int i = 0; i < 10; i++)
    {
      ThreadPool.QueueUserWorkItem(workItem, p);
    }
    Console.WriteLine("All tasks queued");
    Console.ReadLine();
  }
  static void PrintTheNumbers(object state)
  {
    Printer task = (Printer)state;
    task.PrintNumbers();
  }
}
```

У вас может возникнуть вопрос: почему взаимодействовать с пулом потоков, поддерживаемым средой CLR, выгоднее по сравнению с явным созданием объектов `Thread`? Использование пула потоков обеспечивает следующие преимущества.

- Пул потоков эффективно управляет потоками, сводя к минимуму количество потоков, которые должны создаваться, запускаться и останавливаться.
- За счет применения пула потоков можно сосредоточиться на решении задачи, а не на потоковой инфраструктуре приложения.

Тем не менее, в некоторых случаях ручное управление потоками оказывается более предпочтительным. Ниже приведены примеры.

- Когда требуются потоки переднего плана или должен устанавливаться приоритет потока. Потоки из пула *всегда* являются фоновыми и обладают стандартным приоритетом (`ThreadPriority.Normal`).
- Когда требуется поток с фиксированной идентичностью, чтобы его можно было прерывать, приостанавливать или находить по имени.

Исходный код. Проект `ThreadPoolApp` доступен в подкаталоге `Chapter_19`.

На этом исследование пространства имен `System.Threading` завершено. Несомненно, понимание вопросов, рассмотренных в настоящей главе до сих пор (особенно в разделе, посвященном проблемам параллелизма), будет чрезвычайно ценным при создании многопоточного приложения. А теперь, опираясь на этот фундамент, мы переключим внимание на несколько новых аспектов, связанных с потоками, которые присутствуют только в .NET 4.0 и последующих версиях. Для начала мы обратимся к альтернативной потоковой модели, которая называется TPL.

Параллельное программирование с использованием TPL

К этому моменту вы ознакомились с двумя технологиями программирования (асинхронные делегаты и члены пространства имен `System.Threading`), которые позволяют строить многопоточное программное обеспечение. Вспомните, что оба подхода будут работать в любой версии платформы .NET.

Начиная с .NET 4.0 в Microsoft ввели новый подход к разработке многопоточных приложений, предусматривающий применение библиотеки параллельного программирования, которая называется *TPL* (Task Parallel Library — библиотека параллельных задач). С помощью типов из `System.Threading.Tasks` можно строить мелкомодульный масштабируемый параллельный код без необходимости напрямую иметь дело с потоками или пулом потоков. Однако это не говорит о том, что вы не будете использовать типы из пространства имен `System.Threading` во время применения TPL. В реальности эти два инструментальных набора для создания многопоточных приложений могут вполне естественным образом работать вместе. Это особенно верно потому, что пространство имен `System.Threading` по-прежнему предоставляет большинство примитивов синхронизации, которые рассматривались ранее (`Monitor`, `Interlocked` и т.д.). В итоге, скорее всего, вы обнаружите, что иметь дело с TPL предпочтительнее, чем с первоначальным пространством имен `System.Threading`, т.к. те же самые задачи могут быть решены гораздо проще.

На заметку! Кстати, имейте в виду, что новые ключевые слова `async` и `await` языка C# используют разнообразные члены пространства имен `System.Threading.Tasks`.

Пространство имен `System.Threading.Tasks`

Коллективно типы из пространства `System.Threading.Tasks` называются *библиотекой параллельных задач* (Task Parallel Library — TPL). Библиотека TPL будет автоматически распределять нагрузку приложения между доступными процессорами в динамическом режиме с применением пула потоков CLR. Библиотека TPL поддерживает разбиение работы на части, планирование потоков, управление состоянием и другие низкоуровневые детали. В конечном итоге появляется возможность максимизировать производительность приложений .NET, не касаясь сложностей прямой работы с потоками (рис. 19.2).

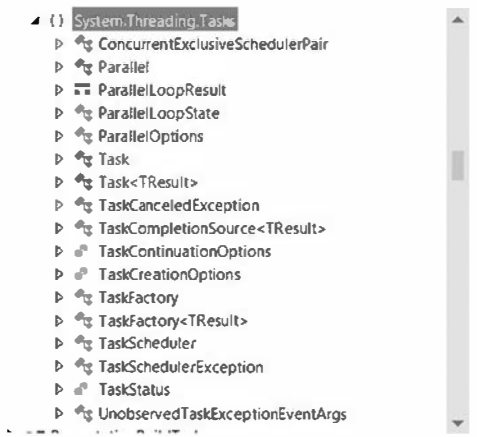


Рис. 19.2. Члены пространства имен `System.Threading.Tasks`

Роль класса Parallel

Основным классом в TPL является System.Threading.Tasks.Parallel. Он поддерживает набор методов, которые позволяют осуществлять итерацию по коллекции данных (точнее по объекту, реализующему интерфейс IEnumerable<T>) в параллельной манере. В документации .NET Framework 4.6 SDK указано, что в этом классе определены два главных статических метода, Parallel.For() и Parallel.ForEach(), каждый из которых имеет много перегруженных версий.

Упомянутые методы позволяют создавать тело из операторов кода, которое будет выполняться в параллельном режиме. Концептуально эти операторы представляют логику того же рода, которая была бы написана в нормальной циклической конструкции (с использованием ключевых слов for и foreach языка C#). Преимущество заключается в том, что класс Parallel будет самостоятельно извлекать потоки из пула потоков (и управлять параллелизмом). Оба метода требуют передачи совместимого с IEnumerable или IEnumerable<T> контейнера, который хранит данные, подлежащие обработке в параллельном режиме. Контейнер может быть простым массивом, необобщенной коллекцией (вроде ArrayList), обобщенной коллекцией (наподобие List<T>) или результатами, полученными из запроса LINQ.

Вдобавок понадобится применять делегаты System.Func<T> и System.Action<T> для указания целевого метода, который будет вызываться при обработке данных. Делегат Func<T> уже встречался в главе 12 во время исследования технологии LINQ to Objects. Вспомните, что Func<T> представляет метод, который возвращает значение и принимает различное количество аргументов. Делегат Action<T> похож на Func<T> в том, что позволяет задавать метод, принимающий несколько параметров, но этот метод должен возвращать void.

Хотя можно было бы вызывать методы Parallel.For() и Parallel.ForEach() и передавать им строго типизированный объект делегата Func<T> или Action<T>, задача программирования упрощается за счет использования подходящих анонимных методов или лямбда-выражений C#.

Обеспечение параллелизма данных с помощью класса Parallel

Первое применение библиотеки TPL связано с обеспечением *параллелизма данных*. Этим термином обозначается задача прохода по массиву или коллекции в параллельной манере с помощью метода Parallel.For() или Parallel.ForEach(). Предположим, что необходимо выполнить некоторые трудоемкие операции файлового ввода-вывода. В частности, требуется загрузить в память большое число файлов *.jpg, повернуть содержащиеся в них изображения и сохранить модифицированные данные изображений в новом месте.

В документации .NET Framework 4.6 SDK приведен пример консольного приложения для такой часто встречающейся задачи. Тем не менее, мы решим ее с использованием графического пользовательского интерфейса, чтобы взглянуть на применение "анонимных делегатов", позволяющих вторичным потокам обновлять первичный поток пользовательского интерфейса.

На заметку! При построении многопоточного приложения с графическим пользовательским интерфейсом вторичные потоки никогда не могут напрямую обращаться к элементам управления пользовательского интерфейса. Причина в том, что элементы управления (кнопки, текстовые поля, метки, индикаторы хода работ и т.п.) привязаны к потоку, в котором они создавались. В следующем примере иллюстрируется один из способов обеспечения для вторичных потоков возможности получать доступ к элементам пользовательского интерфейса в безопасной к потокам манере. Во время рассмотрения ключевых слов async и await языка C# будет предложен более простой подход.

В целях иллюстрации создадим проект приложения Windows Forms по имени DataParallelismWithForEach и в окне Solution Explorer переименуем файл Form1.cs на MainForm.cs. После этого импортируем в главный файл кода следующие пространства имен:

```
// Удостоверьтесь в импортировании этих пространств имен!
using System.Threading.Tasks;
using System.Threading;
using System.IO;
```

Графический пользовательский интерфейс приложения содержит многострочное текстовое поле (TextBox) и одну кнопку (Button) с именем btnProcessImages. Текстовое поле предназначено для ввода данных во время выполнения работы в фоновом режиме, демонстрируя тем самым неблокирующую природу параллельной задачи. В обработчике события Click элемента Button в конечном итоге будет использоваться библиотека TPL, а пока напишем блокирующий код.

На заметку! Вы должны обновить строку, передаваемую методу Directory.GetFiles(), чтобы она указывала на вашей машине конкретный путь к каталогу, который содержит какие-то файлы изображений. Для удобства в каталог Solution проекта включено несколько примеров изображений (поставляемых в составе операционной системы Windows).

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }
  private void btnProcessImages_Click(object sender, EventArgs e)
  {
    ProcessFiles();
  }
  private void ProcessFiles()
  {
    // Загрузить все файлы *.jpg и создать новый каталог
    // для модифицированных данных.
    string[] files = Directory.GetFiles
      (@"C:\TestPictures", "*.jpg",
      SearchOption.AllDirectories);
    string newDir = @"C:\ModifiedPictures";
    Directory.CreateDirectory(newDir);

    // Обработать данные изображений в блокирующей манере.
    foreach (string currentFile in files)
    {
      string filename = Path.GetFileName(currentFile);
      using (Bitmap bitmap = new Bitmap(currentFile))
      {
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(Path.Combine(newDir, filename));

        // Вывести идентификатор потока, обрабатывающего текущее изображение.
        this.Text = string.Format("Processing {0} on thread {1}", filename,
          Thread.CurrentThread.ManagedThreadId);
      }
    }
  }
}
```

Обратите внимание, что метод `ProcessFiles()` выполнит поворот изображения в каждом файле `*.jpg` из указанного каталога, который в текущий момент содержит 37 файлов (при необходимости укажите другой путь в вызове `Directory.GetFiles()`). В настоящее время вся работа происходит в первичном потоке исполняемой программы. Следовательно, после щелчка на кнопке программа выглядит зависшей. Более того, заголовок окна также сообщит о том, что файл обрабатывается тем же самым первичным потоком, т.к. в наличии есть только один поток выполнения.

Чтобы обрабатывать файлы на как можно большем количестве процессоров, текущий цикл `foreach` можно заменить вызовом метода `Parallel.ForEach()`. Вспомните, что этот метод имеет множество перегруженных версий. Простейшая форма метода принимает совместимый с `IEnumerable<T>` объект, который содержит элементы, подлежащие обработке (например, строковый массив `files`), и делегат `Action<T>`, указывающий на метод, который будет выполнять необходимую работу.

Ниже показан модифицированный код, в котором вместо литерального объекта делегата `Action<T>` применяется лямбда-операция C#. Обратите внимание, что в коде *закомментированы* строки, которые отображают идентификатор потока, обрабатывающего текущий файл изображения. Причина объясняется в следующем разделе.

```
// Обработать данные изображений в параллельном режиме!
Parallel.ForEach(files, currentFile =>
  {
    string filename = Path.GetFileName(currentFile);

    using (Bitmap bitmap = new Bitmap(currentFile))
    {
      bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
      bitmap.Save(Path.Combine(newDir, filename));

      // Этот оператор кода теперь вызывает проблемы! См. следующий раздел.
      // this.Text = string.Format("Processing {0} on thread {1}", filename,
      //   Thread.CurrentThread.ManagedThreadId);
    }
  }
);
```

Доступ к элементам пользовательского интерфейса во вторичных потоках

Вы заметили, что в показанном выше коде закомментированы строки, которые обновляют заголовок главного окна значением идентификатора текущего выполняющегося потока. Как отмечалось ранее, элементы управления графического пользовательского интерфейса привязаны к потоку, в котором они были созданы. Если вторичные потоки пытаются получить доступ к элементу управления, который они напрямую не создавали, то при отладке программного обеспечения возникают ошибки времени выполнения. С другой стороны, если *запустить* приложение (нажатием <Ctrl+F5>), то первоначальный код может и не вызвать каких-либо проблем.

На заметку! Не лишним будет повторить: при отладке (по нажатию <F5>) многопоточного приложения IDE-среда Visual Studio часто способна перехватывать ошибки, когда вторичный поток обращается к элементу управления, созданному в первичном потоке. Однако нередко после запуска (с помощью <Ctrl+F5>) приложение может выглядеть функционирующим корректно (или же ошибка может возникнуть довольно скоро). Если не предпринять меры предосторожности (описанные далее), то приложение в подобных обстоятельствах может потенциально сгенерировать ошибку во время выполнения.

Один из подходов, который можно использовать для предоставления вторичным потокам доступа к элементам управления в безопасной к потокам манере, предусматривает применение другого приема — *анонимного делегата*. Родительский класс `Control` в API-интерфейсе Windows Forms определяет метод по имени `Invoke()`, который принимает на входе `System.Delegate`. Этот метод можно вызывать внутри кода, выполняющегося во вторичных потоках, чтобы обеспечить возможность безопасного к потокам обновления пользовательского интерфейса для заданного элемента управления. В то время как весь требуемый код делегата можно было бы написать напрямую, большинство разработчиков используют в качестве простой альтернативы анонимные делегаты. Вот как выглядит модифицированный код:

```
using (Bitmap bitmap = new Bitmap(currentFile))
{
  bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
  bitmap.Save(Path.Combine(newDir, filename));

  // Это больше не работает!
  // this.Text = string.Format("Processing {0} on thread {1}", filename,
  //   Thread.CurrentThread.ManagedThreadId);

  // Вызвать Invoke на объекте Form, чтобы позволить вторичным потокам
  // получать доступ к элементам управления в безопасной к потокам манере.
  this.Invoke((Action)delegate
    {
      this.Text = string.Format("Processing {0} on thread {1}", filename,
                  Thread.CurrentThread.ManagedThreadId);
    }
  );
}
```

На заметку! Метод `this.Invoke()` уникален для API-интерфейса Windows Forms. При построении приложения WPF для той же цели будет применяться вызов `this.Dispatcher.Invoke()`.

Теперь после запуска программы библиотека TPL распределит рабочую нагрузку по множеству потоков из пула, используя столько процессоров, сколько возможно. Тем не менее, имена уникальных потоков в заголовке окна отображаться не будут, а при вводе в текстовой области ничего не будет видно до тех пор, пока не обработаются все файлы изображений! Причина в том, что первичный поток пользовательского интерфейса попрежнему блокируется, ожидая завершения работы всеми остальными потоками.

Класс `Task`

Класс `Task` позволяет легко вызывать метод во вторичном потоке и может применяться как простая альтернатива работе с асинхронными делегатами. Изменим обработчик события `Click` элемента управления `Button` следующим образом:

```
private void btnProcessImages_Click(object sender, EventArgs e)
{
  // Запустить новую "задачу" для обработки файлов.
  Task.Factory.StartNew(() =>
  {
    ProcessFiles();
  });
}
```

Свойство `Factory` класса `Task` возвращает объект `TaskFactory`. Методу `StartNew()` при вызове передается делегат `Action<T>` (здесь это скрыто с помощью подходящего лямбда-выражения), который указывает на метод, подлежащий вызову в асинхронной манере. После этой небольшой модификации вы обнаружите, что заголовок окна отображает информацию о потоке из пула, обрабатывающем конкретный файл, а текстовое поле может принимать ввод, т.к. пользовательский интерфейс больше не блокируется.

Обработка запроса на отмену

В текущий пример можно внести еще одно улучшение — предоставить пользователю способ для останова обработки данных изображения путем щелчка на второй кнопке Cancel (Отмена). К счастью, методы `Parallel.For()` и `Parallel.ForEach()` поддерживают отмену за счет использования *признаков отмены*. При вызове методов на объекте `Parallel` им можно передавать объект `ParallelOptions`, который, в свою очередь, содержит объект `CancellationTokenSource`.

Первым делом определим в производном от `Form` классе закрытую переменную-член `cancelToken` типа `CancellationTokenSource`:

```
public partial class MainForm : Form
{
  // Новая переменная уровня Form.
  private CancellationTokenSource cancelToken =
    new CancellationTokenSource();
  ...
}
```

Предполагая, что в визуальном конструкторе был добавлен новый элемент управления `Button` (по имени `btnCancel`), реализуем его обработчик события `Click`:

```
private void btnCancel_Click(object sender, EventArgs e)
{
  // Используется для сообщения всем рабочим потокам о необходимости останова!
  cancelToken.Cancel();
}
```

Теперь можно внести необходимые модификации в метод `ProcessFiles()`. Вот окончательная реализация этого метода:

```
private void ProcessFiles()
{
  // Использовать экземпляр ParallelOptions для хранения CancellationToken.
  ParallelOptions parOpts = new ParallelOptions();
  parOpts.CancellationToken = cancelToken.Token;
  parOpts.MaxDegreeOfParallelism = System.Environment.ProcessorCount;

  //Загрузить все файлы *.jpg и создать новый каталог для модифицированных данных.
  string[] files = Directory.GetFiles
    (@"C:\Users\Public\Pictures\Sample Pictures", "*.jpg",
       SearchOption.AllDirectories);
  string newDir = @"C:\ModifiedPictures";
  Directory.CreateDirectory(newDir);

  try
  {
    // Обработать данные изображения в параллельном режиме!
    Parallel.ForEach(files, parOpts, currentFile =>
    {
      parOpts.CancellationToken.ThrowIfCancellationRequested();
      string filename = Path.GetFileName(currentFile);
```

```
      using (Bitmap bitmap = new Bitmap(currentFile))
      {
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(Path.Combine(newDir, filename));
        this.Invoke((Action)delegate
          {
            this.Text = string.Format("Processing {0} on thread {1}", filename,
              Thread.CurrentThread.ManagedThreadId);
          }
        );
      }
    }
    );
    this.Invoke((Action)delegate
    {
      this.Text = "Done!";
    });
  }
  catch (OperationCanceledException ex)
  {
    this.Invoke((Action)delegate
    {
      this.Text = ex.Message;
    });
  }
}
```

Обратите внимание, что в начале метода конфигурируется объект `ParallelOptions` с установкой его свойства `CancellationToken` для применения признака `CancellationTokenSource`. Кроме того, этот объект `ParallelOptions` передается во втором параметре методу `Parallel.ForEach()`.

Внутри логики цикла осуществляется вызов `ThrowIfCancellationRequested()` на признаке отмены, гарантируя тем самым, что если пользователь щелкнет на кнопке Cancel, то все потоки будут остановлены и в качестве уведомления сгенерируется исключение времени выполнения. Перехватив исключение `OperationCanceledException`, можно добавить в текст главного окна сообщение об ошибке.

Исходный код. Проект `DataParallelismWithForEach` доступен в подкаталоге `Chapter_19`.

Обеспечение параллелизма задач с помощью класса `Parallel`

В дополнение к обеспечению параллелизма данных библиотека TPL также может использоваться для запуска любого количества асинхронных задач с помощью метода `Parallel.Invoke()`. Такой подход немного проще, чем применение делегатов или типов из пространства имен `System.Threading`, но если нужна более высокая степень контроля над выполняемыми задачами, то следует отказаться от использования `Parallel.Invoke()` и напрямую работать с классом `Task`, как это делалось в предыдущем примере.

Чтобы продемонстрировать параллелизм задач в действии, создадим новый проект приложения Windows Forms по имени `MyEBookReader` и импортируем пространства имен `System.Threading.Tasks` и `System.Net`. Данный пример является модификацией примера из документации .NET Framework 4.6 SDK. Здесь мы будем извлекать публично доступную электронную книгу из сайта проекта Гутенберга (`www.gutenberg.org`) и затем параллельно выполнять набор длительных задач.

Графический пользовательский интерфейс состоит из многострочного текстового поля типа TextBox (по имени txtBook) и двух кнопок типа Button (btnDownload и btnGetStats). Для каждой кнопки понадобится обработать событие Click, а в файле кода формы объявить на уровне класса переменную string по имени theEBook. Ниже приведена реализация обработчика события Click для кнопки btnDownload:

```
private void btnDownload_Click(object sender, EventArgs e)
{
  WebClient wc = new WebClient();
  wc.DownloadStringCompleted += (s, eArgs) =>
  {
    theEBook = eArgs.Result;
    txtBook.Text = theEBook;
  };
  // Загрузить электронную книгу "A Tale of Two Cities" Чарльза Диккенса.
  wc.DownloadStringAsync(new Uri("http://www.gutenberg.org/files/98/98-8.
txt"));
}
```

Класс WebClient определен в пространстве имен System.Net. Он предоставляет несколько методов для отправки и получения данных от ресурса, идентифицируемого посредством URL. В свою очередь, многие из них имеют асинхронные версии, такие как DownloadStringAsync(). Этот метод автоматически порождает новый поток из пула потоков CLR. Когда объект WebClient завершает получение данных, он инициирует событие DownloadStringCompleted, которое обрабатывается с применением лямбда-выражения C#. Если вызвать синхронную версию этого метода (DownloadString()), то форма на некоторое время перестанет реагировать на действия пользователя.

Обработчик события Click кнопки btnGetStats реализован так, чтобы извлекать индивидуальные слова, содержащиеся в переменной theEBook, и передавать строковый массив на обработку нескольким вспомогательным методам:

```
private void btnGetStats_Click(object sender, EventArgs e)
{
  // Получить слова из электронной книги.
  string[] words = theEBook.Split(new char[]
    { ' ', '\u000A', ',', '.', ';', ':', '-', '?', '/' },
    StringSplitOptions.RemoveEmptyEntries);
  // Найти 10 наиболее часто встречающихся слов.
  string[] tenMostCommon = FindTenMostCommon(words);
  // Получить самое длинное слово.
  string longestWord = FindLongestWord(words);
  // Когда все задачи завершены, построить строку,
  // показывающую всю статистику в окне сообщений.
  StringBuilder bookStats = new StringBuilder("Ten Most Common Words
are:\n");
  foreach (string s in tenMostCommon)
  {
    bookStats.AppendLine(s);
  }
  bookStats.AppendFormat("Longest word is: {0}", longestWord);
  bookStats.AppendLine();
  MessageBox.Show(bookStats.ToString(), "Book info");
}
```

Метод `FindTenMostCommon()` использует запрос LINQ для получения списка объектов `string`, которые наиболее часто встречаются в массиве `string`, а метод `FindLongestWord()` находит самое длинное слово:

```
private string[] FindTenMostCommon(string[] words)
{
  var frequencyOrder = from word in words
                       where word.Length > 6
                       group word by word into g
                       orderby g.Count() descending
                       select g.Key;
  string[] commonWords = (frequencyOrder.Take(10)).ToArray();
  return commonWords;
}

private string FindLongestWord(string[] words)
{
  return (from w in words orderby w.Length descending select w).FirstOrDefault();
}
```

После запуска этого проекта выполнение всех задач может занять внушительный промежуток времени в зависимости от количества процессоров машины и их тактовой частоты. В конце концов, должен появиться вывод, представленный на рис. 19.3.

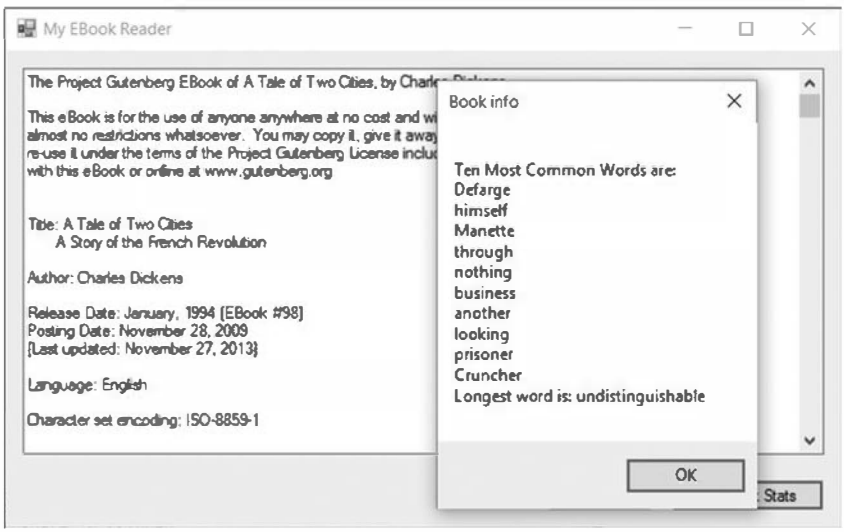


Рис. 19.3. Статистика загруженной электронной книги

Помочь удостовериться в том, что приложение задействует все доступные процессоры машины, может параллельный вызов методов `FindTenMostCommon()` и `FindLongestWord()`.

Для этого необходимо модифицировать метод `btnGetStats_Click()` следующим образом:

```
private void btnGetStats_Click(object sender, EventArgs e)
{
  // Получить слова из электронной книги.
  string[] words = theEBook.Split(
    new char[] { ' ', '\u000A', ',', '.', ';', ':', '-', '?', '/' },
    StringSplitOptions.RemoveEmptyEntries);
```

```
    string[] tenMostCommon = null;
    string longestWord = string.Empty;

    Parallel.Invoke(
      () =>
      {
        // Найти 10 наиболее часто встречающихся слов.
        tenMostCommon = FindTenMostCommon(words);
      },
      () =>
      {
        // Найти самое длинное слово.
        longestWord = FindLongestWord(words);
      });

    // Когда все задачи завершены, построить строку,
    // показывающую всю статистику в окне сообщений.
    ...
  }
```

Метод `Parallel.Invoke()` ожидает передачи в качестве параметра массива делегатов `Action<>`, который предоставляется косвенно с применением лямбда-выражения. В то время как вывод идентичен, преимущество в том, что библиотека TPL теперь будет использовать все доступные процессоры машины для вызова каждого метода параллельно, если это возможно.

Исходный код. Проект `MyEBookReader` доступен в подкаталоге `Chapter_19`.

Запросы Parallel LINQ (PLINQ)

В завершение знакомства с библиотекой TPL следует отметить, что существует еще один способ встраивания параллельных задач в приложения .NET. При желании можно применять набор расширяющих методов, которые позволяют конструировать запрос LINQ, распределяющий свою рабочую нагрузку по параллельным потокам (когда это возможно). Соответственно запросы LINQ, которые спроектированы для параллельного выполнения, называются *запросами Parallel LINQ* (*PLINQ*).

Подобно параллельному коду, написанному с использованием класса `Parallel`, в PLINQ имеется опция игнорирования запроса на обработку коллекции параллельно, если понадобится. Инфраструктура PLINQ оптимизирована во многих отношениях, включая определение того, не будет ли запрос на самом деле более эффективно выполняться в синхронной манере.

Во время выполнения PLINQ анализирует общую структуру запроса, и если есть вероятность, что запрос выиграет от распараллеливания, то он будет выполняться параллельно. Однако если распараллеливание запроса ухудшит производительность, то PLINQ просто запустит запрос последовательно. Когда возникает выбор между потенциально дорогостоящим (в смысле ресурсов) параллельным алгоритмом и экономным последовательным, предпочтение по умолчанию отдается последовательному алгоритму.

Необходимые расширяющие методы находятся в классе `ParallelEnumerable` из пространства имен `System.Linq`. В табл. 19.5 описаны некоторые полезные расширения PLINQ.

Чтобы посмотреть на PLINQ в действии, создадим проект приложения Windows Forms по имени `PLINQDataProcessingWithCancellation` и импортируем в него пространство имен `System.Threading`.

Таблица 19.5. Избранные члены класса `ParallelEnumarable`

Член	Назначение
`AsParallel()`	Указывает, что остаток запроса должен быть по возможности распараллелен
`WithCancellation()`	Указывает, что инфраструктура PLINQ должна периодически отслеживать состояние предоставленного признака отмены и при необходимости отменять выполнение
`WithDegreeOfParallelism()`	Указывает максимальное количество процессоров, которое инфраструктура PLINQ должна задействовать при распараллеливании запроса
`ForAll()`	Позволяет обрабатывать результаты параллельно без предварительного слияния с потоком потребителя, как это происходит при перечислении результата LINQ с применением ключевого слова `foreach`

Эта простая форма потребует только двух кнопок с именами `btnExecute` и `btnCancel`. Щелчок на кнопке Execute (Выполнить) приводит к запуску новой задачи, которая выполняет запрос LINQ. Данный запрос просматривает крупный массив целых чисел в поиске элементов, для которых остаток от деления на 3 равен 0. Вот *непараллельная* версия этого запроса:

```
public partial class MainForm : Form
{
  ...
  private void btnExecute_Click(object sender, EventArgs e)
  {
    // Запустить новую "задачу" для обработки целых чисел.
    Task.Factory.StartNew(() =>
    {
      ProcessIntData();
    });
  }

  private void ProcessIntData()
  {
    // Получить очень большой массив целых чисел.
    int[] source = Enumerable.Range(1, 10000000).ToArray();

    // Найти числа, для которых истинно условие num % 3 == 0,
    // и возвратить их в убывающем порядке.
    int[] modThreeIsZero = (from num in source where num % 3 == 0
          orderby num descending select num).ToArray();

    MessageBox.Show(string.Format("Found {0} numbers that match query!",
      modThreeIsZero.Count()));
  }
}
```

Создание запроса PLINQ

Чтобы проинформировать библиотеку TPL о выполнении запроса в параллельном режиме (если это возможно), необходимо использовать расширяющий метод `AsParallel()`:

```
int[] modThreeIsZero = (from num in source.AsParallel() where num % 3 == 0
                        orderby num descending select num).ToArray();
```

Обратите внимание, что общий формат запроса LINQ идентичен тому, что вы видели в предыдущих главах. Тем не менее, за счет включения вызова `AsParallel()` библиотека TPL попытается распределить рабочую нагрузку по доступным процессорам.

Отмена запроса PLINQ

С помощью объекта `CancellationTokenSource` запрос PLINQ можно информировать о прекращении обработки при определенных условиях (обычно из-за вмешательства пользователя). Объявим на уровне формы объект `CancellationTokenSource` по имени `cancelToken` и реализуем обработчик события `Click` кнопки `btnCancel` для вызова метода `Cancel()` на этом объекте. Ниже показаны соответствующие изменения в коде:

```
public partial class MainForm : Form
{
  private CancellationTokenSource cancelToken = new
CancellationTokenSource();

  private void btnCancel_Click(object sender, EventArgs e)
  {
    cancelToken.Cancel();
  }
  ...
}
```

Теперь запрос PLINQ необходимо информировать о том, что он должен ожидать входящего запроса на отмену выполнения, добавив в цепочку вызов расширяющего метода `WithCancellation()` с передачей ему признака отмены. Кроме того, этот запрос PLINQ понадобится поместить в подходящий блок `try/catch` и обработать возможные исключения. Финальная версия метода `ProcessInData()` выглядит следующим образом:

```
private void ProcessIntData()
{
  // Получить очень большой массив целых чисел.
  int[] source = Enumerable.Range(1, 10000000).ToArray();

  // Найти числа, для которых истинно условие num % 3 == 0,
  // и возвратить их в убывающем порядке.
  int[] modThreeIsZero = null;
  try
  {
    modThreeIsZero = (from num in
      source.AsParallel().WithCancellation(cancelToken.Token)
      where num % 3 == 0 orderby num descending
      select num).ToArray();
    MessageBox.Show(string.Format("Found {0} numbers that match query!",
      modThreeIsZero.Count()));
  }
  catch (OperationCanceledException ex)
  {
    this.Invoke((Action)delegate
    {
      this.Text = ex.Message;
    });
  }
}
```

Исходный код. Проект `PLINQDataProcessingWithCancellation` доступен в подкаталоге `Chapter_19`.

Асинхронные вызовы с помощью ключевого слова async

В этой довольно длинной главе было представлено много сжатых материалов. Конечно, построение, отладка и понимание сложных многопоточных приложений требует усилий в любой инфраструктуре. Хотя TPL, PLINQ и тип делегата могут до некоторой степени упростить решение (особенно по сравнению с другими платформами и языками), разработчики по-прежнему должны хорошо знать детали разнообразных расширенных приемов.

С выходом версии .NET 4.5 в языке программирования C# (а также в VB, если уж на то пошло) появились два новых ключевых слова, которые дополнительно упрощают процесс написания асинхронного кода. В отличие от всех примеров, показанных ранее в главе, когда применяются ключевые слова `async` и `await`, компилятор будет самостоятельно генерировать много кода, связанного с потоками, с использованием многочисленных членов из пространств имен `System.Threading` и `System.Threading.Tasks`.

Знакомство с ключевыми словами async и await языка C#

Ключевое слово `async` языка C# применяется для указания на то, что метод, лямбда-выражение или анонимный метод должен вызываться в асинхронной манере *автоматически*. Да, это правда. Благодаря простой пометке метода с помощью модификатора `async` среда CLR будет создавать новый поток выполнения для обработки текущей задачи. Более того, при вызове метода `async` ключевое слово `await` будет *автоматически* приостанавливать текущий поток до тех пор, пока задача не завершится, давая возможность вызывающему потоку продолжать свою работу.

В целях иллюстрации создадим новый проект приложения Windows Forms по имени `FunWithCSharpAsync` и импортируем пространство имен `System.Threading` в первоначальный файл кода формы (с переименованием исходной формы в `MainForm`). После этого поместим на поверхность визуального конструктора один элемент управления `Button` (по имени `btnCallMethod`) и один элемент управления `TextBox` (по имени `txtInput`), а затем сконфигурируем базовые свойства пользовательского интерфейса (цвета, шрифты, текст) желаемым образом. В обработчике события `Click` кнопки `btnCallMethod` вызовем закрытый вспомогательный метод `DoWork()`, который заставляет вызывающий поток ожидать 10 секунд. Ниже показан код:

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }
  private void btnCallMethod_Click(object sender, EventArgs e)
  {
    this.Text = DoWork();
  }
  private string DoWork()
  {
    Thread.Sleep(10000);
    return "Done with work!";
  }
}
```

Вы знаете, что после запуска программы и щелчка на кнопке придется ожидать 10 секунд до того, как текстовое поле сможет получать клавиатурный ввод. Кроме того, в течение 10 секунд также не будет обновляться заголовок главного окна сообщением `Done with work!`.

Если бы мы решили прибегнуть к одному из описанных ранее приемов, чтобы сделать приложение более отзывчивым, то пришлось бы немало потрудиться. Однако, начиная с .NET 4.5, можно написать следующий код C#:

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }

  private async void btnCallMethod_Click(object sender, EventArgs e)
  {
    this.Text = await DoWork();
  }

  // Ниже приведены пояснения по этому коду...
  private Task<string> DoWork()
  {
    return Task.Run(() =>
    {
      Thread.Sleep(10000);
      return "Done with work!";
    });
  }
}
```

Обратите внимание, что обработчик события `Click` кнопки помечен ключевым словом `async`. Оно указывает, что данный метод должен вызываться в неблокирующей манере. Кроме того, в реализации обработчика события *перед* именем вызываемого метода присутствует ключевое слово `await`. Это важно: если метод декорируется ключевым словом `async`, но не имеет хотя бы одного внутреннего вызова метода с использованием `await`, то получится блокирующий, синхронный вызов (на самом деле компилятор выдаст соответствующее предупреждение).

Нам пришлось применить класс `Task` из пространства имен `System.Threading.Tasks`, чтобы перепроектировать метод `DoWork()` для его функционирования так, как от него ожидается. По существу вместо возвращения специфического значения напрямую (объекта `string` в текущем примере) мы возвращаем объект `Task<T>`, где обобщенный параметр типа `T` представляет собой действительное возвращаемое значение.

Реализация метода `DoWork()` теперь непосредственно возвращает объект `Task<T>`, который является возвращаемым значением `Task.Run()`. Метод `Run()` принимает делегат `Func<>` или `Action<>` и, как вам уже известно, для простоты здесь можно использовать лямбда-выражение. В целом новая версия `DoWork()` может быть описана следующим образом.

> *При вызове запускается новая задача. Эта задача заставляет вызывающий поток уснуть на 10 секунд. По завершении вызывающий поток предоставляет строковое возвращаемое значение. Эта строка помещается в новый объект `Task<string>` и возвращается вызывающему коду.*

Благодаря новой реализации метода `DoWork()` мы можем получить некоторое представление о подлинной роли ключевого слова `await`. Оно всегда будет модифицировать метод, который возвращает объект `Task`. Когда поток выполнения достигает `await`,

вызывающий поток приостанавливается до тех пор, пока вызов не будет завершен. Запустив эту версию приложения, вы обнаружите, что можно щелкнуть на кнопке и сразу же успешно вводить в текстовом поле. Спустя 10 секунд заголовок окна будет обновлен сообщением о завершении работы.

Соглашения об именовании асинхронных методов

А теперь предположим, что новая версия метода `DoWork()` осталась точно такой же, как показанная ранее, но обработчик события `Click` кнопки реализован в следующем виде:

```
private async void btnCallMethod_Click(object sender, EventArgs e)
{
  // Нет ни одного ключевого слова await!
  this.Text = DoWork();
}
```

Обратите внимание, что мы на самом деле пометили метод `DoWork()` ключевым словом `async`, но не указали ключевое слово `await` в его вызове. В таком случае мы получим ошибки на этапе компиляции, потому что возвращаемым значением `DoWork()` является объект `Task`, который мы пытаемся присвоить прямо свойству `Text` (имеющему тип `string`). Вспомните, что ключевое слово `await` отвечает за извлечение внутреннего возвращаемого значения, которое содержится в объекте `Task`. Поскольку `await` отсутствует, возникает несовпадение типов.

На заметку! Метод, поддерживающий `await` — это просто метод, который возвращает `Task<T>`.

С учетом того, что методы, которые возвращают объекты `Task`, теперь могут вызываться в неблокирующей манере посредством конструкций `async` и `await`, в Microsoft рекомендуют (в качестве установившейся практики) снабжать имя любого метода, возвращающего `Task`, суффиксом `Async`. В таком случае разработчики, которым известно данное соглашение об именовании, получают визуальное напоминание о том, что ключевое слово `await` является обязательным, если такой метод планируется вызывать внутри асинхронного контекста.

На заметку! Обработчики событий для элементов управления графического пользовательского интерфейса (вроде нашего обработчика события `Click` кнопки), к которым применяются ключевые слова `async` и `await`, не следуют этому соглашению об именовании.

Более того, метод `DoWork()` также может быть декорирован с помощью ключевых слов `async` и `await` (хотя это не является строго обязательным в текущем примере). С учетом всего сказанного вот как выглядит окончательная модификация текущего примера, которая удовлетворяет рекомендуемым соглашениям об именовании:

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }
  private async void btnCallMethod_Click(object sender, EventArgs e)
  {
    this.Text = await DoWorkAsync();
  }
```

```
  private async Task<string> DoWorkAsync()
  {
    return await Task.Run(() =>
    {
      Thread.Sleep(10000);
      return "Done with work!";
    });
  }
}
```

Асинхронные методы, возвращающие void

В настоящий момент наш метод DoWork() возвращает объект Task, содержащий "действительные данные" для вызывающего кода, которые будут получены прозрачным образом через ключевое слово await. Тем не менее, что если требуется построить асинхронный метод, возвращающий void? В таком случае мы используем необобщенный класс Task и опускаем любые операторы return:

```
private async Task MethodReturningVoidAsync()
{
  await Task.Run(() => { /* Выполнение какой-то работы... */
                         Thread.Sleep(4000);
                       });
}
```

Затем в вызывающем этот метод коде, таком как обработчик события Click второй кнопки, можно применять ключевые слова await и async:

```
private async void btnVoidMethodCall_Click(object sender, EventArgs e)
{
  await MethodReturningVoidAsync();
  MessageBox.Show("Done!");
}
```

Асинхронные методы с множеством контекстов await

Внутри реализации асинхронного метода вполне допустимо иметь множество контекстов await. Предположим, что в приложение добавлен обработчик события Click третьей кнопки, помеченный ключевым словом async. В предыдущих частях этого примера обработчики события Click специально вызывали некоторый внешний метод, который запускал задачу; однако эту логику можно было бы встроить с помощью набора лямбда-выражений:

```
private async void btnMutliAwaits_Click(object sender, EventArgs e)
{
  await Task.Run(() => { Thread.Sleep(2000); });
  MessageBox.Show("Done with first task!");      // завершена первая задача

  await Task.Run(() => { Thread.Sleep(2000); });
  MessageBox.Show("Done with second task!");     // завершена вторая задача

  await Task.Run(() => { Thread.Sleep(2000); });
  MessageBox.Show("Done with third task!");      // завершена третья задача
}
```

Здесь каждая задача всего лишь приостанавливает текущий поток на некоторый период времени; тем не менее, посредством этих задач может быть представлена любая единица работы (обращение к веб-службе, чтение базы данных или что-нибудь еще).

Ниже перечислены ключевые моменты, касающиеся рассматриваемого примера.

- Методы (а также лямбда-выражения или анонимные методы) могут быть помечены ключевым словом `async`, что позволяет методу работать в неблокирующей манере.
- Методы (а также лямбда-выражения или анонимные методы), помеченные ключевым словом `async`, будут выполняться в *блокирующей* манере до тех пор, пока не встретится ключевое слово `await`.
- Один метод `async` может иметь множество контекстов `await`.
- Когда встречается выражение `await`, вызывающий поток приостанавливается до тех пор, пока ожидаемая задача не завершится. Тем временем управление возвращается коду, вызвавшему метод.
- Ключевое слово `await` будет скрывать с глаз возвращаемый объект `Task`, что выглядит как прямой возврат лежащего в основе возвращаемого значения. Методы, не имеющие возвращаемого значения, просто возвращают `void`.
- По соглашению об именовании методы, которые могут быть вызваны асинхронно, должны быть помечены с помощью суффикса `Async`.

Исходный код. Проект `FunWithCSharpAsync` доступен в подкаталоге `Chapter_19`.

Модернизация примера `AddWithThreads` с использованием `asycn/await`

Ранее в главе разрабатывался пример под названием `AddWithThreads` с применением первоначального API-интерфейса для многопоточности в .NET — пространства имен `System.Threading`. Давайте модернизируем этот пример, используя новые ключевые слова `async` и `await` языка C#, чтобы продемонстрировать, насколько яснее может стать логика приложения. Первым делом вспомним, как изначально строился проект `AddWithThreads`.

- Мы создали специальный класс по имени `AddParams`, который представлял данные, подлежащие суммированию.
- Мы применяли класс `Thread` и делегат `ParameterizedThreadStart` для указания на метод `Add()`, принимающий объект `AddParams`.
- Мы использовали класс `AutoResetEvent`, чтобы обеспечить ожидание вызывающим потоком завершения работы вторичного потока.

В общем, нам пришлось приложить немало усилий, чтобы всего лишь подсчитать сумму двух чисел во вторичном потоке выполнения! Ниже приведен код того же самого проекта, переделанный с применением исследуемых приемов (код класса `AddParams` здесь не показан, но вспомните, что он просто содержит два поля, `a` и `b`, для представления суммируемых данных):

```
class Program
{
  static void Main(string[] args)
  {
    AddAsync();
    Console.ReadLine();
  }
```

```
private static async Task AddAsync()
{
  Console.WriteLine("***** Adding with Thread objects *****");
  Console.WriteLine("ID of thread in Main(): {0}",
    Thread.CurrentThread.ManagedThreadId);

  AddParams ap = new AddParams(10, 10);
  await Sum(ap);

  Console.WriteLine("Other thread is done!");
}

static async Task Sum(object data)
{
  await Task.Run(() =>
  {
    if (data is AddParams)
    {
      Console.WriteLine("ID of thread in Add(): {0}",
        Thread.CurrentThread.ManagedThreadId);

      AddParams ap = (AddParams)data;
      Console.WriteLine("{0} + {1} is {2}",
        ap.a, ap.b, ap.a + ap.b);
    }
  });
}
}
```

Первое, что следует отметить — код, который изначально находился в `Main()`, был перемещен в новый метод по имени `AddAsync()`. Причина связана не только с соблюдением ожидаемого соглашения об именовании, но также и со следующим важным моментом.

На заметку! Метод `Main()` исполняемой сборки не может быть помечен с помощью ключевого слова `async`.

Обратите внимание, что метод `AddAsync()` помечен ключевым словом `async` и в нем определен контекст `await`. Кроме того, метод `Sum()` порождает новую задачу для выполнения единицы работы. В любом случае после запуска этой программы выясняется, что 10 плюс 10 по-прежнему равно 20. Однако теперь имеются два уникальных идентификатора потоков.

```
***** Adding with Thread objects *****
ID of thread in Main(): 1
ID of thread in Add(): 3
10 + 10 is 20
Other thread is done!
```

Исходный код. Проект `AddWithThreadsAsync` доступен в подкаталоге `Chapter_19`.

Итак, как видите, ключевые слова `async` и `await` содействуют упрощению процесса вызова методов во вторичном потоке выполнения. В то время как мы рассмотрели лишь несколько примеров того, что можно делать с помощью этого аспекта языка C#, у вас появилась прочная основа для дальнейших исследований.

Резюме

Эта глава началась с исследования особенностей конфигурирования типов делегатов .NET для выполнения метода в асинхронной манере. Вы видели, что методы `BeginInvoke()` и `EndInvoke()` позволяют косвенно манипулировать вторичным потоком с минимальными усилиями. В ходе обсуждения были представлены интерфейс `IAsyncResult` и класс `AsyncResult`. Вы узнали, что эти типы предлагают разнообразные способы синхронизации вызывающего потока и получения возможных возвращаемых значений методов.

Следующая порция главы была посвящена выяснению роли пространства имен `System.Threading`. Как было показано, когда приложение создает дополнительные потоки выполнения, в результате появляется возможность выполнять множество задач (по внешнему виду) одновременно. Также было продемонстрировано несколько способов защиты чувствительных к потокам блоков кода с целью предотвращения повреждения разделяемых ресурсов.

Затем в главе исследовались новые модели для разработки многопоточных приложений, введенные в .NET 4.0, в частности, Task Parallel Library и PLINQ. В завершение главы была раскрыта роль ключевых слов `async` и `await`. Вы видели, что эти ключевые слова используются многими типами в библиотеке TPL; тем не менее, большинство работ по созданию сложного кода для многопоточной обработки и синхронизации компилятор выполняет самостоятельно.

ГЛАВА 20

Файловый ввод-вывод и сериализация объектов

При создании настольных приложений возможность сохранения информации между пользовательскими сеансами является привычным делом. В этой главе рассматривается несколько тем, касающихся ввода-вывода, с точки зрения платформы .NET Framework. Первая задача связана с исследованием основных типов, определенных в пространстве имен `System.IO`, с помощью которых можно программно модифицировать структуру каталогов и файлов. Вторая задача предусматривает изучение разнообразных способов чтения и записи символьных, двоичных, строковых и находящихся в памяти структур данных.

После изучения способов манипулирования файлами и каталогами с использованием основных типов ввода-вывода вы ознакомитесь со связанной темой — *сериализацией объектов*. Сериализацию объектов можно применять для сохранения и извлечения состояния объекта с помощью любого типа, производного от `System.IO.Stream`. Возможность сериализации объектов критична, когда объект необходимо копировать на удаленную машину, используя различные технологии удаленного взаимодействия, такие как Windows Communication Foundation. Однако сериализация довольно полезна сама по себе и, скорее всего, пригодится во многих разрабатываемых приложениях .NET (распределенных или нет).

На заметку! Чтобы можно было успешно выполнять примеры в главе, IDE-среда Visual Studio должна быть запущена с правами администратора (для этого просто щелкните правой кнопкой мыши на значке Visual Studio и выберите в контекстном меню пункт Запуск от имени администратора). В противном случае при доступе к файловой системе компьютера могут возникать исключения, связанные с безопасностью.

Исследование пространства имен `System.IO`

В рамках платформы .NET пространство имен `System.IO` представляет собой раздел библиотек базовых классов, выделенный службам файлового ввода и вывода, а также ввода и вывода в памяти. Подобно любому пространству имен в `System.IO` определен набор классов, интерфейсов, перечислений, структур и делегатов, большинство из которых находятся в сборке `mscorlib.dll`. В дополнение к типам, содержащимся внутри `mscorlib.dll`, в сборке `System.dll` определены дополнительные члены пространства имен `System.IO`. Обратите внимание, что во всех проектах Visual Studio автоматически устанавливаются ссылки на обе сборки.

Многие типы из пространства имен `System.IO` сосредоточены на программной манипуляции физическими каталогами и файлами. Тем не менее, дополнительные типы предоставляют поддержку чтения и записи данных в строковые буферы, а также в области памяти. В табл. 20.1 кратко описаны основные (неабстрактные) классы, которые дают понятие о функциональности, доступной в пространстве имен `System.IO`.

Таблица 20.1. Основные члены пространства имен `System.IO`

Неабстрактные классы ввода-вывода	Описание
`BinaryReader` `BinaryWriter`	Эти классы позволяют сохранять и извлекать данные элементарных типов (целочисленные, булевские, строковые и т.д.) как двоичные значения
`BufferedStream`	Этот класс предоставляет временное хранилище для потока байтов, который может быть зафиксирован в постоянном хранилище в более позднее время
`Directory` `DirectoryInfo`	Эти классы применяются для манипулирования структурой каталогов машины. Тип `Directory` открывает функциональность с использованием *статических членов*. Тип `DirectoryInfo` обеспечивает аналогичную функциональность через действительную *объектную ссылку*
`DriveInfo`	Этот класс предоставляет детальную информацию о дисковых устройствах, присутствующих на заданной машине
`File` `FileInfo`	Эти классы служат для манипулирования набором файлов на машине. Тип `File` открывает функциональность через *статические члены*. Тип `FileInfo` обеспечивает аналогичную функциональность через действительную *объектную ссылку*
`FileStream`	Этот класс предоставляет произвольный доступ к файлу (например, с возможностями поиска) с данными, представленными в виде потока байтов
`FileSystemWatcher`	Этот класс позволяет отслеживать модификацию внешних файлов в указанном каталоге
`MemoryStream`	Этот класс обеспечивает произвольный доступ к данным, хранящимся в памяти, а не в физическом файле
`Path`	Этот класс выполняет операции над типами `System.String`, которые содержат информацию о пути к файлу или каталогу, в независимой от платформы манере
`StreamWriter` `StreamReader`	Эти классы применяются для хранения (и извлечения) текстовой информации в файле. Они не поддерживают произвольный доступ к файлу
`StringWriter` `StringReader`	Подобно `StreamWriter/StreamReader` эти классы также работают с текстовой информацией. Однако лежащим в основе хранилищем является строковый буфер, а не физический файл

В дополнение к перечисленным конкретным типам классов в `System.IO` определено несколько перечислений, а также набор абстрактных классов (скажем, `Stream`, `TextReader` и `TextWriter`), которые формируют разделяемый полиморфный интерфейс для всех наследников. В этой главе вы узнаете о многих типах пространства имен `System.IO`.

Классы Directory (DirectoryInfo) и File (FileInfo)

Пространство имен System.IO предлагает четыре класса, которые позволяют манипулировать индивидуальными файлами, а также взаимодействовать со структурой каталогов машины. Первые два класса, Directory и File, открывают доступ к операциям создания, удаления, копирования и перемещения через разнообразные статические члены. Тесно связанные с ними классы FileInfo и DirectoryInfo обеспечивают похожую функциональность в виде методов уровня экземпляра (следовательно, придется создавать их экземпляры с помощью ключевого слова new). На рис. 20.1 видно, что Directory и File расширяют непосредственно класс System.Object, в то время как DirectoryInfo и FileInfo являются производными от абстрактного класса FileSystemInfo.

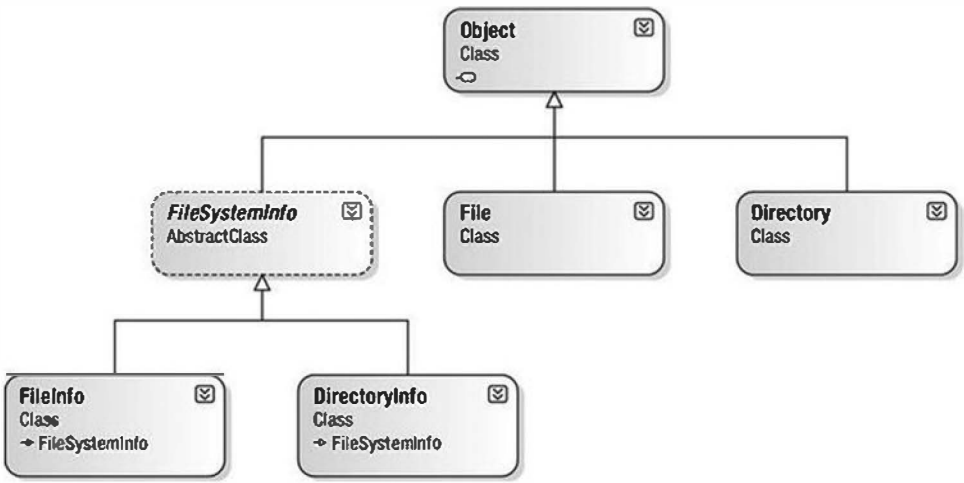


Рис. 20.1. Классы для работы с файлами и каталогами

Обычно FileInfo и DirectoryInfo считаются лучшим выбором для получения полных сведений о файле или каталоге (например, времени создания, возможностей чтения/записи и т.п.), т.к. их члены возвращают строго типизированные объекты. В отличие от этого члены классов Directory и File, как правило, возвращают простые строковые значения, а не строго типизированные объекты. Тем не менее, это всего лишь рекомендация; во многих случаях одну и ту же работу можно делать с использованием File/FileInfo или Directory/DirectoryInfo.

Абстрактный базовый класс FileSystemInfo

Классы DirectoryInfo и FileInfo получают многие линии поведения от абстрактного базового класса FileSystemInfo. По большей части члены класса FileSystemInfo применяются для выяснения общих характеристик (таких как время создания, разнообразные атрибуты и т.д.) заданного файла или каталога. В табл. 20.2 перечислены некоторые основные свойства, представляющие интерес.

В классе FileSystemInfo также определен метод Delete(). Он реализуется производными типами для удаления заданного файла или каталога с жесткого диска. Кроме того, перед получением информации об атрибутах можно вызвать метод Refresh(), чтобы обеспечить актуальность статистики о текущем файле или каталоге.

Таблица 20.2. Избранные свойства класса `FileSystemInfo`

Свойство	Описание
`Attributes`	Получает или устанавливает ассоциированные с текущим файлом атрибуты, которые представлены перечислением `FileAttributes` (например, доступный только для чтения, зашифрованный, скрытый или сжатый)
`CreationTime`	Получает или устанавливает время создания текущего файла или каталога
`Exists`	Определяет, существует ли данный файл или каталог
`Extension`	Извлекает расширение файла
`FullName`	Получает полный путь к файлу или каталогу
`LastAccessTime`	Получает или устанавливает время последнего доступа к текущему файлу или каталогу
`LastWriteTime`	Получает или устанавливает время последней записи в текущий файл или каталог
`Name`	Получает имя текущего файла или каталога

Работа с типом `DirectoryInfo`

Первый неабстрактный тип, связанный с вводом-выводом, который мы исследуем здесь — `DirectoryInfo`. Этот класс содержит набор членов, используемых для создания, перемещения, удаления и перечисления каталогов и подкаталогов. В дополнение к функциональности, предоставленной его базовым классом (`FileSystemInfo`), класс `DirectoryInfo` предлагает ключевые члены, описанные в табл. 20.3.

Таблица 20.3. Основные члены типа `DirectoryInfo`

Член	Описание
`Create()` `CreateSubdirectory()`	Создает каталог (или набор подкаталогов) по заданному путевому имени
`Delete()`	Удаляет каталог и все его содержимое
`GetDirectories()`	Возвращает массив объектов `DirectoryInfo`, которые представляют все подкаталоги в текущем каталоге
`GetFiles()`	Извлекает массив объектов `FileInfo`, представляющий набор файлов в заданном каталоге
`MoveTo()`	Перемещает каталог со всем содержимым в новый путь
`Parent`	Извлекает родительский каталог данного каталога
`Root`	Получает корневую часть пути

Работа с типом `DirectoryInfo` начинается с указания отдельного пути в параметре конструктора. Если требуется получить доступ к текущему рабочему каталогу (каталогу выполняющегося приложения), то следует применять обозначение в виде точки (`.`). Вот некоторые примеры:

```
// Привязаться к текущему рабочему каталогу.
DirectoryInfo dir1 = new DirectoryInfo(".");

// Привязаться к C:\Windows, используя дословную строку.
DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");
```

Во втором примере предполагается, что путь, передаваемый конструктору (C:\Windows), уже существует на физической машине. Однако при попытке взаимодействия с несуществующим каталогом генерируется исключение System.IO.DirectoryNotFoundException. Таким образом, чтобы указать каталог, который пока еще не создан, перед работой с ним понадобится вызвать метод Create():

```
// Привязаться к несуществующему каталогу, затем создать его.
DirectoryInfo dir3 = new DirectoryInfo(@"C:\MyCode\Testing");
dir3.Create();
```

После создания объекта DirectoryInfo можно исследовать содержимое лежащего в основе каталога с помощью любого свойства, унаследованного от FileSystemInfo. В целях иллюстрации создадим новый проект консольного приложения по имени DirectoryApp и импортируем в файл кода C# пространство имен System.IO. Далее модифицируем класс Program, добавив показанный ниже новый статический метод, который создает объект DirectoryInfo, отображенный на C:\Windows (при необходимости подкорректируйте путь), и выводит интересные статистические данные:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Directory(Info) *****\n");
    ShowWindowsDirectoryInfo();
    Console.ReadLine();
  }
  static void ShowWindowsDirectoryInfo()
  {
    // Вывести информацию о каталоге.
    DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");
    Console.WriteLine("***** Directory Info *****");
    Console.WriteLine("FullName: {0}", dir.FullName); // полное имя
    Console.WriteLine("Name: {0}", dir.Name);         // имя каталога
    Console.WriteLine("Parent: {0}", dir.Parent);     // родительский каталог
    Console.WriteLine("Creation: {0}", dir.CreationTime); // время создания
    Console.WriteLine("Attributes: {0}", dir.Attributes); // атрибуты
    Console.WriteLine("Root: {0}", dir.Root);         // корневой каталог
    Console.WriteLine("**************************\n");
  }
}
```

Вывод у вас может быть другим, но должен выглядеть похожим:

```
***** Fun with Directory(Info) *****

***** Directory Info *****
FullName: C:\Windows
Name: Windows
Parent:
Creation: 10/10/2015 10:22:32 PM
Attributes: Directory
Root: C:\
**************************
```

Перечисление файлов с помощью типа DirectoryInfo

В дополнение к получению базовых сведений о существующем каталоге текущий пример можно расширить, чтобы задействовать некоторые методы типа DirectoryInfo.

Первым делом мы используем метод `GetFiles()` для получения информации обо всех файлах `*.jpg`, расположенных в каталоге `C:\Windows\Web\Wallpaper`.

На заметку! Если на вашей машине нет каталога `C:\Windows\Web\Wallpaper`, то скорректируйте код так, чтобы в нем читались файлы из какого-то существующего каталога (например, все файлы `*.bmp` из каталога `C:\Windows`).

Метод `GetFiles()` возвращает массив объектов `FileInfo`, каждый из которых открывает доступ к детальной информации о конкретном файле (тип `FileInfo` будет подробно описан далее в главе). Предположим, что в методе `Main()` вызывается следующий статический метод класса `Program`:

```
static void DisplayImageFiles()
{
  DirectoryInfo dir = new DirectoryInfo(@"C:\Windows\Web\Wallpaper");

  // Получить все файлы с расширением *.jpg.
  FileInfo[] imageFiles = dir.GetFiles("*.jpg", SearchOption.AllDirectories);

  // Сколько файлов найдено?
  Console.WriteLine("Found {0} *.jpg files\n", imageFiles.Length);

  // Вывести информацию о каждом файле.
  foreach (FileInfo f in imageFiles)
  {
    Console.WriteLine("***************************");
    Console.WriteLine("File name: {0}", f.Name);          // имя файла
    Console.WriteLine("File size: {0}", f.Length);        // размер
    Console.WriteLine("Creation: {0}", f.CreationTime);   // время создания
    Console.WriteLine("Attributes: {0}", f.Attributes);   // атрибуты
    Console.WriteLine("***************************\n");
  }
}
```

Обратите внимание на указание в вызове `GetFiles()` варианта поиска: `SearchOption.AllDirectories` обеспечивает просмотр всех подкаталогов корня. В результате запуска приложения выводится список файлов, которые соответствуют поисковому шаблону.

Создание подкаталогов с помощью типа `DirectoryInfo`

Посредством метода `DirectoryInfo.CreateSubdirectory()` можно программно расширять структуру каталогов. Он позволяет создавать одиночный подкаталог, а также множество вложенных подкаталогов в единственном вызове. В приведенном далее методе демонстрируется расширение структуры диска `C:` несколькими специальными подкаталогами:

```
static void ModifyAppDirectory()
{
  DirectoryInfo dir = new DirectoryInfo(@"C:\");

  // Создать \MyFolder в каталоге приложения.
  dir.CreateSubdirectory("MyFolder");

  // Создать \MyFolder2\Data в каталоге приложения.
  dir.CreateSubdirectory(@"MyFolder2\Data");
}
```

Получать возвращаемое значение метода `CreateSubdirectory()` не обязательно, но важно знать, что в случае его успешного выполнения возвращается объект `DirectoryInfo`, представляющий вновь созданный элемент. Взгляните на следующую модификацию предыдущего метода. Обратите внимание на указание строки `"."` в конструкторе `DirectoryInfo`, что дает доступ к месту установки приложения.

```
static void ModifyAppDirectory()
{
  DirectoryInfo dir = new DirectoryInfo(".");

  // Создать \MyFolder в начальном каталоге.
  dir.CreateSubdirectory("MyFolder");

  // Получить возвращенный объект DirectoryInfo.
  DirectoryInfo myDataFolder = dir.CreateSubdirectory(@"MyFolder2\Data");

  // Выводит путь к ..\MyFolder2\Data.
  Console.WriteLine("New Folder is: {0}", myDataFolder);
}
```

Вызвав этот метод в `Main()` и запустив программу, в проводнике Windows можно будет увидеть новые подкаталоги (рис. 20.2).

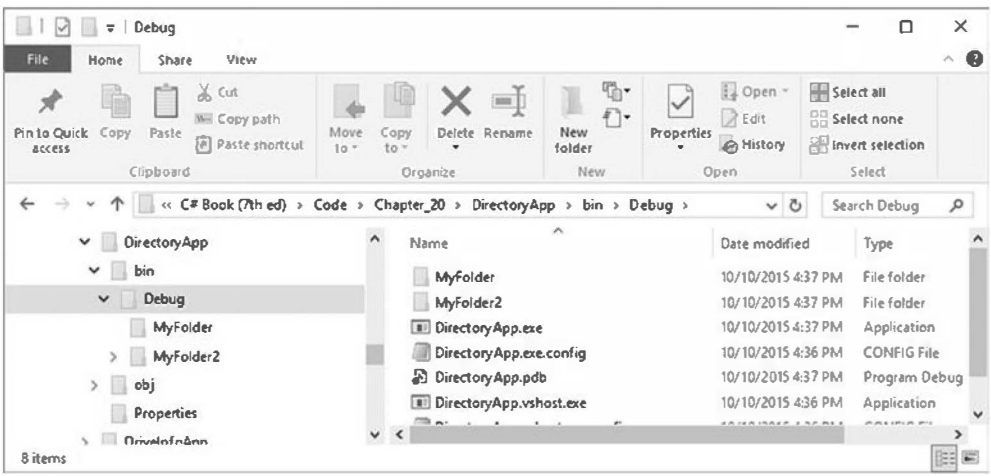


Рис. 20.2. Результат создания подкаталогов

Работа с типом `Directory`

Вы видели тип `DirectoryInfo` в действии и теперь готовы к изучению типа `Directory`. По большей части статические члены типа `Directory` воспроизводят функциональность, которая предоставляется членами уровня экземпляра, определенными в `DirectoryInfo`. Тем не менее, вспомните, что члены типа `Directory` обычно возвращают строковые данные, а не строго типизированные объекты `FileInfo`/`DirectoryInfo`.

Давайте взглянем на функциональность типа `Directory`; показанный ниже вспомогательный метод отображает имена всех логических устройств на текущем компьютере (с помощью метода `Directory.GetLogicalDrives()`) и применяет статический метод `Directory.Delete()` для удаления созданных ранее подкаталогов `\MyFolder` и `\MyFolder2\Data`:

```
static void FunWithDirectoryType()
{
  // Вывести список всех логических устройств на текущем компьютере.
  string[] drives = Directory.GetLogicalDrives();
  Console.WriteLine("Here are your drives:");
  foreach (string s in drives)
    Console.WriteLine("--> {0} ", s);
  // Удалить ранее созданные подкаталоги.
  Console.WriteLine("Press Enter to delete directories");
  Console.ReadLine();
  try
  {
    Directory.Delete(@"C:\MyFolder");
    // Второй параметр указывает, нужно ли удалять внутренние подкаталоги.
    Directory.Delete(@"C:\MyFolder2", true);
  }
  catch (IOException e)
  {
    Console.WriteLine(e.Message);
  }
}
```

Исходный код. Проект `DirectoryApp` доступен в подкаталоге `Chapter_20`.

Работа с типом `DriveInfo`

Пространство имен `System.IO` содержит класс по имени `DriveInfo`. Подобно `Directory.GetLogicalDrives()` статический метод `DriveInfo.GetDrives()` позволяет выяснить имена устройств на машине.

Однако в отличие от `Directory.GetLogicalDrives()` метод `DriveInfo.GetDrives()` предоставляет множество дополнительных деталей (например, тип устройства, доступное свободное пространство и метка тома). Рассмотрим следующий класс `Program`, определенный в новом проекте консольного приложения `DriveInfoApp` (не забудьте импортировать пространство имен `System.IO`):

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with DriveInfo *****\n");
    // Получить информацию обо всех устройствах.
    DriveInfo[] myDrives = DriveInfo.GetDrives();
    // Вывести сведения об устройствах.
    foreach(DriveInfo d in myDrives)
    {
      Console.WriteLine("Name: {0}", d.Name);          // имя
      Console.WriteLine("Type: {0}", d.DriveType);     // тип
      // Проверить, смонтировано ли устройство.
      if(d.IsReady)
      {
        Console.WriteLine("Free space: {0}", d.TotalFreeSpace);
                    // Свободное пространство
        Console.WriteLine("Format: {0}", d.DriveFormat); // формат устройства
```

```
            Console.WriteLine("Label: {0}", d.VolumeLabel);  // метка тома
          }
          Console.WriteLine();
        }
        Console.ReadLine();
      }
    }
```

Вот возможный вывод:

```
***** Fun with DriveInfo *****

Name: C:\
Type: Fixed
Free space: 791699763200
Format: NTFS
Label: Windows10_OS

Name: D:\
Type: Fixed
Free space: 23804067840
Format: NTFS
Label: LENOVO

Press any key to continue . . .
```

К этому моменту вы изучили несколько основных линий поведения классов `Directory`, `DirectoryInfo` и `DriveInfo`. Далее вы ознакомитесь с тем, как создавать, открывать, закрывать и удалять файлы, находящиеся в заданном каталоге.

Исходный код. Проект `DriveInfoApp` доступен в подкаталоге `Chapter_20`.

Работа с классом `FileInfo`

Как было показано в предыдущем примере `DirectoryApp`, класс `FileInfo` позволяет получать сведения о существующих файлах на жестком диске (такие как время создания, размер и атрибуты) и помогает создавать, копировать, перемещать и удалять файлы. В дополнение к набору функциональности, унаследованной от `FileSystemInfo`, класс `FileInfo` имеет ряд уникальных членов, которые описаны в табл. 20.4.

Таблица 20.4. Основные члены `FileInfo`

Член	Описание
`AppendText()`	Создает объект `StreamWriter` (описанный далее в главе) и добавляет текст в файл
`CopyTo()`	Копирует существующий файл в новый файл
`Create()`	Создает новый файл и возвращает объект `FileStream` (описанный далее в главе) для взаимодействия с вновь созданным файлом
`CreateText()`	Создает объект `StreamWriter`, который производит запись в новый текстовый файл
`Delete()`	Удаляет файл, к которому привязан экземпляр `FileInfo`
`Directory`	Получает экземпляр родительского каталога
`DirectoryName`	Получает полный путь к родительскому каталогу

Окончание табл. 20.4

Член	Описание
`Length`	Получает размер текущего файла
`MoveTo()`	Перемещает указанный файл в новое местоположение, предоставляя возможность указания нового имени для файла
`Name`	Получает имя файла
`Open()`	Открывает файл с разнообразными привилегиями чтения/записи и совместного доступа
`OpenRead()`	Создает объект `FileStream`, доступный только для чтения
`OpenText()`	Создает объект `StreamReader` (описанный далее в главе), который производит чтение из существующего текстового файла
`OpenWrite()`	Создает объект `FileStream`, доступный только для записи

Обратите внимание, что большинство методов класса `FileInfo` возвращают специфический объект ввода-вывода (например, `FileStream` и `StreamWriter`), который позволяет начать чтение и запись данных в ассоциированный файл во множестве форматов. Вскоре мы исследуем эти типы, но прежде чем рассмотреть работающий пример, давайте изучим различные способы получения дескриптора файла с использованием класса `FileInfo`.

Метод `FileInfo.Create()`

Один из способов создания дескриптора файла предусматривает применение метода `FileInfo.Create()`:

```
static void Main(string[] args)
{
  // Создать новый файл на диске C:.
  FileInfo f = new FileInfo(@"C:\Test.dat");
  FileStream fs = f.Create();
  // Использовать объект FileStream...
  // Закрыть файловый поток.
  fs.Close();
}
```

Метод `FileInfo.Create()` возвращает тип `FileStream`, который предоставляет синхронную и асинхронную операции записи/чтения лежащего в его основе файла. Имейте в виду, что объект `FileStream`, возвращаемый `FileInfo.Create()`, открывает полный доступ по чтению и записи всем пользователям.

Также обратите внимание, что после окончания работы с текущим объектом `FileStream` необходимо обеспечить закрытие его дескриптора для освобождения внутренних неуправляемых ресурсов потока. Учитывая, что `FileStream` реализует интерфейс `IDisposable`, можно использовать блок `using` и позволить компилятору сгенерировать логику завершения (подробности ищите в главе 8):

```
static void Main(string[] args)
{
  // Определение блока using для типов файлового ввода-вывода.
  FileInfo f = new FileInfo(@"C:\Test.dat");
  using (FileStream fs = f.Create())
  {
    // Использовать объект FileStream...
  }
}
```

Метод FileInfo.Open()

С помощью метода FileInfo.Open() можно открывать существующие файлы, а также создавать новые файлы с гораздо более высокой точностью, чем FileInfo.Create(), поскольку Open() обычно принимает несколько параметров для описания общей структуры файла, с которым будет производиться работа. В результате вызова Open() возвращается объект FileStream. Взгляните на следующий код:

```
static void Main(string[] args)
{
  // Создать новый файл посредством FileInfo.Open().
  FileInfo f2 = new FileInfo(@"C:\Test2.dat");
  using(FileStream fs2 = f2.Open(FileMode.OpenOrCreate,
    FileAccess.ReadWrite, FileShare.None))
  {
    // Использовать объект FileStream...
  }
}
```

Эта версия перегруженного метода Open() требует передачи трех параметров. Первый параметр указывает общий тип запроса ввода-вывода (например, создать новый файл, открыть существующий файл или дописать в файл), который представлен в виде перечисления FileMode (описание его членов приведено в табл. 20.5):

```
public enum FileMode
{
  CreateNew,
  Create,
  Open,
  OpenOrCreate,
  Truncate,
  Append
}
```

Таблица 20.5. Члены перечисления FileMode

Член	Описание
CreateNew	Информирует операционную систему о необходимости создания нового файла. Если файл уже существует, генерируется исключение IOException
Create	Информирует операционную систему о необходимости создания нового файла. Если файл уже существует, он будет перезаписан
Open	Открывает существующий файл. Если файл не существует, генерируется исключение FileNotFoundException
OpenOrCreate	Открывает файл, если он существует; в противном случае создает новый
Truncate	Открывает файл и усекает его до нулевой длины
Append	Открывает файл, переходит в его конец и начинает операции записи (этот флаг может применяться лишь с потоками только для записи). Если файл не существует, создается новый файл

Второй параметр метода Open() — значение перечисления FileAccess — служит для определения поведения чтения/записи лежащего в основе потока:

```
public enum FileAccess
{
  Read,
  Write,
  ReadWrite
}
```

Наконец, третий параметр метода `Open()` — значение перечисления `FileShare` — указывает, каким образом файл может совместно использоваться другими файловыми дескрипторами:

```
public enum FileShare
{
  Delete,
  Inheritable,
  None,
  Read,
  ReadWrite,
  Write
}
```

Методы `FileOpen.OpenRead()` и `FileInfo.OpenWrite()`

Метод `FileOpen.Open()` позволяет получить дескриптор файла в гибкой манере, но класс `FileInfo` также предлагает методы `OpenRead()` и `OpenWrite()`. Как и можно было ожидать, эти методы возвращают подходящим образом сконфигурированный только для чтения или только для записи объект `FileStream` без необходимости в предоставлении значений разных перечислений. Подобно `FileInfo.Create()` и `FileInfo.Open()` методы `OpenRead()` и `OpenWrite()` возвращают объект `FileStream` (обратите внимание, что в следующем коде предполагается наличие на диске C: файлов `Test3.dat` и `Test4.dat`):

```
static void Main(string[] args)
{
  // Получить объект FileStream с правами только для чтения.
  FileInfo f3 = new FileInfo(@"C:\Test3.dat");
  using(FileStream readOnlyStream = f3.OpenRead())
  {
    // Использовать объект FileStream...
  }
  // Теперь получить объект FileStream с правами только для записи.
  FileInfo f4 = new FileInfo(@"C:\Test4.dat");
  using(FileStream writeOnlyStream = f4.OpenWrite())
  {
    // Использовать объект FileStream...
  }
}
```

Метод `FileInfo.OpenText()`

Еще одним членом типа `FileInfo`, связанным с открытием файлов, является `OpenText()`. В отличие от `Create()`, `Open()`, `OpenRead()` и `OpenWrite()` метод `OpenText()` возвращает экземпляр типа `StreamReader`, а не `FileStream`. Исходя из того, что на диске C: имеется файл по имени `boot.ini`, вот как получить доступ к его содержимому:

```
static void Main(string[] args)
{
  // Получить объект StreamReader.
  FileInfo f5 = new FileInfo(@"C:\boot.ini");
  using(StreamReader sreader = f5.OpenText())
  {
    // Использовать объект StreamReader...
  }
}
```

Вскоре вы увидите, что тип `StreamReader` предоставляет способ чтения символьных данных из лежащего в основе файла.

Методы `FileInfo.CreateText()` и `FileInfo.AppendText()`

Последними двумя методами, представляющими в данный момент интерес, являются `CreateText()` и `AppendText()`. Оба они возвращают объект `StreamWriter`:

```
static void Main(string[] args)
{
  FileInfo f6 = new FileInfo(@"C:\Test6.txt");
  using(StreamWriter swriter = f6.CreateText())
  {
    // Использовать объект StreamWriter...
  }
  FileInfo f7 = new FileInfo(@"C:\FinalTest.txt");
  using(StreamWriter swriterAppend = f7.AppendText())
  {
    // Использовать объект StreamWriter...
  }
}
```

Как и можно было ожидать, тип `StreamWriter` предлагает способ записи данных в связанный с ним файл.

Работа с типом `File`

В типе `File` определено несколько статических методов для предоставления функциональности, почти идентичной той, которая доступна в типе `FileInfo`. Подобно `FileInfo` тип `File` поддерживает методы `AppendText()`, `Create()`, `CreateText()`, `Open()`, `OpenRead()`, `OpenWrite()` и `OpenText()`. Во многих случаях типы `File` и `FileInfo` могут применяться взаимозаменяемо. Чтобы увидеть это в действии, упростим каждый из приведенных ранее примеров использования типа `FileStream` за счет применения вместо него типа `File`:

```
static void Main(string[] args)
{
  // Получить объект FileStream через File.Create().
  using(FileStream fs = File.Create(@"C:\Test.dat"))
  {}
  // Получить объект FileStream посредством File.Open().
  using(FileStream fs2 = File.Open(@"C:\Test2.dat",
    FileMode.OpenOrCreate,
    FileAccess.ReadWrite, FileShare.None))
  {}
```

```
    // Получить объект FileStream с правами только для чтения.
    using(FileStream readOnlyStream = File.OpenRead(@"Test3.dat"))
    {}
    // Получить объект FileStream с правами только для записи.
    using(FileStream writeOnlyStream = File.OpenWrite(@"Test4.dat"))
    {}
    // Получить объект StreamReader.
    using(StreamReader sreader = File.OpenText(@"C:\boot.ini"))
    {}
    // Получить несколько объектов StreamWriter.
    using(StreamWriter swriter = File.CreateText(@"C:\Test6.txt"))
    {}
    using(StreamWriter swriterAppend = File.AppendText(@"C:\FinalTest.txt"))
    {}
  }
```

Дополнительные члены `File`

Тип `File` также поддерживает несколько членов, описанных в табл. 20.6, которые могут значительно упростить процессы чтения и записи текстовых данных.

Таблица 20.6. Методы типа `File`

Метод	Описание
`ReadAllBytes()`	Открывает указанный файл, возвращает двоичные данные в виде массива байтов и закрывает файл
`ReadAllLines()`	Открывает указанный файл, возвращает символьные данные в виде массива строк и закрывает файл
`ReadAllText()`	Открывает указанный файл, возвращает символьные данные в виде объекта `System.String` и закрывает файл
`WriteAllBytes()`	Открывает указанный файл, записывает в него массив байтов и закрывает файл
`WriteAllLines()`	Открывает указанный файл, записывает в него массив строк и закрывает файл
`WriteAllText()`	Открывает указанный файл, записывает в него символьные данные из заданной строки и закрывает файл

Приведенные в табл. 20.6 методы типа `File` можно использовать для реализации чтения и записи пакетов данных посредством всего нескольких строк кода. Еще лучше то, что эти методы автоматически закрывают лежащий в основе файловый дескриптор. Например, следующий проект консольного приложения (по имени `SimpleFileIO`) сохраняет строковые данные в новом файле на диске C: (и читает их в память) с минимальными усилиями (здесь предполагается, что было импортировано пространство имен `System.IO`):

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Simple I/O with the File Type *****\n");
    string[] myTasks = {
      "Fix bathroom sink", "Call Dave",
      "Call Mom and Dad", "Play Xbox One"};
```

```
      // Записать все данные в файл на диске C:.
      File.WriteAllLines(@"C:\tasks.txt", myTasks);
      // Прочитать все данные и вывести на консоль.
      foreach (string task in File.ReadAllLines(@"C:\tasks.txt"))
      {
        Console.WriteLine("TODO: {0}", task);
      }
      Console.ReadLine();
    }
  }
```

Из продемонстрированного примера можно сделать вывод: когда необходимо быстро получить файловый дескриптор, тип `File` позволит сэкономить на объеме кодирования. Тем не менее, преимущество предварительного создания объекта `FileInfo` заключается в возможности сбора сведений о файле с применением членов абстрактного базового класса `FileSystemInfo`.

Исходный код. Проект `SimpleFileIO` доступен в подкаталоге `Chapter_20`.

Абстрактный класс `Stream`

Вы уже видели много способов получения объектов `FileStream`, `StreamReader` и `StreamWriter`, но нужно еще читать данные или записывать их в файл, используя упомянутые типы. Чтобы понять, как это делается, необходимо освоить концепцию потока. В мире манипуляций вводом-выводом *поток* (stream) представляет порцию данных, протекающую между источником и приемником. Потоки предоставляют общий способ взаимодействия с *последовательностью байтов* независимо от того, какого рода устройство (скажем, файл, сетевое подключение или принтер) хранит или отображает эти байты.

Абстрактный класс `System.IO.Stream` определяет набор членов, которые обеспечивают поддержку синхронного и асинхронного взаимодействия с хранилищем (например, файлом или областью памяти).

На заметку! Концепция потока не ограничена файловым вводом-выводом. Конечно, библиотеки .NET предоставляют потоковый доступ к сетям, областям памяти и прочим абстракциям, связанным с потоками.

Потомки класса `Stream` представляют данные в виде низкоуровневых потоков байтов; следовательно, работа непосредственно с низкоуровневыми потоками может оказаться не особенно понятной. Некоторые типы, производные от `Stream`, поддерживают *позиционирование*, которое означает процесс получения и корректировки текущей позиции в потоке. В табл. 20.7 приведено описание основных членов класса `Stream`, что помогает понять его функциональность.

Таблица 20.7. Члены абстрактного класса `Stream`

Член	Описание

`CanRead`
`CanWrite`
`CanSeek`
	Определяют, поддерживает ли текущий поток чтение, поиск и/или запись

`Close()`	Закрывает текущий поток и освобождает все ресурсы (такие как сокеты и файловые дескрипторы), ассоциированные с текущим потоком. Внутренне этот метод является псевдонимом `Dispose()`, поэтому *закрытие потока* функционально эквивалентно *освобождению потока*

Окончание табл. 20.7

Член	Описание
`Flush()`	Обновляет лежащий в основе источник данных или хранилище текущим состоянием буфера и затем очищает буфер. Если поток не реализует буфер, то этот метод ничего не делает
`Length`	Возвращает длину потока в байтах
`Position`	Определяет текущую позицию в потоке

`Read()`
`ReadByte()`
`ReadAsync()`
	Читают последовательность байтов (или одиночный байт) из текущего потока и перемещают текущую позицию потока вперед на количество прочитанных байтов

`Seek()`	Устанавливает позицию в текущем потоке
`SetLength()`	Устанавливает длину текущего потока

`Write()`
`WriteByte()`
`WrriteAsync()`
	Записывают последовательность байтов (или одиночный байт) в текущий поток и перемещают текущую позицию вперед на количество записанных байтов

Работа с классом `FileStream`

Класс `FileStream` предоставляет реализацию абстрактных членов `Stream` в манере, подходящей для потоковой работы с файлами. Это элементарный поток; он может записывать или читать только одиночный байт или массив байтов. Однако напрямую взаимодействовать с членами типа `FileStream` вам придется нечасто. Взамен, скорее всего, вы будете применять разнообразные *оболочки потоков*, которые облегчают работу с текстовыми данными или типами .NET. Тем не менее, полезно поэкспериментировать с возможностями синхронного чтения/записи типа `FileStream`.

Пусть имеется новый проект консольного приложения под названием `FileStreamApp` (и в файле кода C# импортировано пространство имен `System.IO` и `System.Text`). Цель заключается в записи простого текстового сообщения в новый файл по имени `myMessage.dat`. Однако с учетом того, что `FileStream` может оперировать только с низкоуровневыми байтами, объект типа `System.String` придется закодировать в соответствующий байтовый массив. К счастью, в пространстве имен `System.Text` определен тип `Encoding`, предоставляющий члены, которые кодируют и декодируют строки в массивы байтов (подробное описание типа `Encoding` ищите в документации .NET Framework 4.6 SDK).

После кодирования байтовый массив сохраняется в файле с помощью метода `FileStream.Write()`. Чтобы прочитать байты обратно в память, понадобится сбросить внутреннюю позицию потока (посредством свойства `Position`) и вызвать метод `ReadByte()`. Наконец, на консоль выводится содержимое низкоуровневого байтового массива и декодированная строка. Ниже приведен полный код метода `Main()`.

```
// Не забудьте импортировать пространства имен System.Text и System.IO.
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with FileStreams *****\n");

  // Получить объект FileStream.
  using(FileStream fStream = File.Open(@"C:\myMessage.dat", FileMode.Create))
  {
    // Закодировать строку в массива байтов.
    string msg = "Hello!";
```

```
    byte[] msgAsByteArray = Encoding.Default.GetBytes(msg);
    // Записать byte[] в файл.
    fStream.Write(msgAsByteArray, 0, msgAsByteArray.Length);
    // Сбросить внутреннюю позицию потока.
    fStream.Position = 0;
    // Прочитать byte[] из файла и вывести на консоль.
    Console.Write("Your message as an array of bytes: ");
    byte[] bytesFromFile = new byte[msgAsByteArray.Length];
    for (int i = 0; i < msgAsByteArray.Length; i++)
    {
      bytesFromFile[i] = (byte)fStream.ReadByte();
      Console.Write(bytesFromFile[i]);
    }
    // Вывести декодированное сообщение.
    Console.Write("\nDecoded Message: ");
    Console.WriteLine(Encoding.Default.GetString(bytesFromFile));
  }
  Console.ReadLine();
}
```

В этом примере не только производится наполнение файла данными, но также демонстрируется основной недостаток прямой работы с типом `FileStream`: необходимость оперирования низкоуровневыми байтами. Другие производные от `Stream` типы работают в похожей манере. Например, чтобы записать последовательность байтов в область памяти, понадобится создать объект `MemoryStream`. Аналогично для передачи массива байтов через сетевое подключение используется класс `NetworkStream` (из пространства имен `System.Net.Sockets`).

Как упоминалось ранее, в пространстве имен `System.IO` доступно несколько типов для средств чтения и записи, которые инкапсулируют детали работы с типами, производными от `Stream`.

Исходный код. Проект `FileStreamApp` доступен в подкаталоге `Chapter_20`.

Работа с классами `StreamWriter` и `StreamReader`

Классы `StreamWriter` и `StreamReader` удобны всякий раз, когда нужно читать или записывать символьные данные (например, строки). Оба типа по умолчанию работают с символами Unicode; тем не менее, это можно изменить за счет предоставления должным образом сконфигурированной ссылки на объект `System.Text.Encoding`. Чтобы не усложнять пример, предположим, что стандартная кодировка Unicode вполне устраивает.

Класс `StreamReader` является производным от абстрактного класса по имени `TextReader`, как и связанный с ним тип `StringReader` (обсуждается далее в главе). Базовый класс `TextReader` предоставляет каждому из своих наследников ограниченный набор функциональности, в частности — возможность читать и "заглядывать" в символьный поток.

Класс `StreamWriter` (а также `StringWriter`, который будет рассматриваться позже) порожден от абстрактного базового класса по имени `TextWriter`. В этом классе определены члены, которые позволяют производным типам записывать текстовые данные в текущий символьный поток.

Чтобы содействовать пониманию основных возможностей записи в классах StreamWriter и StringWriter, в табл. 20.8 описаны основные члены абстрактного базового класса TextWriter.

Таблица 20.8. Основные члены TextWriter

Член	Описание
Close()	Этот метод закрывает средство записи и освобождает все связанные с ним ресурсы. В процессе буфер автоматически сбрасывается (и снова этот член функционально эквивалентен методу Dispose())
Flush()	Этот метод очищает все буферы текущего средства записи и записывает все буферизованные данные на лежащее в основе устройство; однако он не закрывает средство записи
NewLine	Это свойство задает константу новой строки для унаследованного класса средства записи. По умолчанию ограничителем строки в Windows является возврат каретки, за которым следует перевод строки (\r\n)
Write()	Этот перегруженный метод записывает данные в текстовый поток без добавления константы новой строки
WriteLine()	Этот перегруженный метод записывает данные в текстовый поток с добавлением константы новой строки

На заметку! Последние два члена класса TextWriter, вероятно, покажутся знакомыми. Вспомните, что тип System.Console имеет члены Write() и WriteLine(), которые выталкивают текстовые данные на стандартное устройство вывода. В действительности свойство Console.In является оболочкой для объекта TextWriter, а Console.Out — для TextWriter.

Производный класс StreamWriter предоставляет подходящую реализацию методов Write(), Close() и Flush(), а также определяет дополнительное свойство AutoFlush. Установка этого свойства в true заставляет StreamWriter выталкивать данные при каждой операции записи. Имейте в виду, что за счет установки AutoFlush в false можно достичь более высокой производительности, но при этом по завершении работы с объектом StreamWriter должен быть вызван метод Close().

Запись в текстовый файл

Чтобы увидеть класс StreamWriter в действии, создадим новый проект консольного приложения по имени StreamWriterReaderApp и импортируем пространство имен System.IO. В показанном ниже методе Main() с помощью метода File.CreateText() создается новый файл reminders.txt внутри текущего каталога выполнения. С применением полученного объекта StreamWriter в новый файл будут добавляться текстовые данные.

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");

  // Получить объект StreamWriter и записать строковые данные.
  using(StreamWriter writer = File.CreateText("reminders.txt"))
  {
    writer.WriteLine("Don't forget Mother's Day this year...");
    writer.WriteLine("Don't forget Father's Day this year...");
    writer.WriteLine("Don't forget these numbers:");
```

```
    for(int i = 0; i < 10; i++)
      writer.Write(i + " ");
    // Вставить новую строку.
    writer.Write(writer.NewLine);
  }
  Console.WriteLine("Created file and wrote some thoughts...");
  Console.ReadLine();
}
```

После выполнения программы можно просмотреть содержимое созданного файла (рис. 20.3). Этот файл находится в папке `bin\Debug` текущего приложения, т.к. при вызове `CreateText()` абсолютный путь не указывался.

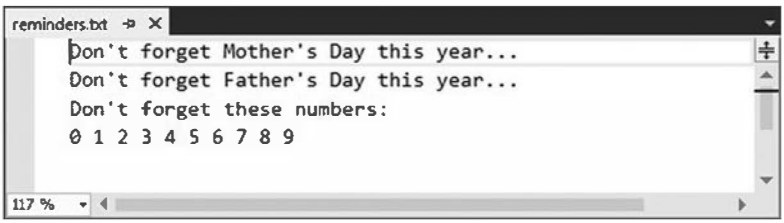


Рис. 20.3. Содержимое созданного текстового файла

Чтение из текстового файла

Далее вы научитесь программно читать данные из файла, используя соответствующий тип `StreamReader`. Вспомните, что `StreamReader` является производным от абстрактного класса `TextReader`, который предлагает функциональность, описанную в табл. 20.9.

Таблица 20.9. Основные члены `TextReader`

Член	Описание
`Peek()`	Возвращает следующий доступный символ (выраженный в виде целого числа), не изменяя текущей позиции средства чтения. Значение `-1` указывает на достижение конца потока
`Read()`	Читает данные из входного потока
`ReadBlock()`	Читает указанное максимальное количество символов из текущего потока и записывает данные в буфер, начиная с заданного индекса
`ReadLine()`	Читает строку символов из текущего потока и возвращает данные в виде строки (строка `null` указывает на признак конца файла)
`ReadToEnd()`	Читает все символы от текущей позиции до конца потока и возвращает их в виде единственной строки

Расширим текущий пример приложения с целью применения класса `StreamReader`, чтобы в нем можно было читать текстовые данные из файла `reminders.txt`:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
  ...
  // Прочитать данные из файла.
  Console.WriteLine("Here are your thoughts:\n");
```

```
  using(StreamReader sr = File.OpenText("reminders.txt"))
  {
    string input = null;
    while ((input = sr.ReadLine()) != null)
    {
      Console.WriteLine (input);
    }
  }
  Console.ReadLine();
}
```

После запуска этой программы в окне консоли отобразятся символьные данные из файла `reminders.txt`.

Прямое создание объектов типа `StreamWriter/StreamReader`

Один из запутывающих аспектов работы с типами пространства имен `System.IO` связан с тем, что идентичных результатов часто можно добиться с использованием разных подходов. Например, ранее вы уже видели, что метод `CreateText()` позволяет получить объект `StreamWriter` с типом `File` или `FileInfo`. Вообще говоря, есть еще один способ работы с объектами `StreamWriter` и `StreamReader`: создание их напрямую. Скажем, текущее приложение можно было бы переделать следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
  // Получить объект StreamWriter и записать строковые данные.
  using(StreamWriter writer = new StreamWriter("reminders.txt"))
  {
    ...
  }
  // Прочитать данные из файла.
  using(StreamReader sr = new StreamReader("reminders.txt"))
  {
    ...
  }
}
```

Несмотря на то что существование такого количества на первый взгляд одинаковых подходов к файловому вводу-выводу может сбивать с толку, имейте в виду, что конечным результатом является высокая гибкость. Теперь, когда вам известно, как перемещать символьные данные в файл и из файла с применением классов `StreamWriter` и `StreamReader`, давайте займемся исследованием роли классов `StringWriter` и `StringReader`.

Исходный код. Проект `StreamWriterReaderApp` доступен в подкаталоге `Chapter_20`.

Работа с классами `StringWriter` и `StringReader`

Классы `StringWriter` и `StringReader` можно использовать для трактовки текстовой информации как потока символов в памяти. Это определенно может быть полезно, когда нужно добавить символьную информацию к лежащему в основе буферу. Для иллюс-

трации в следующем проекте консольного приложения (StringReaderWriterApp) блок строковых данных записывается в объект StringWriter вместо файла на локальном жестком диске (не забудьте импортировать пространство имен System.IO):

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StringWriter / StringReader *****\n");

  // Создать объект StringWriter и записать символьные данные в память.
  using(StringWriter strWriter = new StringWriter())
  {
    strWriter.WriteLine("Don't forget Mother's Day this year...");

    // Получить копию содержимого (хранящегося в строке) и вывести на консоль.
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
  }
  Console.ReadLine();
}
```

Классы StringWriter и StreamWriter порождены от одного и того же базового класса (TextWriter), поэтому логика записи более или менее похожа. Тем не менее, с учетом природы StringWriter вы должны также знать, что этот класс позволяет применять метод GetStringBuilder() для извлечения объекта System.Text.StringBuilder:

```
using (StringWriter strWriter = new StringWriter())
{
  strWriter.WriteLine("Don't forget Mother's Day this year...");
  Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);

  // Получить внутренний объект StringBuilder.
  StringBuilder sb = strWriter.GetStringBuilder();
  sb.Insert(0, "Hey!! ");
  Console.WriteLine("-> {0}", sb.ToString());
  sb.Remove(0, "Hey!! ".Length);
  Console.WriteLine("-> {0}", sb.ToString());
}
```

Когда необходимо прочитать из потока строковые данные, можно использовать соответствующий тип StringReader, который (вполне ожидаемо) функционирует идентично StreamReader. На самом деле класс StringReader всего лишь переопределяет унаследованные члены, чтобы выполнять чтение из блока символьных данных, а не из файла:

```
using (StringWriter strWriter = new StringWriter())
{
  strWriter.WriteLine("Don't forget Mother's Day this year...");
  Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);

  // Читать данные из объекта StringWriter.
  using (StringReader strReader = new StringReader(strWriter.ToString()))
  {
    string input = null;
    while ((input = strReader.ReadLine()) != null)
    {
      Console.WriteLine(input);
    }
  }
}
```

Исходный код. Проект StringReaderWriterApp доступен в подкаталоге Chapter_20.

Работа с классами BinaryWriter и BinaryReader

Последним набором классов средств чтения и записи, которые рассматриваются в настоящем разделе, являются BinaryWriter и BinaryReader; оба они напрямую порождены от System.Object. Эти типы позволяют читать и записывать в поток дискретные типы данных в компактном двоичном формате. В классе BinaryWriter определен многократно перегруженный метод Write(), предназначенный для помещения некоторого типа данных в поток. Помимо Write() класс BinaryWriter предоставляет дополнительные члены, которые позволяют получать или устанавливать объекты производных от Stream типов; кроме того, класс BinaryWriter также предлагает поддержку произвольного доступа к данным (табл. 20.10).

Таблица 20.10. Основные члены BinaryWriter

Член	Описание
BaseStream	Это свойство только для чтения обеспечивает доступ к лежащему в основе потоку, используемому с объектом BinaryWriter
Close()	Этот метод закрывает двоичный поток
Flush()	Этот метод выталкивает буфер двоичного потока
Seek()	Этот метод устанавливает позицию в текущем потоке
Write()	Этот метод записывает значение в текущий поток

Класс BinaryReader дополняет функциональность класса BinaryWriter членами, описанными в табл. 20.11.

Таблица 20.11. Основные члены BinaryReader

Член	Описание
BaseStream	Это свойство только для чтения обеспечивает доступ к лежащему в основе потоку, используемому с объектом BinaryReader
Close()	Этот метод закрывает двоичный поток
PeekChar()	Этот метод возвращает следующий доступный символ без перемещения текущей позиции потока
Read()	Этот метод читает заданный набор байтов или символов и сохраняет их во входном массиве
ReadXXXX()	В классе BinaryReader определены многочисленные методы чтения, которые извлекают из потока объекты различных типов (ReadBoolean(), ReadByte(), ReadInt32() и т.д.)

В показанном далее примере (проект консольного приложения по имени BinaryWriterReader) в файл *.dat записываются данные нескольких типов:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Binary Writers / Readers *****\n");
  // Открыть средство двоичной записи в файл.
  FileInfo f = new FileInfo("BinFile.dat");
```

```
  using(BinaryWriter bw = new BinaryWriter(f.OpenWrite()))
  {
    // Вывести на консоль тип BaseStream
    // (System.IO.FileStream в этом случае).
    Console.WriteLine("Base stream is: {0}", bw.BaseStream);

    // Создать некоторые данные для сохранения в файле.
    double aDouble = 1234.67;
    int anInt = 34567;
    string aString = "A, B, C";

    // Записать данные.
    bw.Write(aDouble);
    bw.Write(anInt);
    bw.Write(aString);
  }
  Console.WriteLine("Done!");
  Console.ReadLine();
}
```

Обратите внимание, что объект `FileStream`, возвращенный методом `FileInfo.OpenWrite()`, передается конструктору типа `BinaryWriter`. Применение такого приема упрощает организацию потока *по уровням* перед записью данных. Конструктор класса `BinaryWriter` принимает любой тип, производный от `Stream` (например, `FileStream`, `MemoryStream` или `BufferedStream`). Таким образом, запись двоичных данных в память сводится просто к использованию допустимого объекта `MemoryStream`.

Для чтения данных из файла `BinFile.dat` в классе `BinaryReader` предлагается несколько способов. Ниже для извлечения каждой порции данных из файлового потока вызываются разнообразные члены, связанные с чтением:

```
static void Main(string[] args)
{
  ...
  FileInfo f = new FileInfo("BinFile.dat");
  ...
  // Читать двоичные данные из потока.
  using(BinaryReader br = new BinaryReader(f.OpenRead()))
  {
    Console.WriteLine(br.ReadDouble());
    Console.WriteLine(br.ReadInt32());
    Console.WriteLine(br.ReadString());
  }
  Console.ReadLine();
}
```

Исходный код. Проект `BinaryWriterReader` доступен в подкаталоге `Chapter_20`.

Программное слежение за файлами

Теперь, когда вы знаете, как применять различные средства чтения и записи, займемся исследованием роли класса `FileSystemWatcher`. Этот тип полезен, когда требуется программно отслеживать состояние файлов в системе. В частности, с помощью `FileSystemWatcher` можно организовать мониторинг файлов на предмет любых действий, указываемых значениями перечисления `System.IO.NotifyFilters` (более подробные сведения об этом перечислении приведены в документации .NET Framework 4.6 SDK):

```
public enum NotifyFilters
{
  Attributes, CreationTime,
  DirectoryName, FileName,
  LastAccess, LastWrite,
  Security, Size
}
```

Чтобы начать работу с типом FileSystemWatcher, в свойстве Path понадобится указать имя (и местоположение) каталога, содержащего файлы, которые нужно отслеживать, а в свойстве Filter — расширения отслеживаемых файлов.

В этот момент можно выбрать обработку событий Changed, Created и Deleted, которые функционируют в сочетании с делегатом FileSystemEventHandler. Данный делегат может вызывать любой метод, соответствующий следующей сигнатуре:

```
// Делегат FileSystemEventHandler должен указывать
// на методы, соответствующие следующей сигнатуре.
void MyNotificationHandler(object source, FileSystemEventArgs e)
```

Событие Renamed может быть также обработано с использованием делегата RenamedEventHandler, который позволяет вызывать методы с такой сигнатурой:

```
// Делегат RenamedEventHandler должен указывать
// на методы, соответствующие следующей сигнатуре.
void MyRenamedHandler(object source, RenamedEventArgs e)
```

В то время как для обработки каждого события можно применять традиционный синтаксис делегатов/событий, вы определенно будете использовать синтаксис лямбда-выражений (как это делается в загружаемом коде этого проекта).

Давайте взглянем на процесс слежения за файлом. Предположим, что на диске C: создан новый каталог по имени MyFolder, который содержит разнообразные файлы *.txt (с произвольными именами). Показанный ниже проект консольного приложения (MyDirectoryWatcher) наблюдает за файлами *.txt в каталоге MyFolder и выводит на консоль сообщения, когда происходит их создание, удаление, модификация и переименование:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Amazing File Watcher App *****\n");

  // Установить путь к каталогу, за которым нужно наблюдать.
  FileSystemWatcher watcher = new FileSystemWatcher();
  try
  {
    watcher.Path = @"C:\MyFolder";
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }

  // Указать цели наблюдения.
  watcher.NotifyFilter = NotifyFilters.LastAccess
    | NotifyFilters.LastWrite
    | NotifyFilters.FileName
    | NotifyFilters.DirectoryName;

  // Следить только за текстовыми файлами.
  watcher.Filter = "*.txt";
```

```
    // Добавить обработчики событий.
    watcher.Changed += new FileSystemEventHandler(OnChanged);
    watcher.Created += new FileSystemEventHandler(OnChanged);
    watcher.Deleted += new FileSystemEventHandler(OnChanged);
    watcher.Renamed += new RenamedEventHandler(OnRenamed);

    // Начать наблюдение за каталогом.
    watcher.EnableRaisingEvents = true;

    // Ожидать команду пользователя на завершение программы.
    Console.WriteLine(@"Press 'q' to quit app.");
    while(Console.Read()!='q')
    ;
  }
```

Следующие два обработчика событий просто выводят сообщения о модификации текущего файла:

```
  static void OnChanged(object source, FileSystemEventArgs e)
  {
    // Сообщить о действии изменения, создания или удаления файла.
    Console.WriteLine("File: {0} {1}!", e.FullPath, e.ChangeType);
  }

  static void OnRenamed(object source, RenamedEventArgs e)
  {
    // Сообщить о действии переименования файла.
    Console.WriteLine("File: {0} renamed to {1}", e.OldFullPath, e.FullPath);
  }
```

Запустите программу и откройте проводник Windows. Попробуйте переименовать файлы, создать файл `*.txt`, удалить файл `*.txt` и т.д. Вы увидите различные сообщения о состоянии текстовых файлов внутри `MyFolder`, например:

```
***** The Amazing File Watcher App *****

Press 'q' to quit app.
File: C:\MyFolder\New Text Document.txt Created!
File: C:\MyFolder\New Text Document.txt renamed to C:\MyFolder\Hello.txt
File: C:\MyFolder\Hello.txt Changed!
File: C:\MyFolder\Hello.txt Changed!
File: C:\MyFolder\Hello.txt Deleted!
```

Исходный код. Проект `MyDirectoryWatcher` доступен в подкаталоге `Chapter_20`.

Итак, знакомство с фундаментальными операциями ввода-вывода, предлагаемыми платформой .NET, завершено. Вы наверняка будете применять все эти приемы во многих приложениях. Кроме того, службы *сериализации объектов* помогают значительно упростить задачу сохранения больших объемов данных.

Понятие сериализации объектов

Термин *сериализация* описывает процесс сохранения (и, возможно, передачи) состояния объекта в потоке (например, файловом потоке или потоке в памяти). Сохраненная последовательность данных содержит всю информацию, необходимую для воссоздания (или *десериализации*) состояния объекта с целью последующего использования. Применение этой технологии делает тривиальным сохранение крупных объемов данных (в разнообразных форматах). Во многих случаях сохранение данных приложения с

использованием служб сериализации дает в результате меньше кода, чем с применением средств чтения/записи из пространства имен System.IO.

Например, пусть требуется создать настольное приложение с графическим пользовательским интерфейсом, которое должно предоставлять конечным пользователям возможность сохранения их предпочтений (цвета окон, размер шрифта и т.д.). Для этого можно определить класс по имени UserPrefs и инкапсулировать в нем около двадцати полей данных. В случае использования типа System.IO.BinaryWriter пришлось бы *вручную* сохранять каждое поле объекта UserPrefs. Подобным же образом при загрузке данных из файла обратно в память понадобилось бы применять класс System.IO.BinaryReader и снова *вручную* читать каждое значение, чтобы повторно сконфигурировать новый объект UserPrefs.

Хотя поступать так вполне допустимо, можно сэкономить значительное время, пометив класс UserPrefs атрибутом [Serializable]:

```
[Serializable]
public class UserPrefs
{
  public string WindowColor;
  public int FontSize;
}
```

В итоге полное состояние объекта может быть сохранено с помощью лишь нескольких строк кода. Не вдаваясь пока в детали, взгляните на показанный ниже метод Main():

```
static void Main(string[] args)
{
  UserPrefs userData= new UserPrefs();
  userData.WindowColor = "Yellow";
  userData.FontSize = 50;

  // BinaryFormatter сохраняет данные в двоичном формате.
  // Чтобы получить доступ к BinaryFormatter, необходимо импортировать
  // пространство имен System.Runtime.Serialization.Formatters.Binary.
  BinaryFormatter binFormat = new BinaryFormatter();

  // Сохранить объект в локальном файле.
  using(Stream fStream = new FileStream("user.dat",
    FileMode.Create, FileAccess.Write, FileShare.None))
  {
    binFormat.Serialize(fStream, userData);
  }
  Console.ReadLine();
}
```

Сериализация объектов .NET упрощает сохранение объектов, но ее внутренний процесс довольно сложен. Например, когда объект сохраняется в потоке, все ассоциированные с ним данные (т.е. данные базового класса и содержащиеся в нем объекты) также автоматически сериализируются. Следовательно, при попытке сериализации производного класса в игру вступают также все данные по цепочке наследования. Вы увидите, что для представления множества взаимосвязанных объектов используется граф объектов.

Службы сериализации .NET также позволяют сохранять граф объектов в разных форматах. В предыдущем примере кода применялся тип BinaryFormatter, поэтому состояние объекта UserPrefs сохранялось в компактном двоичном формате. Граф объектов можно также сохранить в формате SOAP или XML, используя другие типы форматеров. Эти форматы могут быть очень полезными, когда необходимо гарантировать

возможность передачи сохраненных объектов между операционными системами, языками и архитектурами.

На заметку! Инфраструктура WCF предлагает слегка отличающийся механизм для сериализации объектов в и из операций служб WCF; он предусматривает применение атрибутов `[DataContract]` и `[DataMember]`. Подробнее об этом речь пойдет в главе 25.

И, наконец, имейте в виду, что граф объектов может быть сохранен в *любом* типе, производном от `System.IO.Stream`. В предыдущем примере для сохранения объекта `UserPrefs` в локальном файле использовался тип `FileStream`. Однако если необходимо сохранить объект в заданной области памяти, то взамен можно применить тип `MemoryStream`. Главное, чтобы последовательность данных корректно представляла состояние объектов внутри графа.

Роль графов объектов

Как упоминалось ранее, среда CLR будет учитывать все связанные объекты, чтобы обеспечить корректное сохранение данных, когда объект сериализируется. Такой набор связанных объектов называется *графом объектов*. Графы объектов предоставляют простой способ документирования взаимосвязи между множеством элементов. Имейте в виду, что графы объектов *не* обозначают отношения "является" и "имеет" объектно-ориентированного программирования. Вместо этого стрелки в графе объектов можно трактовать как "требует" или "зависит от".

Каждый объект в графе получает уникальное числовое значение. Имейте в виду, что числа, назначенные объектам в графе, являются произвольными и не имеют никакого смысла для внешнего мира.

После того как всем объектам назначены числовые значения, граф объектов может записывать набор зависимостей для каждого объекта.

Для примера предположим, что создано множество классов, которые моделируют автомобили. Существует базовый класс по имени `Car`, который "имеет" класс `Radio`. Другой класс по имени `JamesBondCar` расширяет базовый тип `Car`. На рис. 20.4 показан возможный граф объектов, моделирующий такие отношения.

При чтении графов объектов для описания соединяющих стрелок можно использовать выражение "зависит от" или "ссылается на". Таким образом, на рис. 20.4 видно, что класс `Car` ссылается на класс `Radio` (учитывая отношение "имеет"), `JamesBondCar` ссылается на `Car` (из-за отношения "является"), а также на `Radio` (поскольку наследует эту защищенную переменную-член).

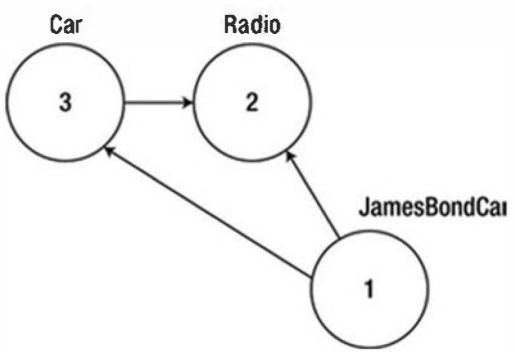


Рис. 20.4. Простой граф объектов

Разумеется, среда CLR не рисует картинки в памяти для представления графа связанных объектов. Взамен отношение, показанное на рис. 20.4, представляется математической формулой, которая выглядит следующим образом:

```
[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]
```

Проанализировав формулу, вы заметите, что объект 3 (`Car`) имеет зависимость от объекта 2 (`Radio`). Объект 2 (`Radio`) — это "одинокий волк", которому никто не нужен. Наконец, объект 1 (`JamesBondCar`) имеет зависимость от объекта 3, а также от объекта 2. В любом случае при сериализации или десериализации экземпляра `JamesBondCar` граф объектов гарантирует, что типы `Radio` и `Car` также примут участие в процессе.

Привлекательность процесса сериализации заключается в том, что граф, представляющий отношения между объектами, устанавливается автоматически "за кулисами". Как будет показано позже в главе, при желании можно вмешиваться в конструирование графа объектов, настраивая процесс сериализации с применением атрибутов и интерфейсов.

На заметку! Строго говоря, тип `XmlSerializer` (описанный далее в главе) не сохраняет состояние с использованием графа объектов; тем не менее, этот тип сериализирует и десериализирует связанные объекты в предсказуемой манере.

Конфигурирование объектов для сериализации

Чтобы сделать объект доступным службам сериализации .NET, понадобится только декорировать каждый связанный класс (или структуру) атрибутом `[Serializable]`. Если выясняется, что какой-то тип имеет члены-данные, которые не должны (или не могут) участвовать в сериализации, их необходимо пометить атрибутом `[NonSerialized]`. Это помогает сократить размер хранимых данных, когда в сериализируемом классе есть переменные-члены, которые запоминать не нужно (к примеру, фиксированные значения, случайные значения и кратковременные данные).

Определение сериализируемых типов

Для начала создадим новый проект консольного приложения под названием `SimpleSerialize`. Добавим в него новый класс по имени `Radio`, помеченный атрибутом `[Serializable]`, в котором исключается одна переменная-член (`radioID`), помеченная атрибутом `[NonSerialized]` и потому не сохраняемая в указанном потоке данных:

```
[Serializable]
public class Radio
{
  public bool hasTweeters;
  public bool hasSubWoofers;
  public double[] stationPresets;

  [NonSerialized]
  public string radioID = "XF-552RR6";
}
```

Затем добавим два дополнительных типа для представления классов `JamesBondCar` и `Car`, которые также помечены атрибутом `[Serializable]`, и определим в них следующие поля данных:

```
[Serializable]
public class Car
```

```
{
  public Radio theRadio = new Radio();
  public bool isHatchBack;
}

[Serializable]
public class JamesBondCar : Car
{
  public bool canFly;
  public bool canSubmerge;
}
```

Имейте в виду, что атрибут `[Serializable]` не наследуется от родительского класса. Таким образом, если вы порождаете класс от типа, помеченного `[Serializable]`, то дочерний класс также должен быть снабжен атрибутом `[Serializable]`, иначе он не сможет сохраняться в потоке. В действительности все объекты внутри графа объектов должны быть помечены атрибутом `[Serializable]`. Попытка сериализировать несериализируемый объект с применением класса `BinaryFormatter` или `SoapFormatter` приведет к генерации исключения `SerializationException` во время выполнения.

Открытые поля, закрытые поля и открытые свойства

Обратите внимание, что в каждом из этих классов поля данных определены как открытые, что способствует упрощению примера. Несомненно, с точки зрения объектно-ориентированного программирования закрытые данные с доступом через открытые свойства были бы предпочтительнее. Кроме того, ради простоты в типах не определены какие-либо специальные конструкторы; следовательно, все неинициализированные поля данных получат ожидаемые стандартные значения.

Оставив в стороне принципы объектно-ориентированного программирования, вас может интересовать вопрос: какое определение полей данных внутри типа ожидают форматеры, чтобы сериализировать их в поток? Ответ: в зависимости от обстоятельств. Если вы сохраняете состояние объекта с использованием типа `BinaryFormatter` или `SoapFormatter`, то это совершенно безразлично. Указанные типы запрограммированы на сериализацию *всех* сериализируемых полей типа независимо от того, являются они открытыми полями, закрытыми полями или закрытыми полями, доступными через открытые свойства. Однако вспомните, что при наличии элементов данных, которые не должны сохраняться в графе объектов, можно выборочно пометить открытые или закрытые поля атрибутом `[NonSerialized]`, как было сделано со строковым полем в типе `Radio`.

Тем не менее, в случае применения тип `XmlSerializer` ситуация совсем другая. Этот тип будет сериализировать *только* открытые поля данных или закрытые данные, доступные посредством открытых свойств. Закрытые данные, которые не доступны через свойства, будут игнорироваться. Например, рассмотрим следующий сериализируемый тип `Person`:

```
[Serializable]
public class Person
{
  // Открытое поле.
  public bool isAlive = true;

  // Закрытое поле.
  private int personAge = 21;

  // Открытое свойство/закрытые данные.
  private string fName = string.Empty;
```

```
  public string FirstName
  {
    get { return fName; }
    set { fName = value; }
  }
}
```

При обработке этого типа с помощью BinaryFormatter или SoapFormatter обнаружится, что поля isAlive, personAge и fName сохраняются в выбранном потоке. Однако тип XmlSerializer *не* сохранит значение personAge, потому что эта часть закрытых данных не инкапсулирована в открытом свойстве. Чтобы сохранять personAge с помощью XmlSerializer, это поле понадобится определить как открытое либо инкапсулировать закрытый член в открытом свойстве.

Выбор форматера сериализации

После конфигурирования типов для участия в схеме сериализации .NET путем применения необходимых атрибутов потребуется выбрать формат (двоичный, SOAP или XML), в котором будет храниться состояние объектов. Перечисленные возможности представлены следующими классами:

- BinaryFormatter
- SoapFormatter
- XmlSerializer

Тип BinaryFormatter сериализирует состояние объекта в поток, используя компактный двоичный формат. Этот тип определен внутри пространства имен System.Runtime.Serialization.Formatters.Binary, которое входит в сборку mscorlib.dll. Для получения доступа к этому типу понадобится указать следующую директиву using:

```
// Получить доступ к BinaryFormatter из сборки mscorlib.dll.
using System.Runtime.Serialization.Formatters.Binary;
```

Тип SoapFormatter сохраняет состояние объекта в виде сообщения SOAP (стандартный XML-формат для передачи и приема сообщений от веб-служб, основанных на SOAP). Этот тип определен в пространстве имен System.Runtime.Serialization.Formatters.Soap, находящемся в *отдельной сборке*. Таким образом, для форматирования графа объектов в сообщение SOAP необходимо сначала установить ссылку на System.Runtime.Serialization.Formatters.Soap.dll с применением диалогового окна Add Reference (Добавить ссылку) в Visual Studio и затем указать директиву using:

```
// Требуется ссылка на сборку System.Runtime.Serialization.Formatters.Soap.dll.
using System.Runtime.Serialization.Formatters.Soap;
```

И, наконец, для сохранения дерева объектов в документе XML предусмотрен тип XmlSerializer. Чтобы его использовать, нужно указать директиву using для пространства имен System.Xml.Serialization и установить ссылку на сборку System.Xml.dll. К счастью, шаблоны проектов Visual Studio автоматически ссылаются на System.Xml.dll, так что достаточно просто поступить так:

```
// Определено внутри сборки System.Xml.dll.
using System.Xml.Serialization;
```

Интерфейсы IFormatter и IRemotingFormatter

Независимо от того, какой форматер выбран, имейте в виду, что все они унаследованы непосредственно от System.Object, поэтому *не* разделяют общий набор чле-

нов от какого-то базового класса сериализации. Тем не менее, типы `BinaryFormatter` и `SoapFormatter` поддерживают общие члены за счет реализации интерфейсов `IFormatter` и `IRemotingFormatter` (как ни странно, тип `XmlSerializer` не реализует ни одного из них).

Интерфейс `System.Runtime.Serialization.IFormatter` определяет основные методы `Serialize()` и `Deserialize()`, которые делают всю черновую работу по перемещению графов объектов в специфический поток и обратно. Помимо них в `IFormatter` определено несколько свойств, которые реализующий тип применяет "за кулисами":

```
public interface IFormatter
{
  SerializationBinder Binder { get; set; }
  StreamingContext Context { get; set; }
  ISurrogateSelector SurrogateSelector { get; set; }
  object Deserialize(Stream serializationStream);
  void Serialize(Stream serializationStream, object graph);
}
```

Интерфейс `System.Runtime.Remoting.Messaging.IRemotingFormatter` (который внутренне используется уровнем удаленного взаимодействия .NET Remoting) перегружает методы `Serialize()` и `Deserialize()` в манере, больше подходящей для распределенного сохранения. Обратите внимание, что интерфейс `IRemotingFormatter` является производным от более общего интерфейса `IFormatter`:

```
public interface IRemotingFormatter : IFormatter
{
  object Deserialize(Stream serializationStream, HeaderHandler handler);
  void Serialize(Stream serializationStream, object graph, Header[] headers);
}
```

Несмотря на то что взаимодействие с этими интерфейсами в большинстве сценариев сериализации может не потребоваться, вспомните, что полиморфизм на основе интерфейсов позволяет указывать экземпляры `BinaryFormatter` или `SoapFormatter` с применением ссылки на `IFormatter`. Следовательно, если необходимо построить метод, который может сериализировать граф объектов с использованием любого из этих классов, то можно написать такой код:

```
static void SerializeObjectGraph(IFormatter itfFormat,
                                  Stream destStream, object graph)
{
  itfFormat.Serialize(destStream, graph);
}
```

Точность типов среди форматеров

Наиболее очевидное отличие между тремя форматерами касается того, каким образом граф объектов сохраняется в потоке (двоичном, SOAP или XML). Вы должны также знать о нескольких более тонких аспектах отличия, в частности, как форматеры справляются с *точностью типов*. Когда применяется тип `BinaryFormatter`, он сохраняет не только данные полей объектов из графа, но также полностью заданное имя каждого типа и полное имя определяющей его сборки (имя, версия, маркер открытого ключа и культура). Эти дополнительные элементы данных делают `BinaryFormatter` идеальным выбором, когда необходимо передавать объекты по значению (например, полные копии) между границами машин для использования в приложениях .NET.

Форматер `SoapFormatter` сохраняет следы сборки источника за счет применения пространства имен XML. Для примера вспомните тип `Person`, определенный ранее в главе. Если бы этот тип был сохранен как сообщение SOAP, то вы обнаружили бы, что

открывающий элемент Person уточнен сгенерированным параметром xlmns. Взгляните на следующее частичное определение, обратив особое внимание на пространство имен XML под названием a1:

```
<a1:Person id="ref-1" xmlns:a1=
  "http://schemas.microsoft.com/clr/nsassem/SimpleSerialize/MyApp%2C%20
  Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
  <isAlive>true</isAlive>
  <personAge>21</personAge>
  <fName id="ref-3">Mel</fName>
</a1:Person>
```

Однако XmlSerializer *не* пытается сберечь точную информацию о типе; следовательно, он не записывает полностью заданное имя типа или сборку, где он определен. На первый взгляд это может показаться ограничением, но сериализация XML используется классическими веб-службами .NET, которые могут вызываться клиентами из любой платформы. Другими словами, нет никакого смысла сериализировать полные метаданные типа .NET. Вот возможное XML-представление типа Person:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <isAlive>true</isAlive>
  <PersonAge>21</PersonAge>
  <FirstName>Frank</FirstName>
</Person>
```

Если необходимо сохранить состояние объекта так, чтобы с ним можно было работать в любой операционной системе (скажем, Windows, Mac OS X и многочисленных дистрибутивах Linux), платформе приложений (например, .NET, Java Enterprise Edition и COM) или языке программирования, то поддерживать полную точность типов не нужно, поскольку нельзя рассчитывать на то, что все возможные получатели способны воспринимать типы данных, специфичные для .NET. С учетом этого типы SoapFormatter и XmlSerializer являются идеальным выбором, когда требуется обеспечить как можно более широкое распространение сохраненного дерева объектов.

Сериализация объектов с использованием BinaryFormatter

Чтобы проиллюстрировать, насколько просто сохранять экземпляр JamesBondCar в физическом файле, можно применить тип BinaryFormatter. Двумя ключевыми методами типа BinaryFormatter, о которых следует знать, являются Serialize() и Deserialize():

1. Serialize() сохраняет граф объектов в указанном потоке как последовательность байтов;
2. Deserialize() преобразует сохраненную последовательность байтов в граф объектов.

Предположим, что после создания экземпляра JamesBondCar и модификации некоторых данных состояния требуется сохранить его в файле *.dat. Начать следует с создания самого файла *.dat. Достичь этого можно путем создания экземпляра типа System.IO.FileStream. Затем можно создать экземпляр BinaryFormatter и передать ему экземпляр FileStream и граф объектов для сохранения. Взгляните на следующий метод Main():

```
// Не забудьте импортировать пространства имен
// System.Runtime.Serialization.Formatters.Binary и System.IO.
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Serialization *****\n");
  // Создать объект JamesBondCar и установить состояние.
  JamesBondCar jbc = new JamesBondCar();
  jbc.canFly = true;
  jbc.canSubmerge = false;
  jbc.theRadio.stationPresets = new double[]{89.3, 105.1, 97.1};
  jbc.theRadio.hasTweeters = true;
  // Сохранить объект JamesBondCar в указанном файле в двоичном формате.
  SaveAsBinaryFormat(jbc, "CarData.dat");
  Console.ReadLine();
}
```

Метод `SaveAsBinaryFormat()` реализован так, как показано ниже:

```
static void SaveAsBinaryFormat(object objGraph, string fileName)
{
  // Сохранить граф объектов в файл CarData.dat в двоичном виде.
  BinaryFormatter binFormat = new BinaryFormatter();
  using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
  {
    binFormat.Serialize(fStream, objGraph);
  }
  Console.WriteLine("=> Saved car in binary format!");
}
```

Как видите, метод `BinaryFormatter.Serialize()` — это член, ответственный за построение графа объектов и передачу последовательности байтов объекту производного от `Stream` типа. В рассматриваемом случае поток представляет физический файл. Сериализировать объекты можно было бы также в любой тип, производный от `Stream`, такой как область памяти или сетевой поток. После выполнения программы можно просмотреть содержимое файла `CarData.dat`, которое представляет данный экземпляр `JamesBondCar`, перейдя в папку `bin\Debug` текущего проекта. На рис. 20.5 показано содержимое этого файла, открытого в Visual Studio.

Десериализация объектов с использованием `BinaryFormatter`

Теперь предположим, что необходимо прочитать сохраненный объект `JamesBondCar` из двоичного файла обратно в объектную переменную. После открытия файла `CataData.dat` (с помощью метода `File.OpenRead()`) можно вызвать метод `Deserialize()` класса `BinaryFormatter`. Имейте в виду, что `Deserialize()` возвращает объект общего типа `System.Object`, так что понадобится применить явное приведение:

```
static void LoadFromBinaryFile(string fileName)
{
  BinaryFormatter binFormat = new BinaryFormatter();
  // Прочитать объект JamesBondCar из двоичного файла.
  using(Stream fStream = File.OpenRead(fileName))
  {
    JamesBondCar carFromDisk =
      (JamesBondCar)binFormat.Deserialize(fStream);
    Console.WriteLine("Can this car fly? : {0}", carFromDisk.canFly);
  }
}
```

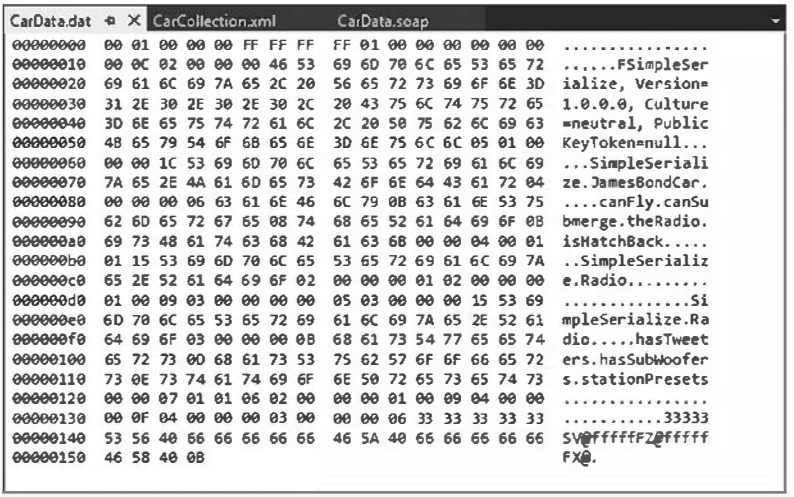


Рис. 20.5. Объект `JamesBondCar`, сериализированный с использованием `BinaryFormatter`

Обратите внимание, что при вызове методу `Deserialize()` передается объект производного от `Stream` типа, который представляет местоположение сохраненного графа объектов. После приведения возвращенного объекта к корректному типу вы обнаружите, что данные состояния восстановлены на момент, когда объект сохранялся.

Сериализация объектов с использованием SoapFormatter

Следующим рассматриваемым форматером будет `SoapFormatter`, который сериализирует данные в подходящем конверте SOAP. Выражаясь кратко, протокол SOAP (Simple Object Access Protocol — простой протокол доступа к объектам) описывает стандартный процесс вызова методов в независимой от платформы и операционной системы манере.

Предполагая, что была добавлена ссылка на сборку `System.Runtime.Serialization.Formatters.Soap.dll` (и импортировано пространство имен `System.Runtime.Serialization.Formatters.Soap`), для сохранения и извлечения объекта `JamesBondCar` как сообщения SOAP в предыдущем примере можно просто заменить все вхождения `BinaryFormatter` типом `SoapFormatter`. Взгляните на следующий новый метод класса `Program`, который сериализирует объект в локальный файл в формате SOAP:

```
// Не забудьте импортировать пространства имен
// System.Runtime.Serialization.Formatters.Soap и установить ссылку
// на сборку System.Runtime.Serialization.Formatters.Soap.dll.
static void SaveAsSoapFormat (object objGraph, string fileName)
{
  // Сохранить граф объектов в файле CarData.soap в формате SOAP.
  SoapFormatter soapFormat = new SoapFormatter();

  using(Stream fStream = new FileStream(fileName,
    FileMode.Create, FileAccess.Write, FileShare.None))
  {
    soapFormat.Serialize(fStream, objGraph);
  }
  Console.WriteLine("=> Saved car in SOAP format!");
}
```

Как и ранее, для перемещения графа объектов в поток и обратно применяются методы `Serialize()` и `Deserialize()`. После вызова метода `SaveAsSoapFormat()` внутри `Main()` и запуска приложения можно открыть результирующий файл `*.soap`. В нем находятся XML-элементы, которые описывают значения состояния текущего объекта `JamesBondCar`, а также отношения между объектами в графе, представленные с использованием лексем `#ref` (рис. 20.6).

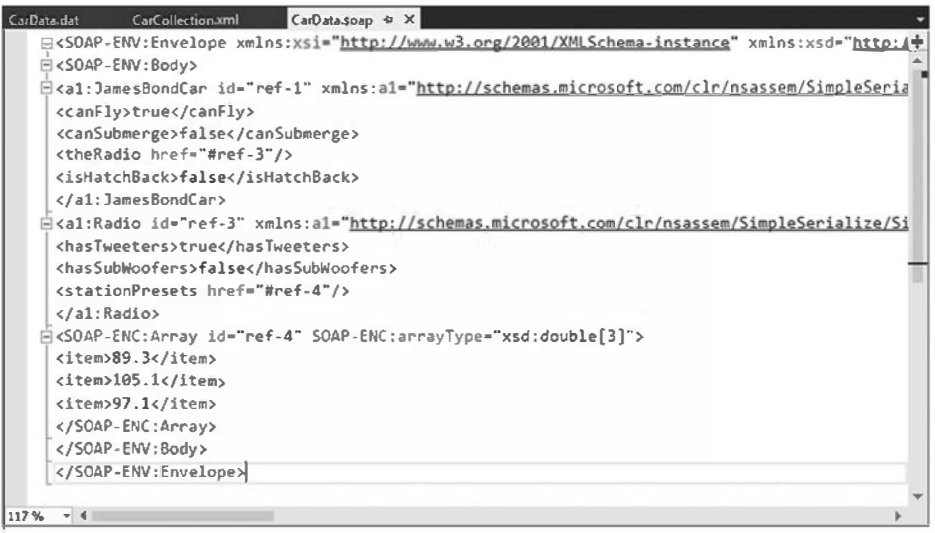


Рис. 20.6. Объект `JamesBondCar`, сериализированный с применением `SoapFormatter`

Сериализация объектов с использованием `XmlSerializer`

В дополнение к двоичному форматеру и форматеру SOAP сборка `System.Xml.dll` предлагает третий класс форматера — `System.Xml.Serialization.XmlSerializer`. Его можно применять для сохранения *открытого* состояния заданного объекта в виде чистой XML-разметки как противоположность XML-данным, помещенным внутрь сообщения SOAP. Работа с этим типом несколько отличается от работы с типами `SoapFormatter` или `BinaryFormatter`. Рассмотрим показанный ниже код, в котором предполагается, что было импортировано пространство имен `System.Xml.Serialization`:

```
static void SaveAsXmlFormat(object objGraph, string fileName)
{
  // Сохранить объект в файле CarData.xml в формате XML.
  XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondCar));
  using(Stream fStream = new FileStream(fileName,
                         FileMode.Create, FileAccess.Write, FileShare.None))
  {
    xmlFormat.Serialize(fStream, objGraph);
  }
  Console.WriteLine("=> Saved car in XML format!");
}
```

Основное отличие здесь в том, что класс `XmlSerializer` требует указания информации о типе, которая представляет класс, подлежащий сериализации. В сгенерированном XML-файле (в случае вызова метода `SaveAsXmlFormat()` внутри `Main()`) находятся следующие данные XML:

```
<?xml version="1.0"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <theRadio>
    <hasTweeters>true</hasTweeters>
    <hasSubWoofers>false</hasSubWoofers>
    <stationPresets>
      <double>89.3</double>
      <double>105.1</double>
      <double>97.1</double>
    </stationPresets>
    <radioID>XF-552RR6</radioID>
  </theRadio>
  <isHatchBack>false</isHatchBack>
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

На заметку! Класс `XmlSerializer` требует, чтобы все сериализированные типы в графе объектов поддерживали стандартный конструктор (поэтому не забудьте его добавить, если определяли специальные конструкторы). В противном случае во время выполнения будет сгенерировано исключение `InvalidOperationException`.

Управление генерацией данных XML

Если у вас есть опыт работы с технологиями XML, то вы знаете, что часто важно гарантировать соответствие данных внутри документа XML набору правил, которые устанавливают *действительность* данных. Понятие *действительного* документа XML не имеет никакого отношения к синтаксической правильности элементов XML (вроде того, что все открывающие элементы должны иметь закрывающие элементы). Действительные документы отвечают согласованным правилам форматирования (например, поле `X` должно быть выражено в виде атрибута, но не подэлемента), которые обычно задаются посредством схемы XML или файла определения типа документа (Document-Type Definition — DTD).

По умолчанию класс `XmlSerializer` сериализирует все открытые поля/свойства как элементы XML, а не как атрибуты XML. Чтобы управлять генерацией результирующего документа XML с помощью класса `XmlSerializer`, необходимо декорировать типы любым количеством дополнительных атрибутов из пространства имен `System.Xml.Serialization`. В табл. 20.12 описаны некоторые (не все) атрибуты, которые влияют на способ кодирования данных XML в потоке.

В следующем простом примере показано текущее представление данных полей объекта `JamesBondCar` в XML:

```
<?xml version="1.0" encoding="utf-8"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ...
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

Таблица 20.12. Избранные атрибуты из пространства имен `System.Xml.Serialization`

Атрибут .NET	Описание
`[XmlAttribute]`	Этот атрибут .NET можно применять к полю или свойству для сообщения `XmlSerializer` о необходимости сериализировать данные как атрибут XML (а не как подэлемент)
`[XmlElement]`	Поле или свойство будет сериализировано как элемент XML, именованный по вашему выбору
`[XmlEnum]`	Этот атрибут предоставляет имя элемента члена перечисления
`[XmlRoot]`	Этот атрибут управляет тем, как будет сконструирован корневой элемент (пространство имен и имя элемента)
`[XmlText]`	Свойство или поле будет сериализировано как текст XML (т.е. содержимое, находящееся между начальным и конечным дескрипторами корневого элемента)
`[XmlType]`	Этот атрибут предоставляет имя и пространство имен типа XML

Если необходимо указать специальное пространство имен XML, которое уточняет `JamesBondCar` и кодирует значения `canFly` и `canSubmerge` в виде атрибутов XML, то можно модифицировать определение класса `JamesBondCar`, как показано ниже:

```
[Serializable, XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
  [XmlAttribute]
  public bool canFly;
  [XmlAttribute]
  public bool canSubmerge;
}
```

Результатом будет следующий документ XML (обратите внимание на открывающий элемент `<JamesBondCar>`):

```
<?xml version="1.0"""?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  canFly="true" canSubmerge="false"
  xmlns="http://www.MyCompany.com">
  ...
</JamesBondCar>
```

Разумеется, для управления тем, как `XmlSerializer` генерирует результирующий документ XML, можно использовать многие другие атрибуты .NET. За подробной информацией обращайтесь к описанию пространства имен `System.Xml.Serialization` в документации .NET Framework 4.6 SDK.

Сериализация коллекций объектов

Теперь, когда вы видели, каким образом сохранять одиночный объект в потоке, давайте посмотрим, как сохранить набор объектов. Вы наверняка заметили, что метод `Serialize()` интерфейса `IFormatter` не предоставляет способа указания произвольного количества объектов в качестве входных данных (а только единственного объекта типа `System.Object`). Вдобавок возвращаемое значение `Deserialize()` также представляет собой одиночный объект `System.Object` (то же самое базовое ограничение касается и `XmlSerializer`):

```
public interface IFormatter
{
  ...
  object Deserialize(Stream serializationStream);
  void Serialize(Stream serializationStream, object graph);
}
```

Вспомните, что с помощью `System.Object` можно представить целое дерево объектов. Поэтому если передать объект, который помечен атрибутом `[Serializable]` и содержит в себе другие объекты `[Serializable]`, то с помощью единственного вызова данного метода сохранится весь набор объектов. К счастью, большинство типов в пространствах имен `System.Collections` и `System.Collections.Generic` уже помечено атрибутом `[Serializable]`. Следовательно, для сохранения набора объектов нужно просто добавить его в контейнер (такой как обычный массив, `ArrayList` или `List<T>`) и сериализировать этот контейнер в выбранный поток.

Предположим, что класс `JamesBondCar` дополнен конструктором, принимающим два аргумента, который позволяет устанавливать несколько фрагментов данных состояния (также должен быть добавлен стандартный конструктор, как того требует `XmlSerializer`):

```
[Serializable,
 XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
  public JamesBondCar(bool skyWorthy, bool seaWorthy)
  {
    canFly = skyWorthy;
    canSubmerge = seaWorthy;
  }
  // XmlSerializer требует стандартного конструктора!
  public JamesBondCar(){}
  ...
}
```

Теперь можно сохранять любое количество объектов `JamesBondCar`:

```
static void SaveListOfCars()
{
  // Сохранить список List<T> объектов JamesBondCar.
  List<JamesBondCar> myCars = new List<JamesBondCar>();
  myCars.Add(new JamesBondCar(true, true));
  myCars.Add(new JamesBondCar(true, false));
  myCars.Add(new JamesBondCar(false, true));
  myCars.Add(new JamesBondCar(false, false));

  using(Stream fStream = new FileStream("CarCollection.xml",
    FileMode.Create, FileAccess.Write, FileShare.None))
  {
    XmlSerializer xmlFormat = new XmlSerializer(typeof(List<JamesBondCar>));
    xmlFormat.Serialize(fStream, myCars);
  }
  Console.WriteLine("=> Saved list of cars!");
}
```

Здесь применяется класс `XmlSerializer`, так что для каждого из подобъектов внутри корневого объекта (`List<JamesBondCar>` в данном случае) должна быть указана информация о типе. Но если взамен использовать тип `BinaryFormatter` или `SoapFormatter`, то логика будет еще проще:

```
static void SaveListOfCarsAsBinary()
{
  // Сохранить объект ArrayList (myCars) в двоичном виде.
  List<JamesBondCar> myCars = new List<JamesBondCar>();

  BinaryFormatter binFormat = new BinaryFormatter();
  using(Stream fStream = new FileStream("AllMyCars.dat",
    FileMode.Create, FileAccess.Write, FileShare.None))
  {
    binFormat.Serialize(fStream, myCars);
  }
  Console.WriteLine("=> Saved list of cars in binary!");
}
```

Исходный код. Проект `SimpleSerialize` доступен в подкаталоге `Chapter_20`.

Настройка процесса сериализации SOAP и двоичной сериализации

В большинстве ситуаций стандартная схема сериализации, предоставляемая платформой .NET, будет в точности тем, что требуется. Нужно лишь применить атрибут `[Serializable]` к связанным типам и передать дерево объектов выбранному форматеру для обработки. Тем не менее, в некоторых случаях может понадобиться вмешательство в процессы конструирования дерева и сериализации. Например, пусть существует бизнес-правило, которое гласит, что все поля данных должны сохраняться в определенном формате, или же в поток необходимо добавить дополнительные данные, которые не отображаются напрямую на поля сохраняемого объекта (скажем, временные метки и уникальные идентификаторы).

В пространстве имен `System.Runtime.Serialization` предусмотрено несколько типов, которые позволяют вмешиваться в процесс сериализации объектов. В табл. 20.13 описаны основные типы, о которых следует знать.

Таблица 20.13. Основные типы пространства имен `System.Runtime.Serialization`

Тип	Описание
`ISerializable`	Этот интерфейс может быть реализован типом `[Serializable]` для управления его сериализацией и десериализацией
`ObjectIDGenerator`	Этот тип генерирует идентификаторы для членов в графе объектов
`[OnDeserialized]`	Этот атрибут позволяет указать метод, который будет вызван немедленно после десериализации объекта
`[OnDeserializing]`	Этот атрибут позволяет указать метод, который будет вызван перед началом процесса десериализации
`[OnSerialized]`	Этот атрибут позволяет указать метод, который будет вызван немедленно после того, как объект сериализирован
`[OnSerializing]`	Этот атрибут позволяет указать метод, который будет вызван перед началом процесса сериализации
`[OptionalField]`	Этот атрибут позволяет определить поле типа, которое может быть пропущено в указанном потоке
`SerializationInfo`	Этот класс является *пакетом свойств*, который поддерживает пары "имя-значение", представляющие состояние объекта во время процесса сериализации

Углубленный взгляд на сериализацию объектов

Прежде чем приступить к исследованию различных способов настройки процесса сериализации, полезно более внимательно присмотреться к тому, что происходит "за кулисами". Когда тип `BinaryFormatter` сериализирует граф объектов, он отвечает за передачу следующей информации в указанный поток:

- полностью заданное имя объекта в графе (например, `MyApp.JamesBondCar`);
- имя сборки, определяющей граф объектов (скажем, `MyApp.exe`);
- экземпляр класса `SerializationInfo`, который содержит все данные состояния, поддерживаемого членами графа объектов.

Во время процесса десериализации `BinaryFormatter` использует ту же самую информацию для построения идентичной копии объекта с применением данных, извлеченных из лежащего в основе потока. Процесс, выполняемый `SoapFormatter`, очень похож.

На заметку! Вспомните, что для обеспечения максимальной мобильности объекта форматер `XmlSerializer` не сохраняет полностью заданное имя типа или имя определяющей его сборки. Тип `XmlSerializer` позволяет сохранять только открытые данные.

Помимо перемещения необходимых данных в поток и обратно форматеры также анализируют члены графа объектов на предмет перечисленных ниже частей инфраструктуры.

- Выполняется проверка, помечен ли объект атрибутом `[Serializable]`. Если объект не помечен, генерируется исключение `SerializationException`.
- Если объект помечен атрибутом `[Serializable]`, производится проверка, реализует ли он интерфейс `ISerializable`. Если да, на этом объекте вызывается метод `GetObjectData()`.
- Если объект не реализует интерфейс `ISerializable`, используется стандартный процесс сериализации, который обрабатывает все поля, не помеченные атрибутом `[NonSerialized]`.

В дополнение к определению того, поддерживает ли тип интерфейс `ISerializable`, форматеры также отвечают за исследование типов на предмет поддержки членов, которые оснащены атрибутами `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` или `[OnDeserialized]`. Мы рассмотрим роль этих атрибутов чуть позже, а сначала выясним предназначение интерфейса `ISerializable`.

Настройка сериализации с использованием `ISerializable`

У объектов, помеченных атрибутом `[Serializable]`, имеется возможность реализации интерфейса `ISerializable`. Это позволяет вмешиваться в процесс сериализации и выполнять любое форматирование данных до или после сериализации.

Интерфейс `ISerializable` довольно прост; в нем определен единственный метод `GetObjectData()`:

```
// Когда необходимо настраивать процесс сериализации,
// следует реализовать интерфейс ISerializable.
public interface ISerializable
{
  void GetObjectData(SerializationInfo info,
    StreamingContext context);
}
```

Метод `GetObjectData()` вызывается автоматически заданным форматером в течение процесса сериализации. Реализация этого метода заполняет входной параметр типа `SerializationInfo` последовательностью пар "имя-значение", которые (обычно) отображаются на данные полей сохраняемого объекта. В классе `SerializationInfo` определены многочисленные вариации перегруженного метода `AddValue()`, а также небольшой набор свойств, которые позволяют устанавливать и получать имя, определяющую сборку и количество членов типа. Вот частичное определение `SerializationInfo`:

```
public sealed class SerializationInfo
{
  public SerializationInfo(Type type, IFormatterConverter converter);
  public string AssemblyName { get; set; }
  public string FullTypeName { get; set; }
  public int MemberCount { get; }
  public void AddValue(string name, short value);
  public void AddValue(string name, ushort value);
  public void AddValue(string name, int value);
  ...
}
```

Типы, которые реализуют интерфейс `ISerializable`, также должны определять специальный конструктор со следующей сигнатурой:

```
// Чтобы позволить исполняющей среде устанавливать состояние объекта,
// вы должны предоставить специальный конструктор с такой сигнатурой:
[Serializable]
class SomeClass : ISerializable
{
  protected SomeClass (SerializationInfo si, StreamingContext ctx) {...}
  ...
}
```

Обратите внимание, что видимость конструктора указана как `protected`. Это вполне допустимо, потому что форматер будет иметь доступ к члену независимо от его видимости. Такие специальные конструкторы обычно определяются как `protected` (или `private`), гарантируя тем самым, что небрежный пользователь объекта никогда не создаст объект с их помощью. Первым параметром конструктора является экземпляр типа `SerializationInfo` (который был показан ранее).

Второй параметр специального конструктора имеет тип `StreamingContext` и содержит информацию относительно источника данных. Наиболее информативным членом `StreamingContext` является свойство `State`, которое хранит значение из перечисления `StreamingContextStates`. Значения этого перечисления представляют базовую композицию текущего потока.

Если только вы не намерены реализовывать какие-то низкоуровневые специальные службы удаленного взаимодействия, то иметь дело с этим перечислением напрямую придется редко. Тем не менее, ниже приведены члены перечисления `StreamingContextStates` (за более подробной информацией обращайтесь в документацию .NET Framework 4.6 SDK):

```
public enum StreamingContextStates
{
  CrossProcess,
  CrossMachine,
  File,
  Persistence,
  Remoting,
  Other,
```

```
  Clone,
  CrossAppDomain,
  All
}
```

Давайте посмотрим, каким образом настраивать процесс сериализации с применением ISerializable. Предположим, что имеется новый проект консольного приложения (по имени CustomSerialization), в котором определен тип класса, содержащий два элемента данных string. Также представим, что для этих объектов string требуется обеспечить сериализацию в поток в верхнем регистре, а десериализацию — в нижнем. Чтобы удовлетворить упомянутые правила, можно реализовать интерфейс ISerializable, как показано ниже (не забудьте импортировать пространство имен System.Runtime.Serialization):

```
[Serializable]
class StringData : ISerializable
{
  private string dataItemOne = "First data block";
  private string dataItemTwo= "More data";
  public StringData(){}
  protected StringData(SerializationInfo si, StreamingContext ctx)
  {
    // Восстановить переменные-члены из потока.
    dataItemOne = si.GetString("First_Item").ToLower();
    dataItemTwo = si.GetString("dataItemTwo").ToLower();
  }
  void ISerializable.GetObjectData(SerializationInfo info, StreamingContext ctx)
  {
    // Наполнить объект SerializationInfo форматированными данными.
    info.AddValue("First_Item", dataItemOne.ToUpper());
    info.AddValue("dataItemTwo", dataItemTwo.ToUpper());
  }
}
```

Обратите внимание, что при наполнении объекта типа SerializationInfo внутри метода GetObjectData() именовать элементы данных идентично именам внутренних переменных-членов типа *не* обязательно. Очевидно, это может быть полезным, если требуется отвязать данные типа от формата хранения. Однако имейте в виду, что получать значения в специальном защищенном конструкторе понадобится с указанием тех же самых имен, которые были назначены внутри GetObjectData().

Чтобы протестировать результаты настройки, давайте сохраним экземпляр MyStringData с использованием SoapFormatter (соответствующим образом обновите ссылки на сборки и директивы using):

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom Serialization *****");
  // Вспомните, что этот тип реализует интерфейс ISerializable.
  StringData myData = new StringData();
  // Сохранить экземпляр в локальный файл в формате SOAP.
  SoapFormatter soapFormat = new SoapFormatter();
  using(Stream fStream = new FileStream("MyData.soap",
        FileMode.Create, FileAccess.Write, FileShare.None))
  {
    soapFormat.Serialize(fStream, myData);
  }
  Console.ReadLine();
}
```

Просмотрев полученный файл *.soap, вы увидите, что строковые поля действительно сохранены в верхнем регистре:

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
  <a1:StringData id="ref-1" ...>
    <First_Item id="ref-3">FIRST DATA BLOCK</First_Item>
    <dataItemTwo id="ref-4">MORE DATA</dataItemTwo>
  </a1:StringData>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Настройка сериализации с использованием атрибутов

Несмотря на то что реализация интерфейса `ISerializable` представляет собой один из способов настройки процесса сериализации, предпочтительным способом такой настройки является определение методов, которые оснащены любыми атрибутами, связанными с сериализацией: `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` или `[OnDeserialized]`. Применение этих атрибутов менее обременительно, чем реализация интерфейса `ISerializable`, т.к. не приходится вручную взаимодействовать с входным параметром `SerializationInfo`. Взамен можно модифицировать данные состояния напрямую, в то время как форматер оперирует с типом.

На заметку! Указанные атрибуты сериализации определены в пространстве имен `System.Runtime.Serialization`.

Метод, декорированный этими атрибутами, должен быть определен так, чтобы принимать параметр `StreamingContext` и ничего не возвращать (иначе сгенерируется исключение времени выполнения). Обратите внимание, что использовать каждый атрибут сериализации необязательно; можно просто вмешиваться в те стадии процесса сериализации, которые желательно перехватывать. Следующий фрагмент кода иллюстрирует это. Здесь новый тип `[Serializable]` устанавливает те же самые требования, что и `StringData`, но на этот раз применяются атрибуты `[OnSerializing]` и `[OnDeserialized]`:

```
[Serializable]
class MoreData
{
  private string dataItemOne = "First data block";
  private string dataItemTwo= "More data";

  [OnSerializing]
  private void OnSerializing(StreamingContext context)
  {
    // Вызывается в течение процесса сериализации.
    dataItemOne = dataItemOne.ToUpper();
    dataItemTwo = dataItemTwo.ToUpper();
  }

  [OnDeserialized]
```

```
  private void OnDeserialized(StreamingContext context)
  {
    // Вызывается, когда процесс десериализации завершен.
    dataItemOne = dataItemOne.ToLower();
    dataItemTwo = dataItemTwo.ToLower();
  }
}
```

Если вы сериализируете этот новый тип, то снова обнаружите, что данные сохраняются в верхнем регистре, а восстанавливаются — в нижнем.

Исходный код. Проект `CustomSerialization` доступен в подкаталоге `Chapter_20`.

Приведенным примером завершается исследование основных деталей служб сериализации объектов, в том числе разнообразных способов настройки процесса сериализации. Вы видели, что процесс сериализации и десериализации упрощает задачу сохранения крупных объемов данных и может быть менее трудоемким, чем работа с различными классами средств чтения/записи из пространства имен `System.IO`.

Резюме

Глава начиналась с демонстрации использования типов `Directory` (`DirectoryInfo`) и `File` (`FileInfo`). Вы узнали, что эти классы позволяют манипулировать физическими файлами и каталогами на жестком диске. Затем вы ознакомились с несколькими классами, производными от абстрактного класса `Stream`. Поскольку производные от `Stream` типы оперируют с низкоуровневым потоком байтов, пространство имен `System.IO` предлагает многочисленные типы средств чтения/записи (например, `StreamWriter`, `StringWriter` и `BinaryWriter`), которые упрощают процесс. Попутно вы взглянули на функциональность типа `DriveType`, научились наблюдать за файлами с применением типа `FileSystemWatcher` и выяснили, каким образом взаимодействовать с потоками в асинхронной манере.

В главе также рассматривались службы сериализации объектов. Вы видели, что платформа .NET использует граф объектов, чтобы учесть полный набор связанных объектов, которые должны сохраняться в потоке. До тех пор, пока каждый член в графе объектов помечен атрибутом `[Serializable]`, данные сохраняются в выбранном формате (двоичном или SOAP).

Вы также ознакомились с возможностями настройки готового процесса сериализации посредством двух подходов. Во-первых, вы узнали, как реализовать интерфейс `ISerializable` (и поддерживать специальный закрытый конструктор), что позволяет вмешиваться в процесс сохранения форматерами предоставленных данных. Во-вторых, вы изучили набор атрибутов .NET, которые упрощают процесс специальной сериализации. Все, что для этого нужно — применить атрибут `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` или `[OnDeserialized]` к членам, которые принимают параметр `StreamingContext`, после чего форматеры будут вызывать их соответствующим образом.

ГЛАВА 21

ADO.NET, часть I: подключенный уровень

Внутри платформы .NET определено несколько пространств имен, которые позволяют взаимодействовать с реляционными базами данных. Все вместе эти пространства имен известны как *ADO.NET*. В настоящей главе вы сначала узнаете об общей роли инфраструктуры ADO.NET, а также основных типов и пространств имен, после чего будет обсуждаться тема поставщиков данных ADO.NET. Платформа .NET поддерживает многочисленные поставщики данных (как являющиеся частью .NET Framework, так и доступные от независимых разработчиков), каждый из которых оптимизирован для взаимодействия с конкретной системой управления базами данных (СУБД), например, Microsoft SQL Server, Oracle, MySQL и т.д.

После освоения общей функциональности, предлагаемой различными поставщиками данных, мы рассмотрим шаблон фабрики поставщиков данных. Вы увидите, что с использованием типов из пространства имен `System.Data.Common` (и связанного файла `App.config`) можно строить единственную кодовую базу, которая способна динамически выбирать поставщик данных без необходимости в повторной компиляции или развертывании кодовой базы приложения.

Пожалуй, наиболее важно то, что в этой главе у вас будет шанс создать собственную сборку библиотеки доступа к данным (`AutoLotDAL.dll`), которая инкапсулирует разнообразные операции, выполняемые в специальной базе данных по имени `AutoLot`. Глава завершается исследованием темы транзакций базы данных.

На заметку! В главе 22 возможности этой библиотеки будут расширены, а в главе 23 она будет создана с нуля посредством Entity Framework (EF). Зачем создавать ее дважды? Хотя инфраструктуры объектно-реляционного отображения (object-relational mapping — ORM) вроде Entity Framework намного упрощают (и ускоряют) процесс создания кода доступа к данным, они по-прежнему применяют ADO.NET в качестве основной технологии доступа к данным. Хорошее понимание принципов работы ADO.NET жизненно важно при поиске и устранении проблем с доступом к данным, особенно когда код был создан какой-то инфраструктурой, а не написан вами самостоятельно. К тому же вы будете встречать задачи, которые решить с помощью EF не удастся (например, массовое копирование данных в SQL), и для их решения потребуется знание ADO.NET.

Высокоуровневое определение ADO.NET

Если у вас есть опыт работы с предшествующей моделью доступа к данным на основе COM от Microsoft (Active Data Objects — ADO) и вы только начинаете использовать платформу .NET, то имейте в виду, что инфраструктура ADO.NET имеет мало общего

с ADO помимо букв *A*, *D* и *O*. В то время как определенная взаимосвязь между этими двумя системами действительно существует (скажем, в обеих присутствует концепция объектов подключений и объектов команд), некоторые знакомые по ADO типы (например, `Recordset`) больше не доступны. Вдобавок вы обнаружите в ADO.NET много новых типов, которые не имеют прямых эквивалентов в классической технологии ADO (к примеру, адаптер данных).

Технология ADO.NET была построена с учетом автономного мира. До более широкого принятия инфраструктур ORM это обычно достигалось с применением объектов `DataSet`. Объекты `DataSet` представляют локальную копию любого количества связанных таблиц данных, каждая из которых содержит коллекцию строк и столбцов. За счет использования объекта `DataSet` вызывающая сборка (такая как веб-страница или настольное приложение) способна манипулировать и обновлять содержимое `DataSet`, будучи отключенной от источника данных, а затем отправлять блоки измененных данных на обработку с применением соответствующего *адаптера данных*.

На заметку! Инфраструктуры ORM не используют объекты `DataSet`, а имеют дело со списками обычных объектов C# (также называемых POCO (plain old C# object — простой старый объект C#)). В этой и следующей главах подробно описана работа ADO.NET. Рассматриваемые здесь фундаментальные принципы важны, даже если вы планируете незамедлительно перейти на какую-то инфраструктуру ORM, подобную Entity Framework. В мире .NET инфраструктуры EF, NHibernate и прочие построены поверх ADO.NET, поэтому если вам придется столкнуться с проблемой, когда код не выглядит функционирующим так, как от него ожидается, то знание работы ADO.NET будет значительным преимуществом. В последующих главах применяется библиотека доступа к данным, разработанная с использованием EF.

С программной точки зрения большая часть ADO.NET представлена основной сборкой по имени `System.Data.dll`. Внутри этого двоичного файла находится приличное количество пространств имен (рис. 21.1), многие из которых представляют типы отдельного поставщика данных ADO.NET.

На самом деле большинство шаблонов проектов Visual Studio автоматически ссылаются на эту ключевую сборку доступа к данным. Вы должны также уяснить, что кроме `System.Data.dll` есть и другие ориентированные на ADO.NET сборки, на которые необходимо вручную ссылаться в текущем проекте посредством диалогового окна Add Reference (Добавление ссылки).

Три грани ADO.NET

Библиотеки ADO.NET можно применять тремя концептуально отличающимися способами: в подключенном режиме, в автономном режиме и через инфраструктуру ORM, такую как Entity Framework. Когда используется *подключенный уровень* (тема настоящей главы), кодовая база явно подключается к лежащему в основе хранилищу данных и отключается от него. В случае применения ADO.NET в такой манере вы обычно взаимодействуете с хранилищем данных с использованием объектов подключений, объектов команд и объектов чтения данных.

Автономный уровень, рассматриваемый в главе 22, позволяет манипулировать набором объектов `DataTable` (содержащихся внутри объекта `DataSet`), который функционирует как копия внешних данных на стороне клиента. При получении объекта `DataSet` с помощью связанного объекта адаптера данных подключение открывается и закрывается автоматически. Как и можно было догадаться, такой подход помогает быстро освобождать подключения для других вызовов и имеет большое значение для увеличения масштабируемости систем.

Рис. 21.1. Основной сборкой ADO.NET является `System.Data.dll`

Получив объект `DataSet`, вызывающий код может просматривать и обрабатывать его содержимым без затрат на сетевой трафик. К тому же если вызывающему коду нужно отправить изменения в хранилище данных, то для обновления источника данных применяется адаптер данных (в сочетании с набором операторов SQL); в этот момент подключение снова открывается для внесения обновлений в базу данных и затем немедленно закрывается.

Наконец, в главе 23 вы ознакомитесь с API-интерфейсом доступа к данным, который называется *Entity Framework* (EF). Инфраструктура EF предоставляет возможность взаимодействия с реляционной базой данных посредством объектов на стороне клиента, которые скрывают от глаз низкоуровневую специфику базы данных. Кроме того, модель программирования EF позволяет взаимодействовать с реляционными СУБД с помощью строго типизированных запросов LINQ, использующих грамматику LINQ to Entities.

Поставщики данных ADO.NET

В ADO.NET нет единого набора типов, которые взаимодействуют с различными СУБД. Вместо этого ADO.NET поддерживает многочисленные *поставщики данных*, каждый из которых оптимизирован для взаимодействия со специфичной СУБД. Первое преимущество такого подхода заключается в том, что вы можете запрограммировать специализированный поставщик данных для доступа к любым уникальным средствам отдельной СУБД. Второе преимущество связано с тем, что поставщик данных может подключаться непосредственно к механизму интересующей СУБД без какого-либо промежуточного уровня отображения.

Выражаясь просто, поставщик данных — это набор типов, определенных в заданном пространстве имен, которым известно, как взаимодействовать с конкретным источником данных. Безотносительно к тому, какой поставщик данных применяется, каждый из них определяет набор классов, предоставляющих основную функциональность. В табл. 21.1 описаны некоторые распространенные основные типы, их базовые классы (определенные в пространстве имен `System.Data.Common`) и ключевые интерфейсы (определенные в пространстве имен `System.Data`), которые они реализуют.

Таблица 21.1. Основные типы поставщиков данных ADO.NET

Тип	Базовый класс	Реализуемые интерфейсы	Описание
`Connection`	`DbConnection`	`IDbConnection`	Предоставляет возможность подключения и отключения от хранилища данных. Объекты подключения также обеспечивают доступ к связанному объекту транзакции
`Command`	`DbCommand`	`IDbCommand`	Представляет запрос SQL или хранимую процедуру. Объекты команд также предоставляют доступ к объекту чтения данных поставщика
`DataReader`	`DbDataReader`	`IDataReader`, `IDataRecord`	Предоставляет доступ к данным в направлении только вперед, допускающий только чтение, с использованием курсора на стороне сервера
`DataAdapter`	`DbDataAdapter`	`IDataAdapter`, `IDbDataAdapter`	Передает объекты `DataSet` между вызывающим кодом и хранилищем данных. Адаптеры данных содержат объект подключения и набор из четырех внутренних объектов команд для выборки, вставки, изменения и удаления информации из хранилища данных
`Parameter`	`DbParameter`	`IDataParameter`, `IDbDataParameter`	Представляет именованный параметр внутри параметризированного запроса
`Transaction`	`DbTransaction`	`IDbTransaction`	Инкапсулирует транзакцию базы данных

Хотя конкретные имена основных классов будут отличаться между поставщиками данных (например, `SqlConnection` в сравнении с `OdbcConnection`), все они являются производными от того же самого базового класса (`DbConnection` в случае объектов подключения), реализующего идентичные интерфейсы (вроде `IDbConnection`). С учетом этого вполне корректно предположить, что после освоения работы с одним поставщиком данных остальные поставщики покажутся довольно простыми.

На заметку! Когда речь идет об *объекте подключения* в ADO.NET, то на самом деле имеется в виду специфичный тип, производный от `DbConnection`; не существует класса с именем *`Connection`*. Сказанное справедливо также в отношении *объекта команды*, *объекта адаптера данных* и т.д. По соглашению об именовании объекты в конкретном поставщике данных снабжаются префиксом в форме названия связанной СУБД (например, `SqlConnection`, `SqlCommand` и `SqlDataReader`).

На рис. 21.2 иллюстрируется место поставщиков данных в инфраструктуре ADO.NET. Следует отметить, что элемент "Клиентская сборка" на этой диаграмме может быть при-

ложением .NET буквально любого типа: консольной программой, приложением Windows Forms, приложением WPF, веб-страницей ASP.NET, службой WCF, службой Web API, библиотекой кода .NET и т.д.

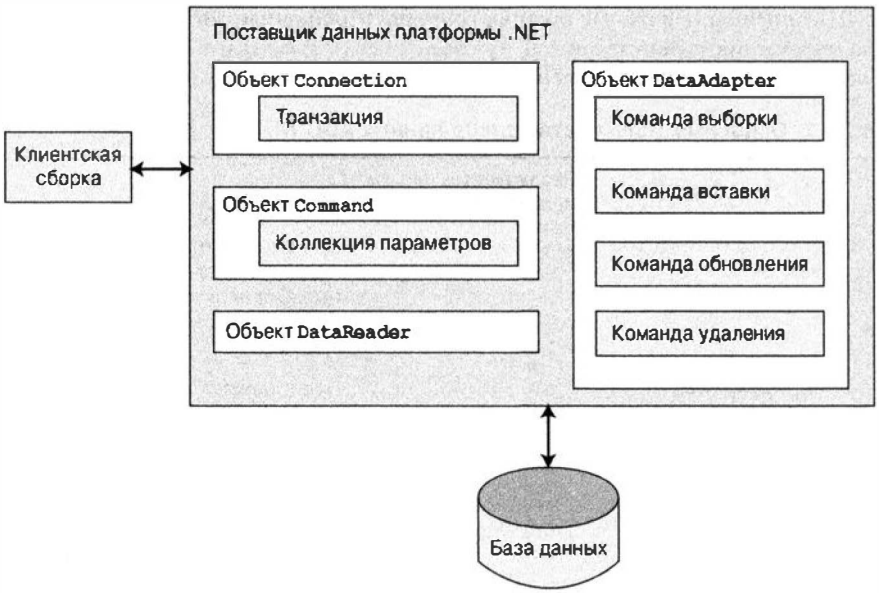


Рис. 21.2. Поставщики данных ADO.NET предоставляют доступ к конкретным СУБД

Кроме типов, показанных на рис. 21.2, поставщики данных будут предоставлять и другие типы; однако эти основные объекты определяют общие характеристики для всех поставщиков данных.

Поставщики данных ADO.NET производства Microsoft

Дистрибутив .NET от Microsoft поставляется с многочисленными поставщиками данных, включая поставщики для Oracle, SQL Server и подключений в стиле OLE DB/ODBC. В табл. 21.2 описаны пространства имен и содержащие их сборки для всех поставщиков данных Microsoft ADO.NET.

Таблица 21.2. Поставщики данных Microsoft ADO.NET

Поставщик данных	Пространство имен	Сборка
OLE DB	`System.Data.OleDb`	`System.Data.dll`
Microsoft SQL Server LocalDb	`System.Data.SqlClient`	`System.Data.dll`
ODBC	`System.Data.Odbc`	`System.Data.dll`

На заметку! В то время как поставщик Oracle по-прежнему входит в состав .NET Framework, рекомендуется применять инструменты разработчика Oracle для Visual Studio (Oracle Developer Tools for Visual Studio), поставляемые компанией Oracle. В действительности, если вы откроете окно проводника серверов (Server Explorer) и выберете элемент New Connection (Новое подключение), а затем Oracle Database (База данных Oracle), то Visual Studio сообщит о необходимости использования Oracle Data Tools и предоставит ссылку, по которой можно загрузить эти инструменты.

На заметку! Не существует специального поставщика данных, который бы отображался прямо на механизм Jet (и, следовательно, на Microsoft Access). Если вам нужно взаимодействовать с файлом данных Access, то можете делать это с применением поставщика данных OLE DB или ODBC.

Поставщик данных OLE DB, который состоит из типов, определенных в пространстве имен `System.Data.OleDb`, позволяет обращаться к данным, находящимся в любом хранилище данных, если оно поддерживает классический основанный на COM протокол OLE DB. Этот поставщик можно использовать для взаимодействия с любой базой данных, совместимой с OLE DB, просто подстраивая сегмент `Provider` строки подключения.

Тем не менее, "за кулисами" поставщик OLE DB взаимодействует с разнообразными объектами COM, что может повлиять на производительность приложения. В общем и целом поставщик данных OLE DB удобен, только если вы взаимодействуете с СУБД, для которой не определен специфичный поставщик данных .NET. Однако, учитывая, что в наши дни любая стоящая СУБД должна предлагать для загрузки специальный поставщик данных ADO.NET, пространство имен `System.Data.OleDb` следует считать устаревшим и малопригодным в мире .NET 4.6. (Это стало еще более справедливым с вводом в .NET 2.0 модели фабрики поставщиков данных, с которой вы вскоре ознакомитесь.)

На заметку! Есть один случай, когда необходимо применять типы из `System.Data.OleDb`: если нужно взаимодействовать с Microsoft SQL Server 6.5 или более ранней версии. Пространство имен `System.Data.SqlClient` допускает взаимодействие только с Microsoft SQL Server 7.0 и последующих версий.

Поставщик данных Microsoft SQL Server предлагает прямой доступ к хранилищам данных Microsoft SQL Server — и *только* к хранилищам данных SQL Server (7.0 и последующих версий). Пространство имен `System.Data.SqlClient` содержит типы, используемые поставщиком SQL Server, и обеспечивает ту же базовую функциональность, что и поставщик OLE DB. Главная разница между ними заключается в том, что поставщик SQL Server игнорирует уровень OLE DB и дает существенный выигрыш в производительности. Поставщик данных Microsoft SQL Server также позволяет получить доступ к уникальным возможностям этой конкретной СУБД.

Оставшийся поставщик от Microsoft (`System.Data.Odbc`) предоставляет доступ к подключениям ODBC. Типы ODBC, определенные в пространстве имен `System.Data.Odbc`, обычно полезны, только если требуется взаимодействие с СУБД, для которой отсутствует специальный поставщик данных .NET. Это так, потому что ODBC является широко распространенной моделью, которая предоставляет доступ к нескольким хранилищам данных.

Замечания относительно сборки `System.Data.OracleClient.dll`

Предшествующие версии платформы .NET поставлялись со сборкой `System.Data.OracleClient.dll`, которая предлагала поставщик данных для взаимодействия с базами данных Oracle. Тем не менее, в версии .NET 4.0 эта сборка была помечена как *устаревшая*, поэтому применять ее не рекомендуется. Клиент Oracle все еще включен в .NET 4.6, но разумно не надеяться, что он будет присутствовать и в будущем.

На первый взгляд может показаться, что инфраструктура ADO.NET постепенно становится сконцентрированной непосредственно на хранилищах данных от Microsoft, но это не так. Просто компания Oracle предоставляет собственную сборку .NET, которая следует тем же общим проектным принципам, что и поставщики данных, поставляемые Microsoft. Если вам понадобится эта сборка, посетите веб-сайт по адресу `http://www.oracle.com/technetwork/topics/dotnet/index-085163.html`.

Получение сторонних поставщиков данных ADO.NET

Кроме поставщиков данных, предлагаемых Microsoft (а также специальной библиотеки .NET от Oracle), существуют многочисленные сторонние поставщики данных для разнообразных баз данных с открытым кодом и коммерческих баз данных. Скорее всего, вы сможете получить поставщик данных ADO.NET непосредственно у разработчика СУБД, но вы должны знать о следующем веб-сайте:

```
https://msdn.microsoft.com/en-us/library/dd363565.aspx
```

Это один из множества веб-сайтов, где документируются все известные поставщики данных ADO.NET и указываются ссылки для получения дополнительных сведений и загрузки. Здесь вы найдете множество поставщиков ADO.NET, в том числе для SQLite, IBM DB2, MySQL, Postgres, Sybase и т.д.

Учитывая большое количество поставщиков данных ADO.NET, в примерах этой книги будет использоваться поставщик данных Microsoft SQL Server (`System.Data.SqlClient.dll`). Вспомните, что он позволяет взаимодействовать с Microsoft SQL Server 7.0 и последующих версий, включая SQL Server Express и LocalDb. Если вы намерены применять ADO.NET для работы с другой СУБД, то проблем возникать не должно при условии, что вы уясните изложенный ниже материал.

Дополнительные пространства имен ADO.NET

В дополнение к пространствам имен .NET, где определены типы специфических поставщиков данных, библиотеки базовых классов .NET предлагают несколько дополнительных пространств имен, относящихся к ADO.NET; некоторые из них описаны в табл. 21.3 (сборки и пространства имен, связанные с Entity Framework, рассматриваются в главе 23).

Таблица 21.3. Избранные дополнительные пространства имен, относящиеся к ADO.NET

Пространство имен	Описание
`Microsoft.SqlServer.Server`	Предоставляет типы, которые содействуют работе со службами интеграции CLR и SQL Server 2005 и последующих версий
`System.Data`	Определяет основные типы ADO.NET, используемые всеми поставщиками данных, в том числе общие интерфейсы и многочисленные типы, которые представляют автономный уровень (например, `DataSet` и `DataTable`)
`System.Data.Common`	Содержит типы, которые разделяются всеми поставщиками ADO.NET, включая общие абстрактные базовые классы
`System.Data.Sql`	Предоставляет типы, которые позволяют обнаружить экземпляры Microsoft SQL Server, установленные в текущей локальной сети
`System.Data.SqlTypes`	Определяет собственные типы данных, применяемые Microsoft SQL Server. Вы всегда можете использовать соответствующие типы данных CLR, но пространство имен `SqlTypes` оптимизировано для работы с SQL Server (скажем, если база данных SQL Server содержит целочисленное значение, то его можно представить с применением либо `int`, либо `SqlTypes.SqlInt32`)

Мы не собираемся исследовать каждый отдельный тип внутри абсолютно всех пространств имен ADO.NET (такая задача потребовала бы написания полноценной книги), но очень важно, чтобы вы понимали типы в пространстве имен `System.Data`.

Типы из пространства имен System.Data

Из всех пространств имен, связанных с ADO.NET, System.Data является "наименьшим общим знаменателем". Не указав это пространство имен, вы не сумеете построить приложение ADO.NET с доступом к данным. Оно содержит типы, которые совместно используются всеми поставщиками данных ADO.NET вне зависимости от лежащего в основе хранилища данных. В дополнение к нескольким исключениям, связанным с базами данных (например, NoNullAllowedException, RowNotInTableException и MissingPrimaryKeyException), пространство имен System.Data содержит типы, которые представляют разнообразные примитивы баз данных (вроде таблиц, строк, столбцов и ограничений), а также общие интерфейсы, реализуемые классами поставщиков данных. В табл. 21.4 описаны основные типы, которые следует знать.

Таблица 21.4. Основные типы пространства имен System.Data

Тип	Описание
Constraint	Представляет ограничение для заданного объекта DataColumn
DataColumn	Представляет одиночный столбец внутри объекта DataTable
DataRelation	Представляет отношение "родительский–дочерний" между двумя объектами DataTable
DataRow	Представляет одиночную строку внутри объекта DataTable
DataSet	Представляет находящийся в памяти кеш данных, который состоит из любого количества взаимосвязанных объектов DataTable
DataTable	Представляет табличный блок данных, находящихся в памяти
DataTableReader	Позволяет трактовать объект DataTable как курсор для доступа к данным в прямом направлении, допускающий только чтение
DataView	Определяет настраиваемое представление DataTable для сортировки, фильтрации, поиска, редактирования и навигации
IDataAdapter	Определяет основное поведение объекта адаптера данных
IDataParameter	Определяет основное поведение объекта параметра
IDataReader	Определяет основное поведение объекта чтения данных
IDbCommand	Определяет основное поведение объекта команды
IDbDataAdapter	Расширяет интерфейс IDataAdapter для обеспечения дополнительной функциональности объекту адаптера данных
IDbTransaction	Определяет основное поведение объекта транзакции

Подавляющее большинство классов из System.Data применяются во время программирования для автономного уровня ADO.NET. В следующей главе вы узнаете подробности о типе DataSet и связанной с ним группе типов (например, DataTable, DataRelation и DataRow), а также научитесь использовать их (и соответствующий адаптер данных) для представления и манипулирования копиями удаленных данных на стороне клиента.

Однако следующей задачей будет исследование основных интерфейсов System.Data на высоком уровне: это поможет лучше понять общую функциональность, предлагаемую любым поставщиком данных. В ходе чтения этой главы вы также ознакомитесь с конкретными деталями, но в настоящий момент лучше сосредоточить внимание на общем поведении каждого типа интерфейса.

Роль интерфейса IDbConnection

Интерфейс IDbConnection реализован *объектом подключения* поставщика данных. В нем определен набор членов, применяемых для конфигурирования подключения к специфичному хранилищу данных. Он также позволяет получить объект транзакции поставщика данных. Вот формальное определение IDbConnection:

```
public interface IDbConnection : IDisposable
{
  string ConnectionString { get; set; }
  int ConnectionTimeout { get; }
  string Database { get; }
  ConnectionState State { get; }

  IDbTransaction BeginTransaction();
  IDbTransaction BeginTransaction(IsolationLevel il);
  void ChangeDatabase(string databaseName);
  void Close();
  IDbCommand CreateCommand();
  void Open();
}
```

На заметку! Как и во многих других типах из библиотек базовых классов .NET, вызов метода Close() функционально эквивалентен прямому или косвенному вызову метода Dispose() внутри области using (см. главу 13).

Роль интерфейса IDbTransaction

Перегруженный метод BeginTransaction(), определенный в интерфейсе IDbConnection, предоставляет доступ к *объекту транзакции* поставщика. Члены, определенные интерфейсом IDbTransaction, позволяют программно взаимодействовать с транзакционным сеансом и лежащим в основе хранилищем данных:

```
public interface IDbTransaction : IDisposable
{
  IDbConnection Connection { get; }
  IsolationLevel IsolationLevel { get; }

  void Commit();
  void Rollback();
}
```

Роль интерфейса IDbCommand

Интерфейс IDbCommand будет реализован *объектом команды* поставщика данных. Как и другие объектные модели доступа к данным, объекты команд позволяют программно манипулировать с операторами SQL, хранимыми процедурами и параметризированными запросами. Объекты команд также обеспечивают доступ к типу чтения данных поставщика данных посредством перегруженного метода ExecuteReader():

```
public interface IDbCommand : IDisposable
{
  IDbConnection Connection { get; set; }
  IDbTransaction Transaction { get; set; }
  string CommandText { get; set; }
  int CommandTimeout { get; set; }
  CommandType CommandType { get; set; }
```

```
  IDataParameterCollection Parameters { get; }
  UpdateRowSource UpdatedRowSource { get; set; }

  void Prepare();
  void Cancel();
  IDbDataParameter CreateParameter();
  int ExecuteNonQuery();
  IDataReader ExecuteReader();
  IDataReader ExecuteReader(CommandBehavior behavior);
  object ExecuteScalar();
}
```

Роль интерфейсов IDbDataParameter и IDataParameter

Обратите внимание, что свойство `Parameters` интерфейса `IDbCommand` возвращает строго типизированную коллекцию, которая реализует интерфейс `IDataParameterCollection`. Этот интерфейс предоставляет доступ к набору классов, совместимых с `IDbDataParameter` (например, объектам параметров):

```
public interface IDbDataParameter : IDataParameter
{
  byte Precision { get; set; }
  byte Scale { get; set; }
  int Size { get; set; }
}
```

Интерфейс `IDbDataParameter` расширяет `IDataParameter` с целью обеспечения дополнительных линий поведения:

```
public interface IDataParameter
{
  DbType DbType { get; set; }
  ParameterDirection Direction { get; set; }
  bool IsNullable { get; }
  string ParameterName { get; set; }
  string SourceColumn { get; set; }
  DataRowVersion SourceVersion { get; set; }
  object Value { get; set; }
}
```

Вы увидите, что функциональность интерфейсов `IDbDataParameter` и `IDataParameter` позволяет представлять параметры внутри команды SQL (включая хранимые процедуры) с помощью специфических объектов параметров ADO.NET вместо жестко закодированных строковых литералов.

Роль интерфейсов IDbDataAdapter и IDataAdapter

Адаптеры данных будут использоваться для помещения и извлечения объектов `DataSet` в и из хранилища данных. Интерфейс `IDbDataAdapter` определяет следующий набор свойств, которые можно применять для поддержки операторов SQL, выполняющих связанные операции выборки, вставки, обновления и удаления:

```
public interface IDbDataAdapter : IDataAdapter
{
  IDbCommand SelectCommand { get; set; }
  IDbCommand InsertCommand { get; set; }
  IDbCommand UpdateCommand { get; set; }
  IDbCommand DeleteCommand { get; set; }
}
```

В дополнение к показанным четырем свойствам адаптер данных ADO.NET также получает поведение, определенное в базовом интерфейсе, т.е. IDataAdapter. Интерфейс IDataAdapter определяет ключевую функцию типа адаптера данных: способность передавать объекты DataSet между вызывающим кодом и внутренним хранилищем данных, используя методы Fill() и Update(). Кроме того, интерфейс IDataAdapter позволяет с помощью свойства TableMappings сопоставлять имена столбцов базы данных с более дружественными к пользователю отображаемыми именами:

```
public interface IDataAdapter
{
  MissingMappingAction MissingMappingAction { get; set; }
  MissingSchemaAction MissingSchemaAction { get; set; }
  ITableMappingCollection TableMappings { get; }

  DataTable[] FillSchema(DataSet dataSet, SchemaType schemaType);
  int Fill(DataSet dataSet);
  IDataParameter[] GetFillParameters();
  int Update(DataSet dataSet);
}
```

Роль интерфейсов IDataReader и IDataRecord

Следующим основным интерфейсом является IDataReader, который представляет общие линии поведения, поддерживаемые отдельно взятым объектом чтения данных. После получения от поставщика данных ADO.NET объекта совместимого с IDataReader типа можно выполнять проход по результирующему набору в прямом направлении с поддержкой только чтения.

```
public interface IDataReader : IDisposable, IDataRecord
{
  int Depth { get; }
  bool IsClosed { get; }
  int RecordsAffected { get; }

  void Close();
  DataTable GetSchemaTable();
  bool NextResult();
  bool Read();
}
```

Наконец, интерфейс IDataReader расширяет IDataRecord. В интерфейсе IDataRecord определено много членов, которые позволяют извлекать из потока строго типизированное значение, а не приводить к нужному типу экземпляр System.Object, полученный из перегруженного метода индексатора объекта чтения данных. Вот определение интерфейса IDataRecord:

```
public interface IDataRecord
{
  int FieldCount { get; }
  object this[ int i ] { get; }
  object this[ string name ] { get; }
  string GetName(int i);
  string GetDataTypeName(int i);
  Type GetFieldType(int i);
  object GetValue(int i);
  int GetValues(object[] values);
  int GetOrdinal(string name);
  bool GetBoolean(int i);
```

```
  byte GetByte(int i);
  long GetBytes(int i, long fieldOffset, byte[] buffer,
                int bufferoffset, int length);
  char GetChar(int i);
  long GetChars(int i, long fieldoffset, char[] buffer,
                int bufferoffset, int length);
  Guid GetGuid(int i);
  short GetInt16(int i);
  int GetInt32(int i);
  long GetInt64(int i);
  float GetFloat(int i);
  double GetDouble(int i);
  string GetString(int i);
  Decimal GetDecimal(int i);
  DateTime GetDateTime(int i);
  IDataReader GetData(int i);
  bool IsDBNull(int i);
}
```

На заметку! Метод `IDataReader.IsDBNull()` можно применять для программного выяснения, установлено ли указанное поле в `null`, прежде чем получать значение из объекта чтения данных (во избежание генерации исключения во время выполнения). Также вспомните, что язык C# поддерживает типы данных, допускающие `null` (см. главу 4), идеально подходящие для взаимодействия со столбцами, которые могут иметь значение `null` в таблице базы данных.

Абстрагирование поставщиков данных с использованием интерфейсов

К этому моменту вы должны лучше понимать общую функциональность, присущую всем поставщикам данных .NET. Вспомните, что хотя точные имена реализуемых типов будут отличаться между поставщиками данных, в коде эти типы применяются в схожей манере — в этом и заключается преимущество полиморфизма на основе интерфейсов. Скажем, если определить метод, который принимает параметр `IDbConnection`, то ему можно передавать любой объект подключения ADO.NET:

```
public static void OpenConnection(IDbConnection connection)
{
  // Открыть входное подключение для вызывающего кода.
  connection.Open();
}
```

На заметку! Использование интерфейсов совершенно не обязательно; аналогичного уровня абстракции можно достичь путем применения абстрактных базовых классов (таких как `DbConnection`) в качестве параметров или возвращаемых значений.

То же самое справедливо для возвращаемых значений. Например, рассмотрим простой проект консольного приложения C# (по имени `MyConnectionFactory`), который позволяет выбирать специфический объект подключения на основе значения из специального перечисления. В целях диагностики мы просто выводим лежащий в основе объект подключения с использованием служб рефлексии:

```
using System;
using static System.Console;
```

```
// Эти пространства имен нужны для получения определений
// общих интерфейсов и разнообразных объектов подключений.
using System.Data;
using System.Data.SqlClient;
using System.Data.Odbc;
using System.Data.OleDb;

namespace MyConnectionFactory
{
  // Список возможных поставщиков.
  enum DataProvider
  { SqlServer, OleDb, Odbc, None }

  class Program
  {
    static void Main(string[] args)
    {
      WriteLine("**** Very Simple Connection Factory *****\n");
      // Получить конкретное подключение.
      IDbConnection myConnection = GetConnection(DataProvider.SqlServer);
      WriteLine($"Your connection is a {myConnection.GetType().Name}");
      // Открыть, использовать и закрыть подключение...
      ReadLine();
    }
    // Этот метод возвращает конкретный объект подключения
    // на основе значения перечисления DataProvider.
    static IDbConnection GetConnection(DataProvider dataProvider)
    {
      IDbConnection connection = null;
      switch (dataProvider)
      {
        case DataProvider.SqlServer:
          connection = new SqlConnection();
          break;
        case DataProvider.OleDb:
          connection = new OleDbConnection();
          break;
        case DataProvider.Odbc:
          connection = new OdbcConnection();
          break;
      }
      return connection;
    }
  }
}
```

На заметку! В Visual Studio 2015 появилась директива `using static`. После добавления `using static System.Console;` к другим операторам `using` можно записывать `WriteLine("some text")` вместо `Console.WriteLine("some text")`. Для всех консольных проектов в этой и последующих главах будет применяться такая более короткая версия за счет добавления в начало файлов кода оператора `using static System.Console;`.

Преимущество работы с общими интерфейсами из пространства имен `System.Data` (или, если на то пошло, с абстрактными базовыми классами из пространства имен `System.Data.Common`) связано с более высокими шансами построить гибкую кодовую базу, которую со временем можно развивать. Например, в настоящий момент вы можете разрабатывать приложение, предназначенное для Microsoft SQL Server; тем не менее,

вполне возможно, что спустя несколько месяцев ваша компания переключится на другую СУБД. Если вы строите решение с жестко закодированными типами из пространства имен `System.Data.SqlClient`, которые специфичны для Microsoft SQL Server, то вполне очевидно, что в случае смены серверной СУБД сборку придется редактировать, заново компилировать и развертывать.

Увеличение гибкости с использованием конфигурационных файлов приложения

Для повышения гибкости приложений ADO.NET можно предусмотреть на стороне клиента файл `*.config`, в котором имеется элемент `<appSettings>`, содержащий специальные пары "ключ-значение". Вспомните из главы 14, что специальные данные, хранящиеся в файле `*.config`, можно получать программно с применением типов из пространства имен `System.Configuration`. Предположим, что внутри конфигурационного файла указано следующее значение поставщика данных:

```
<configuration>
  <appSettings>
    <!-- Это значение ключа отображается на одно из значений перечисления. -->
    <add key="provider" value="SqlServer"/>
  </appSettings>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6"/>
  </startup>
</configuration>
```

Теперь можно обновить метод `Main()`, чтобы программно получить объект поставщика данных. По существу тем самым создается *фабрика объектов подключений*, которая позволяет изменять объект поставщика без необходимости в повторной компиляции кодовой базы (просто модифицируется файл `*.config`). Ниже показаны необходимые изменения в `Main()`:

```
static void Main(string[] args)
{
  WriteLine("**** Very Simple Connection Factory *****\n");
  // Прочитать ключ provider.
  string dataProviderString = ConfigurationManager.AppSettings["provider"];
  // Преобразовать строку в перечисление.
  DataProvider dataProvider = DataProvider.None;
  if (Enum.IsDefined(typeof (DataProvider), dataProviderString))
  {
    dataProvider =
      (DataProvider) Enum.Parse(typeof (DataProvider), dataProviderString);
  }
  else
  {
    WriteLine("Sorry, no provider exists!");  // Поставщики отсутствуют
    ReadLine();
    return;
  }
  // Получить конкретное подключение.
  IDbConnection myConnection = GetConnection(dataProvider);
  WriteLine($"Your connection is a {myConnection?.GetType().Name ??
                                    "unrecognized type"}");
  // Открыть, использовать и закрыть подключение...
  ReadLine();
}
```

На заметку! Чтобы можно было использовать тип `ConfigurationManager`, необходимо установить ссылку на сборку `System.Configuration.dll` и импортировать пространство имен `System.Configuration`.

К настоящему моменту вы построили код ADO.NET, который позволяет динамически указывать лежащее в основе подключение. Однако здесь присутствует одна очевидная проблема: эта абстракция применяется только внутри приложения `MyConnectionFactory.exe`. Если переделать пример в виде библиотеки кода .NET (скажем, `MyConnectionFactory.dll`), то появилась бы возможность создавать любое количество клиентов, которые могли бы получать разнообразные объекты подключений, используя уровни абстракции.

Тем не менее, получение объекта подключения — лишь один аспект работы с ADO.NET. Чтобы построить полезную библиотеку фабрики поставщиков данных, необходимо также учитывать объекты команд, объекты чтения данных, адаптеры данных, объекты транзакций и другие типы, связанные с данными. Создание такой библиотеки кода не обязательно будет трудным, но все же потребует написания значительного объема кода и затрат времени.

Начиная с версии .NET 2.0, такая функциональность встроена прямо в библиотеки базовых классов .NET. Вскоре мы исследуем этот формальный API-интерфейс, но сначала понадобится создать специальную базу данных для применения в настоящей главе (и во многих последующих главах).

Исходный код. Проект `MyConnectionFactory` доступен в подкаталоге `Chapter_21`.

Создание базы данных AutoLot

На протяжении оставшегося материала текущей главы мы будем выполнять запросы в простой тестовой базе данных SQL Server по имени `AutoLot`. В продолжение автомобильной темы, затрагиваемой повсеместно в книге, эта база данных будет содержать три взаимосвязанных таблицы (`Inventory`, `Orders` и `Customers`), которые хранят различные данные о заказах гипотетической компании по продаже автомобилей.

В книге предполагается наличие у вас копии Microsoft SQL Server (7.0 или более новой версии) или копии Microsoft SQL Server Express. Если их у вас нет, произведите загрузку со следующей страницы:

```
http://www.microsoft.com/en-us/server-cloud/products/sql-server-editions/
overview.aspx
```

Этот легковесный сервер баз данных великолепно подходит для нужд книги: он бесплатен, предоставляет графический пользовательский интерфейс (SQL Server Management Tool) для создания и администрирования баз данных, а также интегрируется с Visual Studio/Visual Studio Community. Для иллюстрации последнего утверждения остаток этого раздела будет посвящен конструированию базы данных `AutoLot` с использованием Visual Studio.

На заметку! База данных `AutoLot` будет применяться в оставшихся главах книги.

Создание таблицы Inventory

Чтобы приступить к построению тестовой базы данных, необходимо запустить Visual Studio и открыть окно Server Explorer (Проводник серверов) через меню View (Просмотр). Затем следует щелкнуть правой кнопкой мыши на папке Data Connections

(Подключения к данным) и выбрать в контекстном меню пункт Create New SQL Server Database (Создать новую базу данных SQL Server), как показано на рис. 21.3.

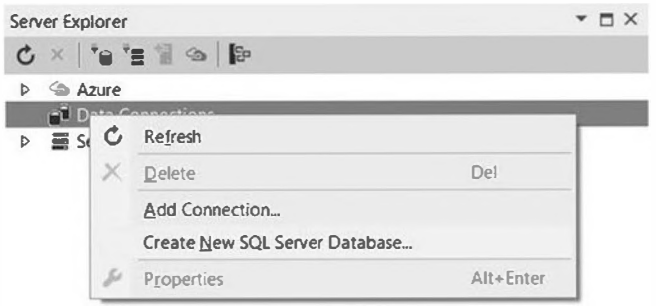


Рис. 21.3. Создание новой базы данных SQL Server в Visual Studio

В открывшемся диалоговом окне понадобится ввести значение в текстовом поле Server name (Имя сервера), которое представляет машину, где будет создана база данных. Имя сервера состоит из идентификатора машины и имени экземпляра. При наличии на машине установленного продукта Microsoft SQL Server (полной версии или версии Express) нужно ввести `(local)`, включая круглые скобки, или точку, за которой следует обратная косая черта и имя экземпляра (или оставить текстовое поле пустым, если используется стандартный экземпляр). Например, на машине у авторов вводится `.\SQLEXPRESS2014`.

Назначим новой базе данных имя `AutoLot` и оставим выбранным переключатель Use Windows Authentication (Использовать аутентификацию Windows), как продемонстрировано на рис. 21.4.

Рис. 21.4. Создание новой базы данных SQL Server Express с помощью Visual Studio

Пока что база данных `AutoLot` не содержит какие-либо объекты (таблицы, хранимые процедуры и т.д.). Для добавления новой таблицы потребуется щелкнуть правой кнопкой мыши на папке Tables (Таблицы) и выбрать в контекстном меню пункт Add New Table (Добавить новую таблицу), как показано на рис. 21.5.

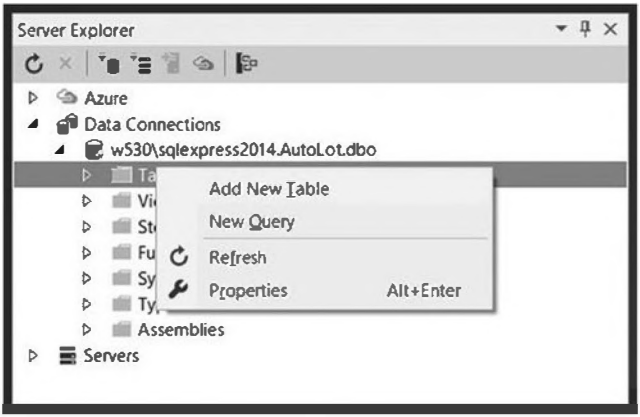


Рис. 21.5. Добавление таблицы `Inventory`

Посредством редактора таблиц добавим четыре столбца (`CarId` (идентификатор автомобиля), `Make` (модель), `Color` (цвет) и `PetName` (дружественное имя)). Укажем для столбца `CarId` тип `int`, а для остальных — тип `nvarchar(50)`. Необходимо удостовериться, что столбец `CarId` установлен как первичный ключ (щелкнув правой кнопкой мыши на `CarId` и выбрав в контекстном меню пункт Set Primary Key (Установить первичный ключ)), а также в качестве описания идентичности (добавив параметр `IDENTITY` в оператор SQL или отметив для столбца `CarId` флажок в колонке Identity (Идентичность), как проиллюстрировано на рис. 21.6). Также обратите внимание, что всем столбцам кроме `CarId` могут присваиваться значения `null`. Финальные параметры таблицы представлены на рис. 21.6.

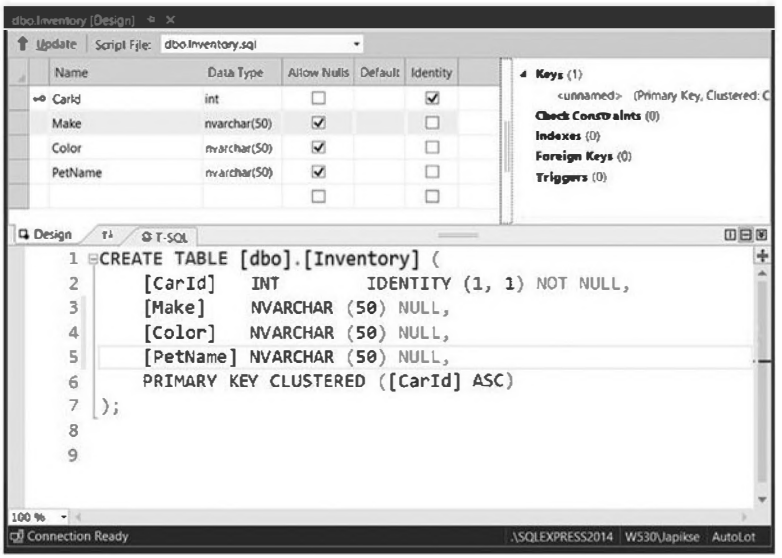


Рис. 21.6. Проектирование таблицы `Inventory`

После создания схемы таблицы назначим ей имя `Inventory` в окне T-SQL и сохраним результаты работы, щелкнув на кнопке Update (Обновить). На следующем экране (рис. 21.7) понадобится щелкнуть на кнопке Update Database (Обновить базу данных), чтобы зафиксировать действие.

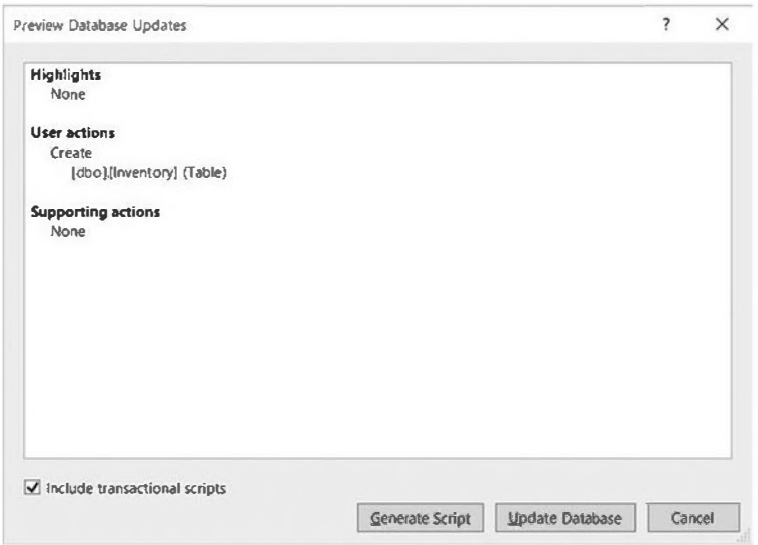


Рис. 21.7. Фиксация таблицы `Inventory` в базе данных

Добавление тестовых записей в таблицу `Inventory`

Чтобы добавить записи в эту первую таблицу, следует щелкнуть правой кнопкой мыши на ее значке и выбрать в контекстном меню пункт Show Table Data (Показать данные таблицы). Имейте в виду, что если таблица `Inventory` не видна, щелкните правой кнопкой мыши на папке Tables и выберите в контекстном меню пункт Refresh (Обновить). Введите данные о нескольких новых автомобилях по своему выбору (для интереса сделайте так, чтобы у некоторых автомобилей были одинаковые цвета и модели). Не забывайте, что поле `CarId` является столбцом идентичности, поэтому база данных самостоятельно позаботится о создании уникальных значений для него. На рис. 21.8 приведен возможный вариант списка товаров.

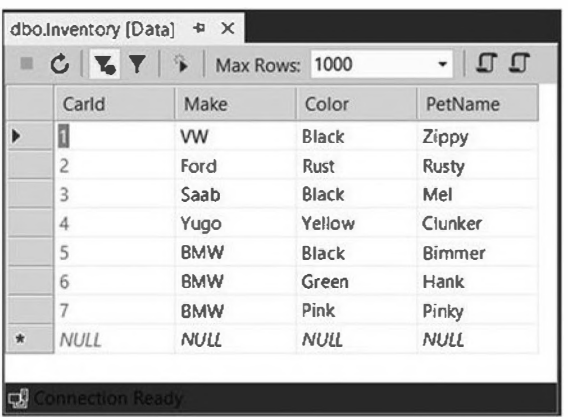


Рис. 21.8. Наполнение таблицы `Inventory`

Создание хранимой процедуры GetPetName()

Позже в этой главе вы научитесь применять ADO.NET для вызова хранимых процедур. Как вам уже может быть известно, хранимые процедуры представляют собой подпрограммы, находящиеся внутри определенной базы данных, которые часто оперируют на табличных данных для выдачи каких-то значений. Мы добавим в базу данных единственную хранимую процедуру, которая будет возвращать дружественное имя автомобиля по его идентификатору. Для этого нужно щелкнуть правой кнопкой мыши на папке Stored Procedures (Хранимые процедуры) базы данных `AutoLot` в окне Server Explorer и выбрать в контекстном меню пункт Add New Stored Procedure (Добавить новую хранимую процедуру). В открывшемся окне редактора необходимо ввести следующий код:

```
CREATE PROCEDURE GetPetName
@carID int,
@petName char(10) output
AS
SELECT @petName = PetName from Inventory where CarId = @carID
```

На заметку! Хранимые процедуры не обязаны возвращать данные, используя выходные параметры, как сделано здесь; однако это формирует почву для обсуждения свойства `Direction` класса `SqlParameter` далее в главе.

После щелчка на кнопке Update для сохранения процедуры она автоматически получит имя `GetPetName`, основанное на операторе `CREATE PROCEDURE`. Затем новая хранимая процедура появится в окне Server Explorer (рис. 21.9).

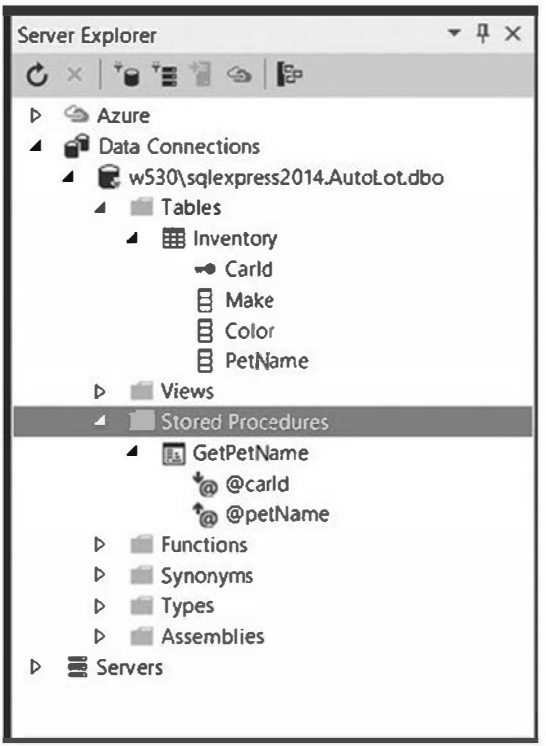


Рис. 21.9. Хранимая процедура `GetPetName`

Создание таблиц Customers и Orders

База данных AutoLot будет иметь две дополнительные таблицы: Customers (клиенты) и Orders (заказы). Таблица Customers будет хранить список клиентов и содержать три столбца: CustId (идентификатор клиента; должен быть установлен в качестве первичного ключа), FirstName (имя) и LastName (фамилия). Создать таблицу Inventory можно, следуя тем же самым шагам, которые выполнялись при создании таблицы Inventory; на рис. 21.10 показана схема таблицы Customers.

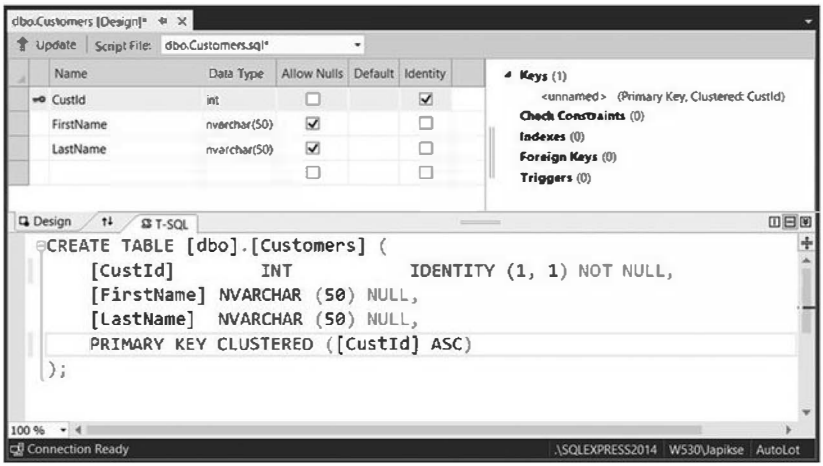


Рис. 21.10. Проектирование таблицы Customers

После сохранения и именования таблицы добавим в нее несколько записей для представления клиентов (рис. 21.11).

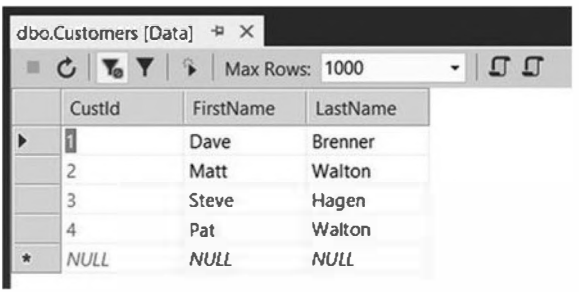


Рис. 21.11. Наполнение таблицы Customers

Финальная таблица, Orders, будет применяться для представления автомобиля, который клиент заинтересован приобрести. Это делается путем отображения значений OrderId на значения CarId/CustId. Структура таблицы Orders приведена на рис. 21.12 (обратите внимание, что OrderId является первичным ключом).

Теперь добавим данные в таблицу Orders. Ни одного отношения между таблицами пока еще не создано, так что понадобится вручную ввести значения, существующие в таблицах. Для каждого значения CustId потребуется выбрать уникальное значение CarId (на рис. 21.13 показаны записи, которые основаны на данных, ранее помещенных в таблицы).

Например, записи здесь указывают на то, что Дэйв Бреннер (Dave Brenner; CustId = 1) интересуется черным автомобилем марки BMW (CarId = 5), а Пэт Уолтон (Pat Walton; CustId = 4) остановила свой выбор на розовом BMW (CarId = 7).

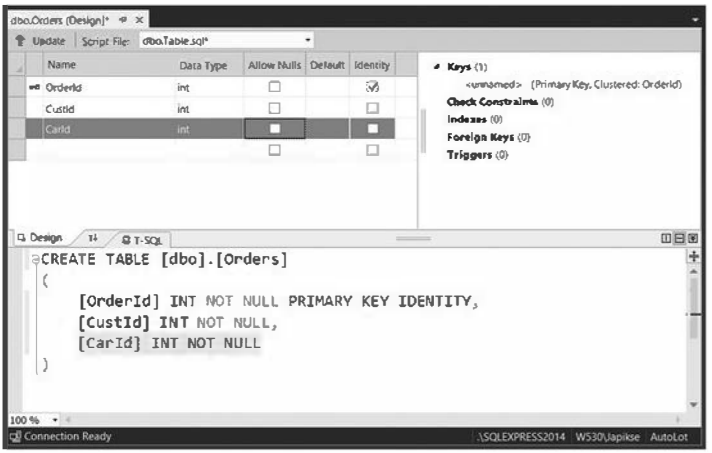


Рис. 21.12. Проектирование таблицы `Orders`

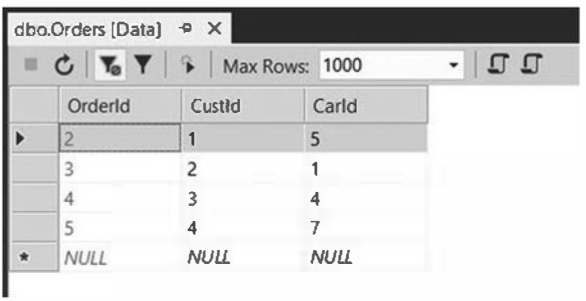


Рис. 21.13. Наполнение таблицы `Orders`

Создание отношений между таблицами в Visual Studio

Последней задачей будет установление отношений "родительский–дочерний" между таблицами `Customers`, `Orders` и `Inventory`. Для этого нужно щелкнуть правой кнопкой мыши на значке таблицы `Orders` и выбрать в контекстном меню пункт Open Table Definition (Открыть определение таблицы). Справа от сетки со столбцами следует щелкнуть правой кнопкой мыши на элементе Foreign Keys (Внешние ключи) и выбрать в контекстном меню пункт Add New Foreign Key (Добавить новый внешний ключ), как показано на рис. 21.14.

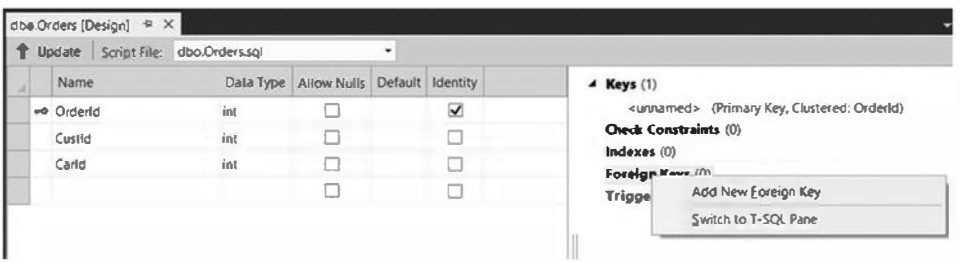


Рис. 21.14. Создание нового внешнего ключа

Стандартное имя для внешнего ключа выглядит как `FK_<ТекущаяТаблица>_ToTable`. В этом случае предлагается `FK_Orders_ToTable`. Заменим `ToTable` именем таблицы `Inventory`.

На момент написания этой главы существовала проблема с сеткой свойств для внешних ключей, так что оператор SQL придется обновить вручную. В окне редактора кода T-SQL строку:

```
CONSTRAINT [FK_Orders_Inventory] FOREIGN KEY ([Column])
  REFERENCES [ToTable]([ToTableColumn])
```

следует привести к такому виду:

```
CONSTRAINT [FK_Orders_Inventory] FOREIGN KEY ([CarId])
  REFERENCES [Inventory]([CarId]),
```

Скопируем конструкцию `CONSTRAINT` в новую строку и модифицируем ее так, как показано ниже (после конструкции `CONSTRAINT` должна присутствовать запятая):

```
CONSTRAINT [FK_Orders_Customers] FOREIGN KEY ([CustId])
  REFERENCES [dbo].[Customers] ([CustId]),
```

Полный оператор SQL для этой таблицы выглядит следующим образом (в зависимости от используемой версии SQL Server могут наблюдаться незначительные отличия):

```
CREATE TABLE [dbo].[Orders] (
  [OrderId] INT IDENTITY (1, 1) NOT NULL,
  [CustId]  INT NOT NULL,
  [CarId]   INT NOT NULL,
  PRIMARY KEY CLUSTERED ([OrderId] ASC),
  CONSTRAINT [FK_Orders_Inventory] FOREIGN KEY ([CarId])
    REFERENCES [Inventory]([CarId]),
  CONSTRAINT [FK_Orders_Customers] FOREIGN KEY ([CustId])
    REFERENCES [Customers]([CustId]),
);
```

После щелчка на кнопке Update в редакторе кода T-SQL и затем на кнопке Update Database (Обновить базу данных) в открывшемся далее диалоговом окне отношения добавятся к базе данных `AutoLot`. Если во время обновления базы данных возникает ошибка, то это значит, что данные, введенные в таблицу `Orders`, оказались некорректными. Возможно, вы поместили в таблицу `Orders` запись со значением `CustId`, которое в таблице `Customers` отсутствует. Исправьте данные и обновите таблицу снова.

На этом создание базы данных `AutoLot` завершено. Разумеется, она сильно отличается от реальной корпоративной базы данных, но успешно будет удовлетворять всем нуждам в оставшемся материале книги. Теперь, имея тестовую базу, можно приступить к исследованию деталей, касающихся модели фабрики поставщиков данных ADO.NET.

Модель фабрики поставщиков данных ADO.NET

Шаблон фабрики поставщиков данных .NET позволяет строить единую кодовую базу с применением обобщенных типов доступа к данным. Кроме того, используя конфигурационные файлы приложения (и подэлемент `<connectionStrings>`), можно получать поставщики и строки подключения декларативным образом без необходимости в повторной компиляции или развертывания сборки, в которой применяются API-интерфейсы ADO.NET.

Чтобы разобраться в реализации фабрики поставщиков данных, вспомните из табл. 21.1, что все классы поставщиков данных являются производными от тех же самых базовых классов, определенных внутри пространства имен `System.Data.Common`:

- `DbCommand` — абстрактный базовый класс для всех классов команд;
- `DbConnection` — абстрактный базовый класс для всех классов подключений;

- `DbDataAdapter` — абстрактный базовый класс для всех классов адаптеров данных;
- `DbDataReader` — абстрактный базовый класс для всех классов чтения данных;
- `DbParameter` — абстрактный базовый класс для всех классов параметров;
- `DbTransaction` — абстрактный базовый класс для всех классов транзакций.

Каждый поставщик данных, предлагаемый Microsoft, содержит класс, производный от `System.Data.Common.DbProviderFactory`. В этом базовом классе определено несколько методов, которые извлекают объекты данных, специфичные для поставщика. Вот члены класса `DbProviderFactory`:

```
public abstract class DbProviderFactory
{
  public virtual bool CanCreateDataSourceEnumerator { get;};
  public virtual DbCommand CreateCommand();
  public virtual DbCommandBuilder CreateCommandBuilder();
  public virtual DbConnection CreateConnection();
  public virtual DbConnectionStringBuilder CreateConnectionStringBuilder();
  public virtual DbDataAdapter CreateDataAdapter();
  public virtual DbParameter CreateParameter();
  public virtual CodeAccessPermission CreatePermission(PermissionState state);
  public virtual DbDataSourceEnumerator CreateDataSourceEnumerator();
}
```

Чтобы получить объект производного от `DbProviderFactory` типа для вашего поставщика данных, нужно использовать класс `DbProviderFactories` из пространства имен `System.Data.Common`. С помощью статического метода `GetFactory()` можно получить конкретный объект `DbProviderFactory` указанного поставщика данных; для этого понадобится передать строковое имя, которое представляет пространство имен .NET, содержащее функциональность поставщика:

```
static void Main(string[] args)
{
  // Получить фабрику для поставщика данных SQL.
  DbProviderFactory sqlFactory =
    DbProviderFactories.GetFactory("System.Data.SqlClient");
  ...
}
```

Конечно, вместо применения жестко закодированного строкового литерала получить фабрику можно было бы, прочитав эту информацию из файла `*.config` на стороне клиента (во многом так же, как в ранее рассмотренном примере `MyConnectionFactory`). Вскоре вы узнаете, как это делать, а пока после получения фабрики для поставщика данных можно получить связанные с ним объекты данных (такие как объекты подключений, объекты команд и объекты чтения данных).

На заметку! Для всех практических целей аргумент, передаваемый методу `DbProviderFactories.GetFactory()`, можно рассматривать как название пространства имен .NET поставщика данных. В реальности это строковое значение используется в `machine.config` для динамической загрузки корректной библиотеки из глобального кеша сборок.

Полный пример фабрики поставщиков данных

Для демонстрации завершенного примера мы создадим новый проект консольного приложения C# (по имени DataProviderFactory), которое выводит инвентарный список автомобилей из базы данных AutoLot. В этом начальном примере мы жестко закодируем логику доступа к данным прямо в сборке DataProviderFactory.exe (чтобы пока излишне не усложнять код). Тем не менее, когда мы начнем углубляться в подробности модели программирования ADO.NET, логика доступа к данным будет изолирована в специальной библиотеке кода .NET, которая будет применяться во всем оставшемся тексте книги.

Первым делом добавим ссылку на сборку System.Configuration.dll и импортируем пространство имен System.Configuration. Затем модифицируем файл App.config, включив в него пустой элемент <appSettings>. Добавим новый ключ provider, отображающийся на название пространства имен поставщика данных, который требуется получить (System.Data.SqlClient). Определим также строку подключения, которая представляет подключение к базе данных AutoLot (на локальном экземпляре SQL Server Express):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <appSettings>
  <!-- Какой поставщик? -->
  <add key="provider" value="System.Data.SqlClient" />
  <!-- Какая строка подключения? -->
  <add key="connectionString" value= "Data Source=(local)\SQLEXPRESS2014;
            Initial Catalog=AutoLot;Integrated Security=True"/>
 </appSettings>
 <startup>
  <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6"/>
 </startup>
</configuration>
```

На заметку! Чуть позже мы рассмотрим строки подключения более детально; однако если вы выберете значок базы данных AutoLot в окне Server Explorer, то сможете скопировать корректную строку подключения из свойства Connection String (Строка подключения) внутри окна свойств среды Visual Studio.

Теперь, когда имеется подходящий файл *.config, можно прочитать значения provider и connectionString с использованием индексатора ConfigurationManager.AppSettings. Значение provider будет передано методу DbProviderFactories.GetFactory() с целью получения объекта типа фабрики, специфичного для поставщика данных. Значение connectionString будет применяться для установки свойства ConnectionString объекта производного от DbConnection типа.

Предполагая, что были импортированы пространства имен System.Data и System.Data.Common, а также добавлен оператор using static System.Console;, метод Main() можно обновить следующим образом:

```
static void Main(string[] args)
{
  WriteLine("***** Fun with Data Provider Factories *****\n");
  // Получить строку подключения и поставщик из файла *.config.
  string dataProvider =
    ConfigurationManager.AppSettings["provider"];
```

```
    string connectionString =
      ConfigurationManager.AppSettings["connectionString"];

    // Получить фабрику поставщиков.
    DbProviderFactory factory = DbProviderFactories.GetFactory(dataProvider);

    // Получить объект подключения.
    using (DbConnection connection = factory.CreateConnection())
    {
      if (connection == null)
      {
        ShowError("Connection");
        return;
      }
      WriteLine($"Your connection object is a: {connection.GetType().Name}");
      connection.ConnectionString = connectionString;
      connection.Open();

      // Создать объект команды.
      DbCommand command = factory.CreateCommand();
      if (command == null)
      {
        ShowError("Command");
        return;
      }
      WriteLine($"Your command object is a: {command.GetType().Name}");
      command.Connection = connection;
      command.CommandText = "Select * From Inventory";

      // Вывести данные с помощью объекта чтения данных.
      using (DbDataReader dataReader = command.ExecuteReader())
      {
        WriteLine($"Your data reader object is a: {dataReader.GetType().Name}");

        WriteLine("\n***** Current Inventory *****");
        while (dataReader.Read())
          WriteLine($"-> Car #{dataReader["CarId"]} is a {dataReader["Make"]}.");
      }
    }
    ReadLine();
  }

  private static void ShowError(string objectName)
  {
    WriteLine($"There was an issue creating the {objectName}");
              // Возникла проблема с созданием объекта
    ReadLine();
  }
```

Обратите внимание, что в целях диагностики с помощью служб рефлексии выводятся имена лежащих в основе объектов подключения, команды и чтения данных. В результате запуска приложения в окне консоли отобразятся текущие данные из таблицы `Inventory` **базы данных** `AutoLot`:

```
***** Fun with Data Provider Factories *****

Your connection object is a: SqlConnection
Your command object is a: SqlCommand
Your data reader object is a: SqlDataReader
```

```
***** Current Inventory *****
-> Car #1 is a VW.
-> Car #2 is a Ford.
-> Car #3 is a Saab.
-> Car #4 is a Yugo.
-> Car #5 is a BMW.
-> Car #6 is a BMW.
-> Car #7 is a BMW.
```

Теперь изменим содержимое файла `*.config`, указав `System.Data.OleDb` в качестве поставщика данных (а также модифицируем строку подключения с сегментом `Provider` и сменим значение `Integrated Security` с `true` на `SSPI`):

```
<configuration>
 <appSettings>
  <!-- Какой поставщик? -->
  <add key="provider" value="System.Data.OleDb" />
  <!-- Какая строка подключения? -->
  <add key="connectionString" value= "Data Source=(local)\SQLEXPRESS2014;
            Initial Catalog=AutoLot;Integrated Security=SSPI"/>
 </appSettings>
 <startup>
  <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6"/>
 </startup>
</configuration>
```

Вы обнаружите, что "за кулисами" использовались типы из пространства имен `System.Data.OleDb`; ниже показан вывод:

```
***** Fun with Data Provider Factories *****

Your connection object is a: OleDbConnection
Your command object is a: OleDbCommand
Your data reader object is a: OleDbDataReader

***** Current Inventory *****
-> Car #1 is a VW.
-> Car #2 is a Ford.
-> Car #3 is a Saab.
-> Car #4 is a Yugo.
-> Car #5 is a BMW.
-> Car #6 is a BMW.
-> Car #7 is a BMW.
```

Конечно, в зависимости от опыта работы с ADO.NET у вас может не быть полного понимания того, что в действительности *делают* объекты подключений, команд и чтения данных. Не вдаваясь в детали, пока просто запомните, что модель фабрики поставщиков данных ADO.NET позволяет строить единственную кодовую базу, которая может потреблять разнообразные поставщики данных в декларативной манере.

Потенциальный недостаток модели фабрики поставщиков данных

Хотя модель фабрики поставщиков данных отличается довольно высокой мощностью, вы должны гарантировать, что в кодовой базе применяются только типы и методы, которые являются общими для всех поставщиков благодаря членам абстрактных базовых классов. Следовательно, при разработке кодовой базы вы ограничены членами `DbConnection`, `DbCommand` и других типов из пространства имен `System.Data.Common`.

С учетом этого вы можете прийти к заключению, что такой обобщенный подход предотвращает прямой доступ к дополнительным возможностям отдельной СУБД. Если вы должны быть в состоянии обращаться к специфическим членам лежащего в основе поставщика (например, SqlConnection), то сможете это делать с использованием явного приведения, как показано ниже:

```
using (DbConnection connection = factory.CreateConnection())
{
  if (connection == null)
  {
    ShowError("Connection");
    return;
  }
  WriteLine($"Your connection object is a: {connection.GetType().Name}");
  connection.ConnectionString = connectionString;
  connection.Open();

  var sqlConnection = connection as SqlConnection;
  if (sqlConnection != null)
  {
    // Вывести информацию об используемой версии SQL Server.
    WriteLine(sqlConnection.ServerVersion);
  }
  // Ради краткости оставшийся код удален.
}
```

Однако в таком случае кодовая база становится немного труднее в сопровождении (и менее гибкой), потому что придется добавить некоторое количество проверок времени выполнения. Тем не менее, если необходимо строить библиотеки доступа к данным самым гибким способом из возможных, то модель фабрики поставщиков данных предлагает для этого замечательный механизм.

Элемент <connectionStrings>

В настоящий момент данные строки подключения находятся в элементе <appSettings> внутри файла *.config. В конфигурационных файлах приложений допускается указывать элемент <connectionStrings>. Внутри него можно определять любое количество пар "имя-значение", которые будут программно читаться в память с применением индексатора ConfigurationManager.ConnectionStrings. Одно из преимуществ такого подхода (в противоположность использованию элемента <appSettings> и индексатора ConfigurationManager.AppSettings) связано с возможностью определения нескольких строк подключения для одного приложения в согласованной манере.

Чтобы посмотреть на все это в действии, модифицируем текущий файл App.config следующим образом (обратите внимание, что каждая строка подключения описана с применением атрибутов name и connectionString, а не key и value, как в <appSettings>):

```
<configuration>
 <appSettings>
  <!-- Какой поставщик? -->
  <add key="provider" value="System.Data.SqlClient" />
 </appSettings>

 <!-- Несколько строк подключения. -->
 <connectionStrings>
   <add name ="AutoLotSqlProvider" connectionString =
```

```
      "Data Source=(local)\SQLEXPRESS2014;
      Integrated Security=SSPI;Initial Catalog=AutoLot"/>
   <add name ="AutoLotOleDbProvider" connectionString =
      "Provider=SQLOLEDB;Data Source=(local)\SQLEXPRESS2014;
      Integrated Security=SSPI;Initial Catalog=AutoLot"/>
 </connectionStrings>
 <startup>
  <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6"/>
 </startup>
</configuration>
```

Теперь можно обновить метод `Main()`:

```
static void Main(string[] args)
{
  WriteLine("***** Fun with Data Provider Factories *****\n");
  string dataProvider =
   ConfigurationManager.AppSettings["provider"];
  string connectionString =
   ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"].ConnectionString;
  ...
}
```

К настоящему моменту вы располагаете приложением, которое способно отображать содержимое таблицы `Inventory` **базы данных** `AutoLot`, **используя нейтральную кодовую базу. Вынесение имени поставщика и строки подключения во внешний файл** `*.config` **означает, что модель фабрики поставщиков данных может динамически загружать корректный поставщик на заднем плане. Первый пример завершен, и теперь можно углубиться в детали работы с подключенным уровнем ADO.NET.**

На заметку! Теперь, когда вы понимаете роль фабрик поставщиков данных ADO.NET, в оставшихся примерах книги внимание будет сосредоточено на текущих задачах за счет явного применения типов из пространства имен `System.Data.SqlClient`. Если вы используете другую СУБД (такую как Oracle), то понадобится соответствующим образом изменить кодовую базу.

Исходный код. Проект `DataProviderFactory` доступен в подкаталоге `Chapter_21`.

Понятие подключенного уровня ADO.NET

Вспомните, что *подключенный уровень* **ADO.NET позволяет взаимодействовать с базой данных с помощью объектов подключения, чтения данных и команд вашего поставщика данных. Вы уже применяли упомянутые объекты в предыдущем приложении** `DataProviderFactory`, **и мы пройдем через процесс снова, используя на этот раз расширенный пример. Чтобы подключиться к базе данных и прочитать записи с помощью объекта чтения данных, необходимо выполнить следующие шаги.**

1. **Создать, сконфигурировать и открыть объект подключения.**
2. **Создать и сконфигурировать объект команды, указав объект подключения в аргументе конструктора или посредством свойства** `Connection`.
3. **Вызвать метод** `ExecuteReader()` **на сконфигурированном объекте команды.**
4. **Обработать каждую запись с применением метода** `Read()` **объекта чтения данных.**

Итак, для начала создадим новый проект консольного приложения по имени AutoLotDataReader и импортируем пространства имен System.Data и System.Data.SqlClient. Ниже приведен полный код метода Main() с последующим анализом:

```
class Program
{
  static void Main(string[] args)
  {
    WriteLine("***** Fun with Data Readers *****\n");

    // Создать и открыть подключение.
    using (SqlConnection connection = new SqlConnection())
    {
      connection.ConnectionString =
        @"Data Source=(local)\SQLEXPRESS2014;Integrated Security=SSPI;" +
        "Initial Catalog=AutoLot";
      connection.Open();

      // Создать объект команды SQL.
      string sql = "Select * From Inventory";
      SqlCommand myCommand = new SqlCommand(sql, connection);

      // Получить объект чтения данных с помощью ExecuteReader().
      using (SqlDataReader myDataReader = myCommand.ExecuteReader())
      {
        // Пройти в цикле по результатам.
        while (myDataReader.Read())
        {
          WriteLine($"-> Make: {myDataReader["Make"]},
     PetName: {myDataReader["PetName"]}, Color: {myDataReader["Color"]}.");
        }
      }
    }
    ReadLine();
  }
}
```

Работа с объектами подключения

При работе с поставщиком данных первым делом понадобится установить сеанс с источником данных, используя объект подключения (производного от DbConnection типа). Объекты подключений .NET обеспечиваются форматированной *строкой подключения*, которая содержит несколько пар "имя-значение", разделенных точками с запятой. Эта информация идентифицирует имя машины, к которой нужно подключиться, требуемые настройки безопасности, имя базы данных на машине и другие специфичные для поставщика сведения.

Из приведенного выше кода можно сделать вывод, что имя Initial Catalog относится к базе данных, с которой необходимо установить сеанс. Имя Data Source указывает имя машины, на которой находится база данных. Здесь (local) позволяет задать текущую локальную машину (независимо от ее конкретного имени), а конструкция \SQLEXPRESS2014 сообщает поставщику SQL Server о том, что требуется подключение к экземпляру SQL Server Express по имени SQLEXPRESS2014. Если база данных AutoLot была создана в стандартном экземпляре Microsoft SQL Server на локальном компьютере, то следует указать Data Source=(local).

На заметку! Существует еще один механизм, в котором для имени сервера применяется `(LocalDb)`. Он сохраняет базу данных вместе с проектом/решением. Вы по-прежнему можете обращаться к такой базе данных через окно Server Explorer, используя `(LocalDb)\MSSQLLocalDB` в качестве имени сервера (или `(LocalDb)\v11.0` в зависимости от версии начальных шаблонов Visual Studio).

Кроме того, можно указать любое количество конструкций, которые представляют учетные данные безопасности. Здесь `Integrated Security` устанавливается в `SSPI` (эквивалент `true` для строк подключения SQL Server), что предусматривает применение для аутентификации текущей учетной записи Windows.

На заметку! Для получения дополнительных сведений по каждой паре "имя-значение", связанной со специфичной СУБД, ищите описание свойства `ConnectionString` объекта подключения поставщика данных в документации .NET Framework 4.6 SDK.

Когда строка подключения готова, можно вызывать метод `Open()` для установления подключения к базе данных. В дополнение к членам `ConnectionString`, `Open()` и `Close()` объект подключения предоставляет несколько членов, которые позволяют конфигурировать дополнительные настройки подключения, такие как таймаут и транзакционная информация. В табл. 21.5 кратко описаны избранные члены базового класса `DbConnection`.

Таблица 21.5. Избранные члены типа `DbConnection`

Член	Описание
`BeginTransaction()`	Этот метод позволяет начать транзакцию базы данных
`ChangeDatabase()`	Этот метод изменяет базу данных, связанную с открытым подключением
`ConnectionTimeout`	Это свойство только для чтения возвращает промежуток времени, в течение которого происходит ожидание установления подключения, прежде чем будет сгенерирована ошибка (стандартное значение составляет 15 секунд). Чтобы изменить стандартное значение, необходимо указать в строке подключения конструкцию `Connect Timeout` (например, `Connect Timeout=30`)
`Database`	Это свойство только для чтения возвращает имя базы данных, обслуживаемой объектом подключения
`DataSource`	Это свойство только для чтения возвращает местоположение базы данных, обслуживаемой объектом подключения
`GetSchema()`	Этот метод возвращает объект `DataTable`, содержащий информацию схемы из источника данных
`State`	Это свойство только для чтения возвращает текущее состояние подключения, представленное перечислением `ConnectionState`

Свойства типа `DbConnection` обычно по своей природе допускают только чтение и полезны, только когда требуется получить характеристики подключения во время выполнения. Если необходимо переопределить стандартные настройки, то придется изменить саму строку подключения. Например, в следующей строке подключения время таймаута устанавливается равным 30 секундам вместо 15:

```
static void Main(string[] args)
{
  WriteLine("***** Fun with Data Readers *****\n");

  using(SqlConnection connection = new SqlConnection())
  {
    connection.ConnectionString =
      @"Data Source=(local)\SQLEXPRESS2014;" +
      "Integrated Security=SSPI;Initial Catalog=AutoLot;Connect Timeout=30";
    connection.Open();

  // Новая вспомогательная функция (см. ниже).
  ShowConnectionStatus(connection);
  ...
}
```

В приведенном выше коде объект подключения передается в виде параметра новому статическому вспомогательному методу класса `Program` по имени `ShowConnectionStatus()` с такой реализацией:

```
static void ShowConnectionStatus(SqlConnection connection)
{
  // Вывести различные сведения о текущем объекте подключения.
  WriteLine("***** Info about your connection *****");
  WriteLine($"Database location: {connection.DataSource}");
            // Местоположение базы данных
  WriteLine($"Database name: {connection.Database}");     // Имя базы данных
  WriteLine($"Timeout: {connection.ConnectionTimeout}"); // Таймаут
  WriteLine($"Connection state: {connection.State}\n");  // Состояние
}
```

Большинство этих свойств понятно без объяснений, но свойство `State` требует дополнительного внимания. Ему можно присвоить любое значение из перечисления `ConnectionState`:

```
public enum ConnectionState
{
  Broken, Closed,
  Connecting, Executing,
  Fetching, Open
}
```

Однако допустимыми значениями `ConnectionState` будут только `ConnectionState.Open`, `ConnectionState.Connecting` и `ConnectionState.Closed` (остальные члены перечисления зарезервированы для будущего использования). Кроме того, закрывать подключение всегда безопасно, когда его состоянием в текущий момент является `ConnectionState.Closed`.

Работа с объектами `ConnectionStringBuilder`

Работа со строками подключения в коде может быть утомительной, т.к. они часто представлены в виде строковых литералов, которые в лучшем случае трудно обрабатывать и контролировать на наличие ошибок. Предлагаемые Microsoft поставщики данных ADO.NET поддерживают *объекты построителей строк подключения*, которые позволяют устанавливать пары "имя-значение" с применением строго типизированных свойств. Взгляните на следующую модификацию текущего метода `Main()`:

```
static void Main(string[] args)
{
  WriteLine("***** Fun with Data Readers *****\n");

  // Создать строку подключения с помощью объекта построителя.
  var cnStringBuilder = new SqlConnectionStringBuilder
  {
    InitialCatalog = "AutoLot",
    DataSource = @"(local)\SQLEXPRESS2014",
    ConnectTimeout = 30,
    IntegratedSecurity = true
  };

  using(SqlConnection connection = new SqlConnection())
  {
    connection.ConnectionString = cnStringBuilder.ConnectionString;
    connection.Open();
    ShowConnectionStatus(connection);
    ...
  }
  ReadLine();
}
```

В этой версии метода `Main()` создается экземпляр класса `SqlConnectionStringBuilder`, соответствующим образом устанавливаются его свойства, после чего с использованием свойства `ConnectionString` получается внутренняя строка. Обратите внимание, что здесь применяется стандартный конструктор типа. При желании объект построителя строки подключения для поставщика данных можно также создать, передав в качестве отправной точки существующую строку подключения (это может быть удобно, когда значения динамически читаются из файла `App.config`). После наполнения объекта начальными строковыми данными отдельные пары "имя-значение" можно изменить с помощью связанных свойств, как демонстрируется ниже:

```
static void Main(string[] args)
{
  WriteLine("***** Fun with Data Readers *****\n");

  // Предположим, что значение connectionString
  // на самом деле получено из файла *.config.
  string connectionString = @"Data Source=(local)\SQLEXPRESS;" +
              "Integrated Security=SSPI;Initial Catalog=AutoLot";

  SqlConnectionStringBuilder cnStringBuilder =
    new SqlConnectionStringBuilder(connectionString);

  // Изменить значение таймаута.
  cnStringBuilder.ConnectTimeout = 5;
  ...
}
```

Работа с объектами команд

Теперь, когда вы лучше понимаете роль объекта подключения, следующей задачей будет выяснение, каким образом отправлять SQL-запросы базе данных. Тип `SqlCommand` (производный от `DbCommand`) — это объектно-ориентированное представление SQL-запроса, имени таблицы или хранимой процедуры. Тип команды указывается с использованием свойства `CommandType`, которое принимает любое значение из перечисления `CommandType`:

```
public enum CommandType
{
  StoredProcedure,
  TableDirect,
  Text  // Стандартное значение.
}
```

При создании объекта команды SQL-запрос можно указывать как параметр конструктора или напрямую через свойство `CommandText`. Кроме того, когда создается объект команды, необходимо задать желаемое подключение. Его также можно указать как параметр конструктора либо с применением свойства `Connection`. Взгляните на следующий фрагмент кода:

```
// Создать объект команды через аргументы конструктора.
string sql = "Select * From Inventory";
SqlCommand myCommand = new SqlCommand(sql, connection);

// Создать еще один объект команды посредством свойств.
SqlCommand testCommand = new SqlCommand();
testCommand.Connection = connection;
testCommand.CommandText = sql;
```

Учтите, что в этот момент вы еще не отправили SQL-запрос базе данных `AutoLot`, а только подготовили состояние объекта команды для будущего использования. В табл. 21.6 описаны некоторые дополнительные члены типа `DbCommand`.

Таблица 21.6. Члены типа `DbCommand`

Член	Описание
`CommandTimeout`	Получает или устанавливает время ожидания, пока не завершится попытка выполнить команду и сгенерируется ошибка. Стандартное значение составляет 30 секунд
`Connection`	Получает или устанавливает объект `DbConnection`, применяемый текущим объектом `DbCommand`
`Parameters`	Получает коллекцию типов `DbParameter`, используемых для параметризированного запроса
`Cancel()`	Отменяет выполнение команды
`ExecuteReader()`	Выполняет запрос SQL и возвращает объект `DbDataReader` поставщика данных, который предоставляет доступ к результату запроса в прямом направлении, допускающий только чтение
`ExecuteNonQuery()`	Выполняет оператор SQL, отличный от запроса (например, вставку, обновление, удаление или создание таблицы)
`ExecuteScalar()`	Легковесная версия метода `ExecuteReader()`, которая спроектирована специально для одноэлементных запросов (например, получение количества записей)
`Prepare()`	Создает подготовленную (или скомпилированную) версию команды для источника данных. Как вам может быть известно, *подготовленный запрос* выполняется несколько быстрее и удобен, когда один и тот же запрос необходимо выполнить многократно (обычно каждый раз с разными параметрами)

Работа с объектами чтения данных

После установления активного подключения и объекта команды SQL следующим действием будет отправка запроса источнику данных. Как вы наверняка догадались, это можно делать несколькими путями. Самый простой и быстрый способ получения информации из хранилища данных предлагает тип `DbDataReader` (реализующий интерфейс `IDataReader`). Вспомните, что объекты чтения данных представляют поток данных, допускающий только чтение в прямом направлении, который возвращает по одной записи за раз. Таким образом, объекты чтения данных полезны, только когда лежащему в основе хранилищу данных отправляются SQL-операторы выборки.

Объекты чтения данных удобны, если нужно быстро пройти по большому объему данных без необходимости иметь их представление в памяти. Например, в случае запрашивания 20 000 записей из таблицы с целью их сохранения в текстовом файле помещение этой информации в объект `DataSet` приведет к значительному расходу памяти (поскольку `DataSet` хранит полный результат запроса в памяти).

Более эффективный подход предусматривает создание объекта чтения данных, который максимально быстро проходит по всем записям. Тем не менее, имейте в виду, что объекты чтения данных (в отличие от объектов адаптеров данных, которые рассматриваются позже) поддерживают подключение к источнику данных открытым до тех пор, пока вы явно его не закроете.

Объекты чтения данных получаются из объекта команды с применением вызова `ExecuteReader()`. Объект чтения данных представляет текущую запись, прочитанную из базы данных. Он имеет метод индексатора (например, синтаксис `[]` в C#), который позволяет обращаться к столбцам текущей записи. Доступ к конкретному столбцу возможен либо по имени, либо по целочисленному индексу, начинающемуся с нуля.

В приведенном ниже примере использования объекта чтения данных задействован метод `Read()`, с помощью которого выясняется, когда достигнут конец записей (в случае чего он возвращает `false`). Для каждой прочитанной из базы данных записи с применением индексатора типа выводится модель, дружественное имя и цвет каждого автомобиля. Обратите внимание, что сразу после завершения обработки записей вызывается метод `Close()`, которые освобождает объект подключения.

```
static void Main(string[] args)
{
  ...
  // Получить объект чтения данных посредством ExecuteReader().
  using(SqlDataReader myDataReader = myCommand.ExecuteReader())
  {
    // Пройти в цикле по результатам.
    while (myDataReader.Read())
    {
      WriteLine($"-> Make: { myDataReader["Make"]},
   PetName: { myDataReader["PetName"]}, Color: { myDataReader["Color"]}.");
    }
  }
  ReadLine();
}
```

Индексатор объекта чтения данных перегружен для приема либо значения `string` (имя столбца), либо значения `int` (порядковый номер столбца). Таким образом, текущую логику объекта чтения можно сделать яснее (и избежать жестко закодированных строковых имен), сделав следующую модификацию (обратите внимание на использование свойства `FieldCount`):

```
while (myDataReader.Read())
{
  WriteLine("***** Record *****");
  for (int i = 0; i < myDataReader.FieldCount; i++)
  {
    WriteLine($"{myDataReader.GetName(i)} = { myDataReader.GetValue(i)} ");
  }
  WriteLine();
}
```

Если в настоящий момент скомпилировать проект и запустить на выполнение, то должен отобразиться список всех автомобилей из таблицы `Inventory` базы данных `AutoLot`. В следующем выводе показано несколько начальных записей:

```
***** Fun with Data Readers *****

***** Info about your connection *****
Database location: (local)\SQLEXPRESS2014
Database name: AutoLot
Timeout: 30
Connection state: Open

***** Record *****
CarId = 1
Make = VW
Color = Black
PetName = Zippy

***** Record *****
CarId = 2
Make = Ford
Color = Rust
PetName = Rusty
```

Получение множества результирующих наборов с использованием объекта чтения данных

Объекты чтения данных могут получать несколько результирующих наборов с применением одиночного объекта команды. Например, если вы хотите получить все строки из таблицы `Inventory`, а также все строки из таблицы `Customers`, то можете указать два SQL-оператора `Select`, разделив их точкой с запятой:

```
string sql = "Select * From Inventory;Select * from Customers";
```

После получения объекта чтения данных можно выполнить проход по каждому результирующему набору, используя метод `NextResult()`. Обратите внимание, что автоматически возвращается первый результирующий набор. Таким образом, если нужно прочитать все строки каждой таблицы, понадобится построить показанную ниже итерационную конструкцию:

```
do
{
  while (myDataReader.Read())
  {
    WriteLine("***** Record *****");
    for (int i = 0; i < myDataReader.FieldCount; i++)
    {
      WriteLine($"{myDataReader.GetName(i)} = {myDataReader.GetValue(i)}");
    }
    WriteLine();
  }
} while (myDataReader.NextResult());
```

К этому моменту вы уже должны лучше понимать функциональность, предлагаемую объектами чтения данных. Не забывайте, что объект чтения данных может обрабатывать только SQL-операторы `Select`; его нельзя применять для изменения существующей таблицы базы данных с использованием запросов `Insert`, `Update` или `Delete`. Модификация существующей базы данных требует дальнейшего исследования объектов команд.

Исходный код. Проект `AutoLotDataReader` доступен в подкаталоге `Chapter_21`.

Построение многократно используемой библиотеки доступа к данным

Метод `ExecuteReader()` извлекает объект чтения данных, который позволяет просматривать результаты SQL-оператора `Select` с помощью потока информации, допускающего только чтение в прямом направлении. Однако если требуется отправить операторы SQL, которые в итоге модифицируют таблицу данных (или любой другой отличный от запроса оператор SQL, такой как создание таблицы либо выдача разрешений), то понадобится вызов метода `ExecuteNonQuery()` объекта команды. Этот единственный метод выполняет вставки, обновления и удаления в зависимости от формата текста команды.

На заметку! Говоря формально, "отличный от запроса" означает оператор SQL, который не возвращает результирующий набор. Таким образом, операторы `Select` являются запросами, тогда как `Insert`, `Update` и `Delete` — нет. С учетом этого метод `ExecuteNonQuery()` возвращает значение `int`, которое представляет количество строк, затронутых операторами, а не новый набор записей.

Далее вы узнаете, как модифицировать содержимое существующей базы данных с применением только вызова метода `ExecuteNonQuery()`; вашей следующей целью будет построение специальной библиотеки доступа к данным, которая может инкапсулировать процесс работы с базой данных `AutoLot`. В реальной производственной среде логика, связанная с ADO.NET, почти всегда будет изолирована в .NET-сборке `.dll` по одной простой причине — возможность многократного использования кода. В начальных примерах главы это не делалось, чтобы не отвлекаться от решаемых задач. Тем не менее, написание *той же самой* логики подключения, *той же самой* логики чтения данных и *той же самой* логики команд для каждого приложения, которому необходимо взаимодействовать с базой данных `AutoLot`, было бы бесполезной тратой времени.

Изоляция логики доступа к данным в библиотеке кода .NET означает, что множество приложений с любым пользовательским интерфейсом (например, консольным, настольным или в веб-стиле) могут ссылаться на такую библиотеку в независимой от языка манере. Таким образом, если вы создали библиотеку доступа к данным на C#, то другие разработчики могут строить пользовательские интерфейсы на любом языке .NET по своему выбору.

В этой главе библиотека доступа к данным (`AutoLotDAL.dll`) будет содержать единственное пространство имен (`AutoLotConnectedLayer`), которое взаимодействует с базой данных `AutoLot` с применением подключенных типов ADO.NET. В следующей главе в библиотеку будет добавлено новое пространство имен (`AutoLotDisconnectionLayer`), которое содержит типы для взаимодействия с базой данных `AutoLot`, используя автономный уровень. В главе 23 будет создан завершенный уровень доступа к данным с применением Entity Framework. В оставшихся главах книги библиотека `AutoLotDAL.dll` будет неоднократно использоваться в различных приложениях.

Начнем с создания нового проекта библиотеки классов C# по имени AutoLotDAL (сокращение от *AutoLot Data Access Layer* — уровень доступа к данным AutoLot) и удалим стандартный файл класса. Добавим новую папку с помощью пункта меню Project⇨New Folder (Проект⇨Новая папка), удостоверившись в том, что в окне Solution Explorer выбран узел Project (Проект)), и назовем ее ConnectedLayer. Затем добавим в новую папку класс по имени InventoryDAL.cs и изменим его на public. Этот класс будет определять разнообразные члены, предназначенные для взаимодействия с таблицей Inventory базы данных AutoLot. Наконец, импортируем следующие пространства имен .NET:

```
using System;

// Будет использоваться поставщик SQL Server; однако
// для обеспечения более высокой гибкости разрешено
// также применять шаблон фабрики поставщиков ADO.NET.
using System.Data;
using System.Data.SqlClient;
using System.Collections.Generic;

namespace AutoLotDAL.ConnectedLayer
{
  public class InventoryDAL
  {
  }
}
```

На заметку! Вы можете вспомнить из главы 13, что когда объекты используют типы, управляющие низкоуровневыми ресурсами (например, подключением к базе данных), рекомендуется реализовать интерфейс IDisposable и написать подходящий финализатор. В производственной среде классы, подобные InventoryDAL, делают то же самое, но здесь мы поступать так не будем, чтобы сосредоточиться на особенностях инфраструктуры ADO.NET.

Добавление логики подключения

Первая задача, которую понадобится решить, связана с определением ряда методов, которые позволят вызывающему коду подключаться и отключаться от источника данных с применением допустимой строки подключения. Поскольку в сборке AutoLotDAL.dll будет жестко закодирована работа с типами из пространства имен System.Data.SqlClient, необходимо определить закрытую переменную-член типа SqlConnection, которая будет размещаться во время создания объекта InventoryDAL. Вдобавок мы определим методы OpenConnection() и CloseConnection() для взаимодействия с этой переменной-членом.

```
public class InventoryDAL
{
  // Этот член будет использоваться всеми методами.
  private SqlConnection _sqlConnection = null;

  public void OpenConnection(string connectionString)
  {
    _sqlConnection = new SqlConnection {ConnectionString = connectionString};
    _sqlConnection.Open();
  }
  public void CloseConnection()
  {
    _sqlConnection.Close();
  }
}
```

Ради краткости внутри типа InventoryDAL не будет предприниматься проверка на предмет возникновения возможных исключений, равно как не будут генерироваться специальные исключения в разнообразных обстоятельствах (скажем, из-за неправильно сформированной строки подключения). Если же строится библиотека доступа к данным производственного уровня, то определенно понадобится использовать приемы структурированной обработки исключений, чтобы во время выполнения учесть любые аномалии.

Добавление логики вставки

Вставка новой записи в таблицу Inventory сводится к построению SQL-оператора Insert (на основе пользовательского ввода) и вызову метода ExecuteNonQuery() с применением объекта команды. Это можно увидеть в действии, добавив к типу InventoryDAL открытый метод по имени InsertAuto(), который принимает четыре параметра, отображаемые на четыре столбца таблицы Inventory (CarId, Color, Make и PetName). Упомянутые аргументы используются при форматировании строки для вставки новой записи. И, наконец, для выполнения полученного оператора SQL применяется объект SqlConnection.

```
public void InsertAuto(int id, string color, string make, string petName)
{
  // Сформатировать и выполнить оператор SQL.
  string sql = "Insert Into Inventory" +
    $"(Make, Color, PetName) Values ('{make}', '{color}', '{petName}')";
  // Выполнить, используя наше подключение.
  using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
  {
    command.ExecuteNonQuery();
  }
}
```

Синтаксически этот метод нормален, но для него можно было бы предусмотреть перегруженную версию, которая позволила бы вызывающему коду передавать объект строго типизированного класса, который представляет данные для новой строки. Добавим в проект новую папку по имени Models, а в ней определим новый открытый класс под названием NewCar, представляющий новую строку в таблице Inventory:

```
public class NewCar
{
  public int CarId { get; set; }
  public string Color { get; set; }
  public string Make { get; set; }
  public string PetName { get; set; }
}
```

Теперь добавим в класс InventoryDAL следующую версию метода InsertAuto(), не забыв об операторе using для пространства имен AutoLotDAL.Models:

```
public void InsertAuto(NewCar car)
{
  // Сформатировать и выполнить оператор SQL.
  string sql = "Insert Into Inventory" +
    "( Make, Color, PetName) Values" +
    $"('{car.Make}', '{car.Color}', '{car.PetName}')";
  // Выполнить, используя наше подключение.
  using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
  {
    command.ExecuteNonQuery();
  }
}
```

Определение классов, представляющих записи в реляционной базе данных, является распространенным способом построения библиотеки доступа к данным. На самом деле, как вы увидите в главе 23, инфраструктура ADO.NET Entity Framework может автоматически генерировать строго типизированные классы, которые позволяют взаимодействовать с базой данных. В качестве связанного замечания: *автономный уровень* ADO.NET (рассматриваемый в главе 22) генерирует строго типизированные объекты `DataSet` для представления данных из отдельно взятой таблицы в реляционной базе данных.

На заметку! Вполне вероятно вам известно, что построение оператора SQL с применением конкатенации строк может быть рискованным с точки зрения безопасности (вспомните об атаках внедрением в SQL). Текст команды предпочтительнее формировать с использованием параметризированного запроса, с которым вы вскоре ознакомитесь.

Добавление логики удаления

Удаление существующей записи не сложнее вставки новой записи. В отличие от метода `InsertAuto()` на этот раз вы узнаете о важном блоке `try/catch`, который обрабатывает возможную попытку удалить запись об автомобиле, уже заказанном кем-то из таблицы `Customers`. Добавьте в класс `InventoryDAL` следующий метод:

```
public void DeleteCar(int id)
{
  // Удалить запись об автомобиле с указанным CarId.
  string sql = $"Delete from Inventory where CarId = '{id}'";
  using(SqlCommand command = new SqlCommand(sql, _sqlConnection))
  {
    try
    {
      command.ExecuteNonQuery();
    }
    catch(SqlException ex)
    {
      Exception error = new Exception("Sorry! That car is on order!", ex);
                                              // Этот автомобиль заказан
      throw error;
    }
  }
}
```

Добавление логики обновления

Когда речь идет об обновлении существующей записи в таблице `Inventory`, первым делом потребуется решить, какие характеристики позволить изменять вызывающему коду: цвет автомобиля, его дружественное имя, модель или все перечисленное? Один из способов предоставления вызывающему коду полной гибкости заключается в определении метода, принимающего параметр типа `string`, который представляет любой оператор SQL, но это в лучшем случае рискованно.

В идеале лучше иметь набор методов, которые позволяют вызывающему коду обновлять запись разнообразными способами. Тем не менее, для этой простой библиотеки доступа к данным мы определим единственный метод, который дает вызывающему коду возможность обновить дружественное имя указанного автомобиля:

```
public void UpdateCarPetName(int id, string newPetName)
{
  // Обновить PetName в записи об автомобиле с указанным CarId.
  string sql = $"Update Inventory Set PetName = '{newPetName}' Where CarId = '{id}'";
  using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
  {
    command.ExecuteNonQuery();
  }
}
```

Добавление логики выборки

Теперь необходимо добавить метод выборки. Как было показано ранее в главе, объект чтения данных поставщика позволяет получать выборку записей с применением курсора, допускающего только чтение в прямом направлении. Вызвав метод `Read()`, можно обработать каждую запись подходящим образом. Однако при этом придется решить проблему с возвращением записей вызывающему уровню приложения.

Один из подходов предусматривает наполнение и возврат многомерного массива (или другого подобного возвращаемого значения, такого как обобщенный объект `List<NewCar>`) с данными, полученными с помощью метода `Read()`. Ниже демонстрируется получение данных из таблицы `Inventory` с использованием этого подхода:

```
public List<NewCar> GetAllInventoryAsList()
{
  // Здесь будут храниться записи.
  List<NewCar> inv = new List<NewCar>();

  // Подготовить объект команды.
  string sql = "Select * From Inventory";
  using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
  {
    SqlDataReader dataReader = command.ExecuteReader();
    while (dataReader.Read())
    {
      inv.Add(new NewCar
      {
        CarId = (int)dataReader["CarId"],
        Color = (string)dataReader["Color"],
        Make = (string)dataReader["Make"],
        PetName = (string)dataReader["PetName"]
      });
    }
    dataReader.Close();
  }
  return inv;
}
```

Еще один подход предполагает возвращение объекта `System.Data.DataTable`, который в действительности является частью автономного уровня ADO.NET. Полное описание автономного уровня вы найдете в следующей главе, а пока достаточно знать, что `DataTable` — это класс, который представляет табличный блок данных (например, сетку электронной таблицы).

Внутренне класс `DataTable` представляет данные в виде коллекции строк и столбцов. Хотя такую коллекцию можно заполнять программно, в типе `DataTable` имеется метод `Load()`, который автоматически наполняет ее с применением объекта чтения данных.

Взгляните на следующие методы, которые возвращают данные из таблицы Inventory в виде объекта DataTable:

```
public DataTable GetAllInventoryAsDataTable()
{
  // Здесь будут храниться записи.
  DataTable dataTable = new DataTable();

  // Подготовить объект команды.
  string sql = "Select * From Inventory";
  using (SqlCommand cmd = new SqlCommand(sql, _sqlConnection))
  {
    SqlDataReader dataReader = cmd.ExecuteReader();
    // Заполнить DataTable данными из объекта чтения и выполнить очистку.
    dataTable.Load(dataReader);
    dataReader.Close();
  }
  return dataTable;
}
```

Работа с параметризированными объектами команд

В настоящий момент в логике вставки, обновления и удаления для типа InventoryDAL используются жестко закодированные строковые литералы для каждого запроса SQL. Скорее всего, вы знаете о существовании *параметризированных запросов*, которые позволяют трактовать параметры SQL как объекты, а не простые порции текста. Обращение с запросами SQL в более объектно-ориентированной манере помогает сократить количество опечаток (учитывая строго типизированные свойства). Вдобавок параметризированные запросы обычно выполняются значительно быстрее запросов в виде строковых литералов, т.к. они подвергаются разбору только однажды (а не каждый раз, когда строка с запросом SQL присваивается свойству CommandText). Параметризированные запросы также содействуют в защите против атак внедрением в SQL (хорошо известная проблема безопасности доступа к данным).

Для поддержки параметризированных запросов объекты команд ADO.NET содержат коллекцию индивидуальных объектов параметров. По умолчанию эта коллекция пуста, но в нее можно вставить любое количество объектов параметров, которые отображаются на *параметры-заполнители* в запросе SQL. Чтобы ассоциировать параметр внутри запроса SQL с членом коллекции параметров в объекте команды, параметр запроса SQL необходимо снабдить префиксом в виде символа @ (во всяком случае, когда применяется Microsoft SQL Server; не все СУБД поддерживают такую систему обозначений).

Указание параметров с использованием типа DbParameter

Перед построением параметризированного запроса вы должны ознакомиться с типом DbParameter (который является базовым классом для объекта параметра поставщика). Класс DbParameter поддерживает несколько свойств, которые позволяют конфигурировать имя, размер и тип параметра, а также другие характеристики, включая направление движения параметра. Некоторые основные свойства типа DbParameter описаны в табл. 21.7.

Давайте теперь посмотрим, как заполнять коллекцию совместимых с DBParameter объектов, содержащуюся в объекте команды, для чего переделаем метод InsertAuto() так, чтобы в нем были задействованы объекты параметров (аналогичным образом можно переписать остальные методы, но в настоящем примере это не обязательно):

Таблица 21.7. Основные свойства типа `DbParameter`

Свойство	Описание
`DbType`	Получает или устанавливает собственный тип данных параметра, представленный как тип данных CLR
`Direction`	Получает или устанавливает направление движения параметра: только для ввода, только для вывода, для ввода и для вывода или для возвращаемого значения
`IsNullable`	Получает или устанавливает признак, может ли параметр принимать значения `null`
`ParameterName`	Получает или устанавливает имя `DbParameter`
`Size`	Получает или устанавливает максимальный размер данных в байтах для параметра (полезно только для текстовых данных)
`Value`	Получает или устанавливает значение параметра

```
public void InsertAuto(int id, string color, string make, string petName)
{
  // Обратите внимание на "заполнители" в запросе SQL.
  string sql = "Insert Into Inventory" +
               "(Make, Color, PetName) Values" +
               "(@Make, @Color, @PetName)";
  // Эта команда будет иметь внутренние параметры.
  using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
  {
    // Заполнить коллекцию параметров.
    SqlParameter parameter = new SqlParameter
    {
      ParameterName = "@Make",
      Value = make,
      SqlDbType = SqlDbType.Char,
      Size = 10
    };
    command.Parameters.Add(parameter);
    parameter = new SqlParameter
    {
      ParameterName = "@Color",
      Value = color,
      SqlDbType = SqlDbType.Char,
      Size = 10
    };
    command.Parameters.Add(parameter);
    parameter = new SqlParameter
    {
      ParameterName = "@PetName",
      Value = petName,
      SqlDbType = SqlDbType.Char,
      Size = 10
    };
    command.Parameters.Add(parameter);
    command.ExecuteNonQuery();
  }
}
```

Снова обратите внимание, что наш запрос SQL содержит четыре символа-заполнителя, снабженные префиксами @. Тип SqlParameter применяется для сопоставления с каждым заполнителем за счет указания в свойстве ParameterName его имени и предоставления других деталей (например, значения, типа данных и размера) в строго типизированной манере. После того, как объект параметра готов, он добавляется в коллекцию объекта команды вызовом метода Add().

На заметку! В приведенном примере для установки объектов параметров используются различные свойства. Тем не менее, следует отметить, что объекты параметров поддерживают несколько перегруженных конструкторов, которые позволяют устанавливать значения разнообразных свойств (в результате давая более компактную кодовую базу). Также имейте в виду, что Visual Studio предлагает множество визуальных конструкторов, которые автоматически генерируют большой объем этого утомительного в написании кода работы с параметрами (см. главы 22 и 23).

В то время как построение параметризированного запроса часто требует большего объема кода, в результате получается более удобный способ для программной настройки операторов SQL и достигается лучшая производительность. Хотя такой прием можно применять для любого запроса SQL, параметризированные запросы удобнее всего, когда требуется запускать хранимые процедуры.

Выполнение хранимой процедуры

Вспомните, что *хранимая процедура* — это именованный блок кода SQL, сохраненный в базе данных. Хранимые процедуры можно конструировать так, чтобы они возвращали набор строк либо скалярных типов данных или выполняли еще какие-то осмысленные действия (например, вставку, обновление или удаление); в них также можно предусмотреть любое количество необязательных параметров. Конечным результатом будет единица работы, которая ведет себя подобно типичной функции, но только находится в хранилище данных, а не в двоичном бизнес-объекте. В текущий момент в базе данных AutoLot определена единственная хранимая процедура по имени GetPetName, имеющая такую форму:

```
GetPetName
@carID int,
@petName char(10) output
AS
SELECT @petName = PetName from Inventory where CarId = @carID
```

Теперь рассмотрим следующий финальный метод типа InventoryDAL, в котором вызывается эта хранимая процедура:

```
public string LookUpPetName(int carID)
{
  string carPetName;
  // Установить имя хранимой процедуры.
  using (SqlCommand command = new SqlCommand("GetPetName", _sqlConnection))
  {
    command.CommandType = CommandType.StoredProcedure;
    // Входной параметр.
    SqlParameter param = new SqlParameter
    {
      ParameterName = "@carID",
      SqlDbType = SqlDbType.Int,
      Value = carID,
      Direction = ParameterDirection.Input
    };
    command.Parameters.Add(param);
```

```
    // Выходной параметр.
    param = new SqlParameter
    {
      ParameterName = "@petName",
      SqlDbType = SqlDbType.Char,
      Size = 10,
      Direction = ParameterDirection.Output
    };
    command.Parameters.Add(param);

    // Выполнить хранимую процедуру.
    command.ExecuteNonQuery();

    // Возвратить выходной параметр.
    carPetName = (string)command.Parameters["@petName"].Value;
  }
  return carPetName;
}
```

С вызовом хранимых процедур связан один важный аспект: объект команды может представлять оператор SQL (по умолчанию) *либо* имя хранимой процедуры. Когда объекту команды необходимо сообщить о том, что он будет вызывать хранимую процедуру, потребуется указать имя этой процедуры (в аргументе конструктора или в свойстве `CommandText`) и установить свойство `CommandType` в `CommandType.StoredProcedure`. (Если этого не сделать, то возникнет исключение времени выполнения, т.к. по умолчанию объект команды ожидает оператор SQL.)

```
SqlCommand command = new SqlCommand("GetPetName", _sqlConnection);
command.CommandType = CommandType.StoredProcedure;
```

Далее обратите внимание, что свойство `Direction` объекта параметра позволяет указать направление движения каждого параметра, передаваемого хранимой процедуре (например, входной параметр, выходной параметр, входной/выходной параметр или возвращаемое значение). Как и ранее, все объекты параметров добавляются в коллекцию параметров объекта команды:

```
// Входной параметр.
SqlParameter param = new SqlParameter
{
  ParameterName = "@carID",
  SqlDbType = SqlDbType.Int,
  Value = carID,
  Direction = ParameterDirection.Input
};
command.Parameters.Add(param);
```

После того, как хранимая процедура, запущенная вызовом метода `ExecuteNonQuery()`, завершила работу, можно получить значение выходного параметра, просмотрев коллекцию параметров объекта команды и применив соответствующее приведение:

```
// Возвратить выходной параметр.
carPetName = (string)command.Parameters["@petName"].Value;
```

Итак, начальная версия библиотеки доступа к данным `AutoLotDAL.dll` готова. Эту сборку можно задействовать при построении пользовательского интерфейса любого вида для отображения и редактирования данных (например, в консольном, настольном или основанном на HTML веб-приложении). Вопросы создания графических пользовательских интерфейсов пока не обсуждались, поэтому протестируем полученную библиотеку доступа к данным с помощью нового консольного приложения.

Исходный код. Проект AutoLotDAL доступен в подкаталоге Chapter_21.

Создание приложения с консольным пользовательским интерфейсом

Создадим новый проект консольного приложения по имени AutoLotCUIClient. После создания проекта понадобится добавить ссылки на сборки AutoLotDAL.dll и System.Configuration.dll. При работе с загружаемым кодом примеров можно сослаться на проект AutoLotDAL внутри подкаталога Chapter_21. В случае создания нового решения для каждого примера необходимо перейти к решению AutoLotDAL и отыскать файл AutoLotDAL.dll в подкаталоге build. Затем добавим в файл кода C# следующие операторы using:

```
using AutoLotDAL.ConnectedLayer;
using AutoLotDAL.Models;
using System.Configuration;
using System.Data;
using static System.Console;
```

Откроем файл App.config в проекте (или добавим новый, если он еще не существует) и поместим в него элемент <connectionString>, который будет применяться для подключения к имеющемуся экземпляру базы данных AutoLot:

```
<configuration>
  <connectionStrings>
    <add name ="AutoLotSqlProvider" connectionString =
    "Data Source=(local)\SQLEXPRESS2014;
    Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </connectionStrings>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
  </startup>
</configuration>
```

Реализация метода Main()

Метод Main() предлагает пользователю выбрать специфическое действие и выполняет его с использованием оператора switch. Пользователь может вводить следующие команды:

- I — вставка новой записи в таблицу Inventory;
- U — обновление существующей записи в таблице Inventory;
- D — удаление существующей записи из таблицы Inventory;
- L — отображение текущего инвентарного списка автомобилей с применением объекта чтения данных;
- S — вывод пользователю списка допустимых команд;
- P — поиск дружественного имени автомобиля по его идентификатору;
- Q — завершение работы программы.

Каждая команда поддерживается специальным статическим методом внутри класса Program. Ниже приведена полная реализация метода Main(). Обратите внимание, что все методы, вызываемые в цикле do/while (за исключением ShowInstructions()), принимают в качестве единственного параметра объект InventoryDAL.

```
static void Main(string[] args)
{
  WriteLine("***** The AutoLot Console UI *****\n");
  // Получить строку подключения из файла App.config.
  string connectionString =
   ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"].ConnectionString;
  bool userDone = false;
  string userCommand = "";
  // Создать объект InventoryDAL.
  InventoryDAL invDAL = new InventoryDAL();
  invDAL.OpenConnection(connectionString);
  // Продолжать запрашивать у пользователя ввод вплоть до получения команды Q.
  try
  {
    ShowInstructions();
    do
    {
      Write("\nPlease enter your command: ");
      userCommand = ReadLine();
      WriteLine();
      switch (userCommand?.ToUpper()??"")
      {
      case "I":
        InsertNewCar(invDAL);
        break;
      case "U":
        UpdateCarPetName(invDAL);
        break;
      case "D":
        DeleteCar(invDAL);
        break;
      case "L":
        ListInventory(invDAL);
        break;
      case "S":
        ShowInstructions();
        break;
      case "P":
        LookUpPetName(invDAL);
        break;
      case "Q":
        userDone = true;
        break;
      default:
        WriteLine("Bad data! Try again");
        break;
      }
    } while (!userDone);
  }
  catch (Exception ex)
  {
    WriteLine(ex.Message);
  }
  finally
  {
    invDAL.CloseConnection();
  }
}
```

Реализация метода ShowInstructions()

Метод ShowInstructions() делает то, что и можно было ожидать — выводит доступные команды:

```
private static void ShowInstructions()
{
  WriteLine("I: Inserts a new car.");
            // Вставить новую запись об автомобиле.
  WriteLine("U: Updates an existing car.");
            // Обновить существующую запись об автомобиле.
  WriteLine("D: Deletes an existing car.");
            // Удалить существующую запись об автомобиле.
  WriteLine("L: Lists current inventory.");
            // Вывести текущий инвентарный список автомобилей.
  WriteLine("S: Shows these instructions.");
            // Вывести информацию об этих командах.
  WriteLine("P: Looks up pet name.");
            // Найти дружественное имя автомобиля.
  WriteLine("Q: Quits program.");
}
```

Реализация метода ListInventory()

Метод ListInventory() можно реализовать двумя способами, основываясь на том, как была сконструирована библиотека доступа к данным. Вспомните, что метод GetAllInventoryAsDataTable() класса InventoryDAL возвращает объект DataTable. Этот подход можно реализовать следующим образом:

```
private static void ListInventory(InventoryDAL invDAL)
{
  // Получить список автомобилей на складе.
  DataTable dt = invDAL.GetAllInventoryAsDataTable();
  // Передать объект DataTable во вспомогательную функцию для отображения.
  DisplayTable(dt);
}
```

Вспомогательный метод DisplayTable() отображает данные таблицы, используя свойства Rows и Columns входного объекта DataTable (объект DataTable подробно рассматривается в следующей главе, поэтому пока не обращайте внимания на детали):

```
private static void DisplayTable(DataTable dt)
{
  // Вывести имена столбцов.
  for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
  {
    Write($"{dt.Columns[curCol].ColumnName}\t");
  }
  WriteLine("\n-----------------------------------");
  // Вывести содержимое объекта DataTable.
  for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
  {
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write($"{dt.Rows[curRow][curCol]}\t");
    }
    WriteLine();
  }
}
```

Если вы предпочитаете вызывать метод `GetAllInventoryAsList()` класса `InventoryDAL`, то можно реализовать метод `ListInventoryViaList()`, как показано ниже (понадобится добавить оператор `using` для пространства имен `AutoLotDAL.Models`):

```
private static void ListInventoryViaList(InventoryDAL invDAL)
{
  // Получить список автомобилей на складе.
  List<NewCar> record = invDAL.GetAllInventoryAsList();
  WriteLine("CarId:\tMake:\tColor:\tPetName:");
  foreach (NewCar c in record)
  {
    WriteLine($"{c.CarId}\t{c.Make}\t{c.Color}\t{c.PetName}");
  }
}
```

Реализация метода `DeleteCar()`

Удаление существующей записи об автомобиле сводится просто к запросу у пользователя идентификатора автомобиля и его передаче методу `DeleteCar()` класса `InventoryDAL`:

```
private static void DeleteCar(InventoryDAL invDAL)
{
  // Получить идентификатор автомобиля, запись о котором должна быть удалена.
  Write("Enter ID of Car to delete: ");
  int id = int.Parse(ReadLine()??"0");
  // На случай нарушения ссылочной целостности.
  try
  {
    invDAL.DeleteCar(id);
  }
  catch(Exception ex)
  {
    WriteLine(ex.Message);
  }
}
```

Реализация метода `InsertNewCar()`

Для вставки новой записи в таблицу `Inventory` необходимо запросить у пользователя информацию о новом автомобиле (с применением вызовов `Console.ReadLine()`) и передать эти данные в метод `InsertAuto()` класса `InventoryDAL`:

```
private static void InsertNewCar(InventoryDAL invDAL)
{
  // Получить пользовательские данные.
  Write("Enter Car ID: ");
  var newCarId = int.Parse(ReadLine()??"0");     // Ввод идентификатора
                                                 // автомобиля
  Write("Enter Car Color: ");
  var newCarColor = ReadLine();    // Ввод цвета автомобиля
  Write("Enter Car Make: ");
  var newCarMake = ReadLine();     // Ввод модели автомобиля
  Write("Enter Pet Name: ");
  var newCarPetName = ReadLine();  // Ввод дружественного имени автомобиля
  // Передать информацию библиотеке доступа к данным.
  invDAL.InsertAuto(newCarId, newCarColor, newCarMake, newCarPetName);
}
```

Вспомните, что метод `InsertAuto()` был перегружен, чтобы вместо набора независимых аргументов принимать объект `NewCar`. Следовательно, метод `InsertNewCar()` можно было бы реализовать и так:

```
private static void InsertNewCar(InventoryDAL invDAL)
{
  // Получить пользовательские данные.
  // ...для краткости код не показан...
  // Передать информацию библиотеке доступа к данным.
  var c = new NewCar
  {
    CarId = newCarId,
    Color = newCarColor,
    Make = newCarMake,
    PetName = newCarPetName
  };
  invDAL.InsertAuto(c);
}
```

Реализация метода `UpdateCarPetName()`

Реализация метода `UpdateCarPetName()` выглядит похожей:

```
private static void UpdateCarPetName(InventoryDAL invDAL)
{
  // Получить пользовательские данные.
  Write("Enter Car ID: ");
  var carID = int.Parse(ReadLine()??"0");
  Write("Enter New Pet Name: ");
  var newCarPetName = ReadLine();

  // Передать информацию библиотеке доступа к данным.
  invDAL.UpdateCarPetName(carID, newCarPetName);
}
```

Реализация метода `LookUpPetName()`

Получение дружественного имени указанного автомобиля реализуется аналогично предыдущим методам; дело в том, что библиотека доступа к данным инкапсулирует все низкоуровневые вызовы ADO.NET:

```
private static void LookUpPetName(InventoryDAL invDAL)
{
  // Получить идентификатор автомобиля для поиска дружественного имени.
  Write("Enter ID of Car to look up: ");
  int id = int.Parse(ReadLine()??"0");
  WriteLine($"Petname of {id} is {invDAL.LookUpPetName(id).TrimEnd()}.");
}
```

На этом разработка приложения с консольным пользовательским интерфейсом завершена. Самое время запустить полученную программу и протестировать каждый метод. Вот частичный вывод с тестированием команд `L`, `P` и `Q`:

```
***** The AutoLot Console UI *****

I: Inserts a new car.
U: Updates an existing car.
D: Deletes an existing car.
L: Lists current inventory.
S: Shows these instructions.
```

```
P: Looks up pet name.
Q: Quits program.

Please enter your command: L

CarId:  Make:   Color:   PetName:
1       VW      Black    Zippy
2       Ford    Rust     Rusty
3       Saab    Black    Mel
4       Yugo    Yellow   Cluncker
5       BMW     Black    Bimmer
6       BMW     Green    Hank
7       BMW     Pink     Pinkey

Please enter your command: P

Enter ID of Car to look up: 6
Petname of 6 is Hank.

Please enter your command: Q

Press any key to continue . . .
```

Исходный код. Проект `AutoLotCUIClient` доступен в подкаталоге `Chapter_21`.

Понятие транзакций базы данных

Давайте завершим исследование подключенного уровня ADO.NET рассмотрением концепции транзакции базы данных. Выражаясь просто, *транзакция* — это набор операций в базе данных, которые должны быть либо *все* выполнены, либо *все* потерпеть неудачу как единая группа. Несложно предположить, что транзакции по-настоящему важны для обеспечения безопасности, достоверности и согласованности табличных данных.

Транзакции также важны в ситуациях, когда операция базы данных включает в себя взаимодействие с множеством таблиц или хранимых процедур (либо с комбинацией атомарных элементов базы данных). Классическим примером транзакции может служить процесс перевода денежных средств с одного банковского счета на другой. Например, если вам понадобилось перевести $500 с депозитного счета на текущий чековый счет, то следующие шаги должны быть выполнены в транзакционной манере:

1. Банк должен снять $500 с вашего депозитного счета.
2. Банк должен добавить $500 на ваш текущий чековый счет.

Вряд ли бы вам понравилось, если бы деньги были сняты с депозитного счета, но не переведены (из-за какой-то ошибки со стороны банка) на текущий чековый счет, потому что вы попросту лишились бы $500. Однако если поместить эти шаги внутрь транзакции базы данных, то СУБД гарантирует, что все взаимосвязанные шаги будут выполнены как единое целое. Если любая часть транзакции откажет, будет произведен *откат* всей операции в исходное состояние. С другой стороны, если все шаги выполняются успешно, то транзакция будет *зафиксирована*.

На заметку! Из литературы, посвященной транзакциям, вам может быть известно сокращение ACID. Оно обозначает четыре ключевых характеристики транзакции: *атомарность* (Atomic; все или ничего), *согласованность* (Consistent; данные остаются устойчивыми на протяжении транзакции), *изоляция* (Isolated; транзакции не влияют друг на друга) и *постоянство* (Durable; транзакции сохраняются и протоколируются в журнале).

Платформа .NET поддерживает транзакции различными способами. Здесь мы рассмотрим объект транзакции поставщика данных ADO.NET (`SqlTransaction` в случае `System.Data.SqlClient`). Библиотеки базовых классов ADO.NET также предоставляют поддержку транзакций внутри многочисленных API-интерфейсов, включая перечисленные ниже.

- `System.EnterpriseServices`. Это пространство имен (находящееся в сборке `System.EnterpriseServices.dll`) содержит типы, которые позволяют интегрироваться с уровнем исполняющей среды COM+, в том числе с ее поддержкой распределенных транзакций.
- `System.Transactions`. Это пространство имен (находящееся в сборке `System.Transactions.dll`) содержит классы, позволяющие писать собственные транзакционные приложения и диспетчеры ресурсов для разнообразных служб (скажем, MSMQ, ADO.NET и COM+).
- *Windows Communication Foundation (WCF).* API-интерфейс WCF предоставляет службы для упрощения организации транзакций с помощью различных классов распределенной привязки.
- *Windows Workflow Foundations (WF).* API-интерфейс WF предлагает транзакционную поддержку для действий рабочих потоков.

В дополнение к готовой поддержке транзакций внутри библиотек базовых классов .NET можно также использовать язык SQL имеющейся СУБД. Например, вы могли бы написать хранимую процедуру, в которой применяются операторы `BEGIN TRANSACTION`, `ROLLBACK` и `COMMIT`.

Основные члены объекта транзакции ADO.NET

Несмотря на то что транзакционные типы существуют повсюду в библиотеках базовых классов, мы сосредоточим внимание на объектах транзакций, находящихся в поставщике данных ADO.NET; все они являются производными от `DBTransaction` и реализуют интерфейс `IDbTransaction`. Как было показано в начале главы, интерфейс `IDbTransaction` определяет несколько членов:

```
public interface IDbTransaction : IDisposable
{
  IDbConnection Connection { get; }
  IsolationLevel IsolationLevel { get; }
  void Commit();
  void Rollback();
}
```

Обратите внимание на свойство `Connection`, возвращающее ссылку на объект подключения, который инициировал текущую транзакцию (как вы вскоре увидите, объект транзакции получается из заданного объекта подключения). Метод `Commit()` вызывается, если все операции в базе данных завершились успешно. Это приводит к сохранению в хранилище данных всех ожидающих изменений. И наоборот, метод `Rollback()` можно вызвать в случае генерации исключения времени выполнения, что информирует СУБД о необходимости проигнорировать все ожидающие изменения и оставить первоначальные данные незатронутыми.

На заметку! Свойство `IsolationLevel` объекта транзакции позволяет указать, насколько активно транзакция должна защищаться от действий со стороны параллельно выполняющихся транзакций. По умолчанию транзакции полностью изолируются вплоть до их фиксации. Подробное описание значений перечисления `IsolationLevel` ищите в документации .NET Framework 4.6 SDK.

Помимо членов, определенных в интерфейсе `IDbTransaction`, тип `SqlTransaction` определяет дополнительный член под названием `Save()`, который предназначен для определения *точек сохранения*. Такая концепция позволяет откатить отказавшую транзакцию до именованной точки вместо того, чтобы осуществлять откат всей транзакции. При вызове метода `Save()` с использованием объекта `SqlTransaction` можно задавать удобный строковый псевдоним. А при вызове `Rollback()` этот псевдоним можно указать в качестве аргумента для выполнения *частичного отката*. Вызов `Rollback()` без аргументов приводит к отмене всех ожидающих изменений.

Добавление таблицы `CreditRisks` в базу данных `AutoLot`

Давайте теперь посмотрим, как применять транзакции ADO.NET. Начнем с использования окна Server Explorer в Visual Studio для добавления в базу данных `AutoLot` новой таблицы по имени `CreditRisks`, которая содержит такие же столбцы, как и созданная ранее таблица `Customers`: `CustId` (первичный ключ), `FirstName` и `LastName`. Таблица `CreditRisks` предназначена для отсеивания нежелательных клиентов с плохой кредитной историей (рис. 21.15).

Рис. 21.15. Таблица `CreditRisks`

Как и в предыдущем примере, где переводились деньги с депозитного на текущий чековый счет, перемещение рискованного клиента из таблицы `Customers` в таблицу `CreditRisks` должно происходить под недремлющим оком транзакционного контекста (в конце концов, вы хотите запомнить имена некредитоспособных клиентов). В частности, вам необходимо гарантировать, что *либо* текущие кредитные риски будут успешно удалены из таблицы `Customers` и добавлены в таблицу `CreditRisks`, *либо* ни одна из этих операций базы данных не выполнится.

На заметку! В производственной среде вам не придется строить совершенно новую таблицу базы данных для хранения рискованных клиентов; взамен в существующей таблице `Customers` можно предусмотреть булевский столбец по имени `IsCreditRisk`. Тем не менее, наличие этой новой таблицы даст возможность поработать с простой транзакцией.

Добавление метода транзакции в `InventoryDAL`

Давайте посмотрим, как работать с транзакциями ADO.NET программным образом. Для начала откроем созданный ранее проект библиотеки кода `AutoLotDAL` и добавим в класс `InventoryDAL` новый открытый метод по имени `processCreditRisk()`, предназначенный для работы с кредитными рисками. (Обратите внимание, что ради простоты в этом примере не применяется параметризированный запрос, но в аналогичном методе производственного уровня он должен быть задействован.)

```
// Новый член класса InventoryDAL.
public void ProcessCreditRisk(bool throwEx, int custID)
{
```

```
// Первым делом найти текущее имя по идентификатору клиента.
string fName;
string lName;
var cmdSelect =
  new SqlCommand($"Select * from Customers where CustId = {custID}",
  _sqlConnection);
using (var dataReader = cmdSelect.ExecuteReader())
{
  if (dataReader.HasRows)
  {
    dataReader.Read();
    fName = (string) dataReader["FirstName"];
    lName = (string) dataReader["LastName"];
  }
  else
  {
    return;
  }
}
// Создать объекты команд, которые представляют каждый шаг операции.
var cmdRemove =
  new SqlCommand($"Delete from Customers where CustId = {custID}",
  _sqlConnection);
var cmdInsert =
  new SqlCommand("Insert Into CreditRisks" +
  $"(FirstName, LastName) Values('{fName}', '{lName}')",
  _sqlConnection);
// Это будет получено из объекта подключения.
SqlTransaction tx = null;
try
{
  tx = _sqlConnection.BeginTransaction();
  // Включить команды в транзакцию.
  cmdInsert.Transaction = tx;
  cmdRemove.Transaction = tx;
  // Выполнить команды.
  cmdInsert.ExecuteNonQuery();
  cmdRemove.ExecuteNonQuery();
  // Эмулировать ошибку.
  if (throwEx)
  {
    throw new Exception("Sorry!  Database error! Tx failed...");
  }
  // Зафиксировать транзакцию.
  tx.Commit();
}
catch (Exception ex)
{
  WriteLine(ex.Message);
  // Любая ошибка приведет к откату транзакции.
  // Использование новой условной операции для проверки на предмет null
  tx?.Rollback();
}
}
```

Здесь используется входной параметр типа `bool`, который указывает, нужно ли генерировать произвольное исключение при попытке обработки проблемного клиента. Такой прием позволяет эмулировать непредвиденные обстоятельства, которые могут привести к неудачному завершению транзакции. Понятно, что это делается лишь в демонстрационных целях; настоящий метод транзакции не должен позволять вызывающему процессу нарушать работу логики по своему усмотрению!

Обратите внимание на применение двух объектов `SqlCommand` для представления каждого шага транзакции, которая будет запущена. После получения имени и фамилии клиента на основе входного параметра `custId` с помощью метода `BeginTransaction()` объекта подключения можно получить допустимый объект `SqlTransaction`. Затем (и это очень важно) потребуется *включить каждый объект команды*, присвоив его свойству `Transaction` только что полученного объекта транзакции. Если этого не сделать, то логика вставки и удаления не будет находиться в транзакционном контексте.

После вызова метода `ExecuteNonQuery()` на каждой команде генерируется исключение, если (и только если) значение параметра `bool` равно `true`. В таком случае происходит откат всех ожидающих операций базы данных. Если исключение не было сгенерировано, то оба шага будут зафиксированы в таблицах базы данных в результате вызова `Commit()`. Теперь нужно скомпилировать модифицированный проект `AutoLotDAL`, чтобы удостовериться в отсутствии ошибок.

Тестирование транзакции базы данных

Можно было бы обновить ранее построенное приложение `AutoLotCUIClient`, оснастив его возможностью вызова метода `ProcessCreditRisk()`, но взамен мы создадим новый проект консольного приложения под названием `AdoNetTransaction`. Добавим в него ссылку на сборку `AutoLotDAL.dll` и импортируем пространства имен `AutoLotDAL.ConnectedLayer` и `AutoLotDAL.Models`, а также включим оператор `using static System.Console`.

После этого откроем таблицу `Customers` для ввода данных, щелкнув правой кнопкой мыши на значке таблицы в окне Server Explorer и выбрав в контекстном меню пункт Show Table Data (Показать данные таблицы). Теперь добавим новую запись о клиенте с плохой кредитной историей (запомните идентификатор, назначенный новой записи):

- `FirstName`: `Homer`
- `LastName`: `Simpson`

Наконец, модифицируем метод `Main()` следующим образом:

```
static void Main(string[] args)
{
  WriteLine("***** Simple Transaction Example *****\n");

  // Простой способ позволить транзакции успешно выполниться или отказать.
  bool throwEx = true;

  Write("Do you want to throw an exception (Y or N): ");
      // Генерировать ли исключение?
  var userAnswer = ReadLine();
  if (userAnswer?.ToLower() == "n")
  {
    throwEx = false;
  }

  var dal = new InventoryDAL();
  dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS2014;
   Integrated Security=SSPI;" + "Initial Catalog=AutoLot");
```

```
    // Обработать клиента с идентификатором 5 - в следующей строке кода
    // должен быть указан идентификатор записи для клиента Homer Simpson.
    dal.ProcessCreditRisk(throwEx, 5);
    WriteLine("Check CreditRisk table for results");
    ReadLine();
  }
```

Если запустить программу и указать на необходимость генерации исключения, то запись о клиенте Homer Simpson *не* будет удалена из таблицы `Customers`, т.к. произойдет откат всей транзакции. Однако если исключение не генерируется, то окажется, что клиент с идентификатором 5 больше не находится в таблице `Customers`, а перемещен в таблицу `CreditRisks`.

Исходный код. Проект `AdoNetTransaction` доступен в подкаталоге `Chapter_21`.

Резюме

Инфраструктура ADO.NET — это собственная технология доступа к данным платформы .NET, которую можно использовать тремя отдельными способами: подключенным, автономным и через Entity Framework. В настоящей главе исследовался подключенный уровень, а также объяснялась роль поставщиков данных, которые по существу являются конкретными реализациями нескольких абстрактных базовых классов (из пространства имен `System.Data.Common`) и интерфейсных типов (из пространства имен `System.Data`). Вы увидели, что с применением модели фабрики поставщиков данных ADO.NET можно построить кодовую базу, не зависящую от поставщика.

Вы также узнали, что с помощью объектов подключений, объектов транзакций, объектов команд и объектов чтения данных из подключенного уровня можно выбирать, обновлять, вставлять и удалять записи. Кроме того, было показано, что объекты команд поддерживают внутреннюю коллекцию параметров, которые можно использовать для обеспечения безопасности к типам в запросах SQL; эта коллекция также удобна при запуске хранимых процедур.

ГЛАВА 22

ADO.NET, часть II: автономный уровень

В предыдущей главе рассматривался подключенный уровень и фундаментальные компоненты ADO.NET, которые позволяют отправлять в базу данных операторы SQL с использованием объекта подключения, объектов команд и объекта чтения данных выбранного поставщика данных. В настоящей главе вы ознакомитесь с *автономным уровнем* ADO.NET. Эта грань ADO.NET позволяет моделировать данные из базы в памяти, внутри вызывающего уровня, за счет применения многочисленных членов из пространства имен `System.Data` (в особенности `DataSet`, `DataTable`, `DataRow`, `DataColumn`, `DataView` и `DataRelation`). В таком случае можно обеспечить иллюзию того, что вызывающий уровень постоянно подключен к внешнему источнику данных, хотя в реальности все операции происходят с локальной копией реляционных данных.

На заметку! Как упоминалось во введении к предыдущей главе, инфраструктура Entity Framework (EF) набирает обороты и наблюдается ее растущее внедрение. Инфраструктура EF будет раскрыта в следующей главе, но важно понимать, каким образом работает ADO.NET, т.к. EF (и другие средства объектно-реляционного отображения .NET) построена поверх ADO.NET. Хотя этот *автономный* аспект ADO.NET возможно использовать даже без подключения к реляционной базе данных, чаще всего вы будете получать заполненные объекты `DataSet` с помощью объекта адаптера данных своего поставщика данных. Вы увидите, что объекты адаптеров данных функционируют в качестве моста между клиентским уровнем и реляционной базой данных. С применением этих объектов можно получать объекты `DataSet`, манипулировать их содержимым и отправлять модифицированные строки обратно. Конечным результатом будет в высшей степени масштабируемое приложение .NET, ориентированное на обработку данных.

В главе также будут продемонстрированы некоторые приемы привязки данных с использованием контекста настольного приложения, имеющего графический пользовательский интерфейс Windows Forms, и раскрыта роль *строго типизированного* объекта `DataSet`. Мы обновим созданную в главе 21 библиотеку доступа к данным `AutoLotDAL.dll`, добавив новое пространство имен, в котором задействован автономный уровень ADO.NET. В завершение главы мы обсудим технологию LINQ to DataSet, которая позволяет применять запросы LINQ к находящемуся в памяти кешу данных.

На заметку! Вы узнаете о разнообразных технологиях привязки данных для приложений Windows Presentation Foundation и ASP.NET позже в этой книге.

Понятие автономного уровня ADO.NET

В предыдущей главе вы видели, что работа с подключенным уровнем позволяет взаимодействовать с базой данных с использованием первичных объектов подключения, команд и чтения данных. Этот небольшой набор классов можно применять для выборки, вставки, обновления и удаления записей из основного содержимого (а также вызывать хранимые процедуры или выполнять другие операции над данными (например, DDL для создания таблицы и DCL для выдачи разрешений)). Однако была показана только часть истории ADO.NET. Вспомните, что объектную модель ADO.NET можно использовать также в автономной манере.

С применением автономного уровня появляется возможность моделирования реляционных данных с помощью объектной модели, расположенной в памяти. Выходя далеко за пределы простого моделирования табличных блоков строк и столбцов, типы из пространства имен `System.Data` позволяют представлять отношения между таблицами, ограничения столбцов, первичные ключи, представления и другие примитивы баз данных. После построения модели данных к ней можно применять фильтры, отправлять запросы в памяти и сохранять (или загружать) данные в формате XML и в двоичном формате. Все это можно делать, даже не подключаясь к СУБД (отсюда и название *автономный уровень*), а загружая данные из локального файла XML или вручную создавая объект `DataSet` в коде.

На заметку! В главе 23 будет исследоваться инфраструктура ADO.NET Entity Framework, которая построена на описанных здесь концепциях автономного уровня.

Автономные типы можно было бы использовать, не подключаясь к базе данных, но обычно вы по-прежнему будете применять объекты подключений и команд. Вдобавок для выборки и обновления данных будет задействован специфичный объект — *адаптер данных* (тип которого расширяет абстрактный класс `DbDataAdapter`). В отличие от подключенного уровня, данные, полученные посредством адаптера данных, не обрабатываются с использованием объектов чтения данных. Вместо этого объекты адаптеров перемещают данные между вызывающим кодом и источником данных с применением объектов `DataSet` (точнее — объектов `DataTable` в `DataSet`). Тип `DataSet` представляет собой контейнер для любого количества объектов `DataTable`, каждый из которых содержит коллекцию объектов `DataRow` и `DataColumn`.

Объект адаптера данных вашего поставщика данных автоматически обслуживает подключение к базе данных. В стремлении увеличить масштабируемость адаптеры данных удерживают подключение открытым в течение минимально возможного времени. После того как вызывающий код получил объект `DataSet`, вызывающий уровень полностью отключается от базы данных и остается с локальной копией удаленных данных. Вызывающий код может вставлять, удалять или модифицировать строки в выбранном объекте `DataTable`, но физическая база данных не обновляется до тех пор, пока в вызывающем коде не будет явно передан объект `DataTable` из `DataSet` адаптеру данных для обновления. В сущности, объекты `DataSet` позволяют клиентам делать вид, что они постоянно подключены, хотя в действительности они оперируют с базой данных, находящейся в памяти (рис. 22.1).

Поскольку центральной частью автономного уровня является класс `DataSet`, то первой задачей главы будет объяснение способа манипулирования им вручную. После этого никаких проблем с обработкой содержимого `DataSet`, извлеченного из объекта адаптера данных, у вас не возникнет.

Рис. 22.1. Объекты адаптеров данных перемещают объекты `DataSet` на клиентский уровень и обратно

Роль объектов `DataSet`

Как отмечалось ранее, объект `DataSet` представляет реляционные данные в памяти. Более конкретно, `DataSet` — это класс, который внутренне поддерживает три строго типизированных коллекции (рис. 22.2).

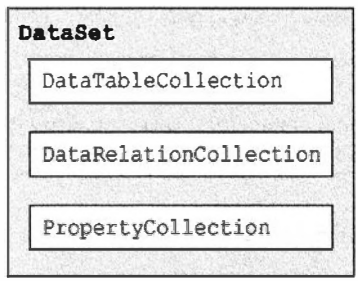


Рис. 22.2. Внутреннее устройство класса `DataSet`

Свойство `Tables` класса `DataSet` предоставляет доступ к коллекции `DataTableCollection`, которая содержит отдельные объекты `DataTable`. В `DataSet` присутствует еще одна важная коллекция — `DataRelationCollection`. Учитывая, что объект `DataSet` является автономной версией схемы базы данных, его можно использовать для программного представления отношений "родительский–дочерний" между таблицами. Например, с применением типа `DataRelation` можно создать отношение между двумя таблицами для моделирования ограничения внешнего ключа. Затем результирующий объект `DataRelation` можно добавить в коллекцию `DataRelationCollection` через свойство `Relations`. С этого момента при поиске данных можно перемещаться между связанными таблицами. Вы увидите, как это делать, далее в главе.

Свойство `ExtendedProperties` предоставляет доступ к объекту `PropertyCollection`, который позволяет ассоциировать с `DataSet` любую дополнительную информацию в виде пар "имя-значение". Информация может быть совершенно произвольной, даже не имеющей отношения к самим данным. Например, с объектом `DataSet` можно связать название компании, которое затем будет выступать в качестве метаданных в памяти. Другими примерами расширенных свойств могут служить временные метки, зашифрованный пароль, который должен быть указан для доступа к содержимому `DataSet`, число, представляющее частоту обновления данных, и многое другое.

На заметку! Классы `DataTable` и `DataColumn` также поддерживают свойство `ExtendedProperties`.

Основные свойства класса DataSet

Прежде чем погрузиться во множество других деталей программирования, давайте взглянем на ряд основных членов класса DataSet. Помимо свойств Tables, Relations и ExtendedProperties класс DataSet определяет дополнительные полезные свойства, которые кратко описаны в табл. 22.1.

Таблица 22.1. Свойства класса DataSet

Свойство	Описание
CaseSensitive	Указывает, чувствительно ли к регистру сравнение строк в объектах DataTable. По умолчанию равно false (сравнение строк выполняется без учета регистра)
DataSetName	Представляет дружественное имя объекта DataSet. Обычно это значение передается в параметре конструктора
EnforceConstraints	Получает или устанавливает значение, которое определяет, соблюдаются ли правила ограничений при попытке выполнить любые операции обновления (по умолчанию равно true)
HasErrors	Получает значение, указывающее на то, есть ли ошибки в любой строке любого объекта DataTable в DataSet
RemotingFormat	Позволяет определить, каким образом объект DataSet должен сериализировать свое содержимое (в двоичном виде или по умолчанию в XML)

Основные методы класса DataSet

Методы класса DataSet работают в сочетании с некоторой функциональностью, предоставляемой упомянутыми выше свойствами. В дополнение к взаимодействию с потоками XML класс DataSet предлагает методы, позволяющие копировать содержимое DataSet, перемещаться между внутренними таблицами и устанавливать начальные и конечные точки пакета обновлений. Избранные основные методы описаны в табл. 22.2.

Таблица 22.2. Избранные основные методы класса DataSet

Метод	Описание
AcceptChanges()	Фиксирует все изменения, внесенные в текущий объект DataSet с момента его загрузки или последнего вызова метода AcceptChanges()
Clear()	Полностью очищает данные DataSet, удаляя все строки из каждого объекта DataTable
Clone()	Клонирует структуру, но не данные DataSet, включая все объекты DataTable, а также все отношения и любые ограничения
Copy()	Копирует структуру и данные текущего объекта DataSet
GetChanges()	Возвращает копию объекта DataSet, которая содержит все изменения, внесенные в объект DataSet с момента его загрузки или последнего вызова AcceptChanges(). Этот метод имеет перегруженные версии, позволяющие получать только новые строки, только измененные строки или только удаленные строки
HasChanges()	Получает значение, которое указывает, был ли объект DataSet изменен, включая добавление, удаление либо изменение строк
Merge()	Объединяет текущий объект DataSet с указанным объектом DataSet

Окончание табл. 22.2

Метод	Описание
`ReadXml()`	Позволяет определить структуру объекта `DataSet` и заполнить его данными, основываясь на XML-схеме и данных из потока
`RejectChanges()`	Производит откат всех изменений, которые были внесены в текущий объект `DataSet` с момента его загрузки или последнего вызова метода `AcceptChanges()`
`WriteXml()`	Позволяет записать содержимое объекта `DataSet` в действительный поток

Построение объекта DataSet

Теперь, когда вы лучше понимаете роль класса `DataSet` (и имеете некоторое представление о том, что можно делать с его помощью), создадим новый проект консольного приложения по имени `SimpleDataSet` и импортируем пространство имен `System.Data`. Внутри метода `Main()` определим новый объект `DataSet`, содержащий три расширенных свойства, которые представляют временную метку, уникальный идентификатор (типа `System.Guid`) и название компании (также понадобится добавить оператор `using static System.Console;`):

```
using static System.Console;
static void Main(string[] args)
{
  WriteLine("***** Fun with DataSets *****\n");

  // Создать объект DataSet и добавить несколько свойств.
  var carsInventoryDS = new DataSet("Car Inventory");

  carsInventoryDS.ExtendedProperties["TimeStamp"] = DateTime.Now;
  carsInventoryDS.ExtendedProperties["DataSetID"] = Guid.NewGuid();
  carsInventoryDS.ExtendedProperties["Company"] =
    "Mikko's Hot Tub Super Store";

  FillDataSet(carsInventoryDS);
  PrintDataSet(carsInventoryDS);

  ReadLine();
}
```

На заметку! Глобально уникальный идентификатор (globally unique identifier — GUID) представляет собой статически уникальное 128-битное число.

Объект `DataSet` не особенно интересен, пока не вставить в него несколько объектов `DataTable`. Следовательно, понадобится исследовать внутреннее устройство `DataTable`, начав с типа `DataColumn`.

Работа с объектами DataColumn

Тип `DataColumn` представляет одиночный столбец внутри объекта `DataTable`. В целом набор всех объектов `DataColumn`, привязанных к заданному объекту `DataTable`, представляет основу информации *схемы* таблицы. Например, в случае моделирования таблицы `Inventory` из базы данных `AutoLot` (см. главу 21) будут созданы четыре объекта `DataColumn`, по одному для каждого столбца (`CarId`, `Make`, `Color` и `PetName`). Полученные объекты `DataColumn` обычно добавляются в коллекцию столбцов объекта `DataTable` (с использованием свойства `Columns`).

Вероятно, вы уже знаете, что столбцу в таблице базы данных можно назначить набор ограничений (например, сконфигурировать как первичный ключ, присвоить стандартное значение или разрешить только чтение). Кроме того, каждый столбец таблицы должен отображаться на лежащий в основе тип данных. Скажем, схема таблицы `Inventory` требует, чтобы столбец `CarId` отображался на целочисленное значение, а столбцы `Make`, `Color` и `PetName` — на массив символов. Класс `DataColumn` имеет множество свойств, которые позволяют точно конфигурировать такие аспекты. Описание ряда основных свойств приведено в табл. 22.3.

Таблица 22.3. Свойства класса `DataColumn`

Свойство	Описание
`AllowDBNull`	Применяется для указания, может ли столбец содержать значения `null`. По умолчанию равно `true`

`AutoIncrement`
`AutoIncrementSeed`
`AutoIncrementStep`
	Используются в целях настройки поведения автоинкремента для заданного столбца. Это может быть удобно, когда нужно гарантировать уникальность значений в отдельном объекте `DataColumn` (таком как первичный ключ). По умолчанию `DataColumn` не поддерживает поведение автоинкремента

`Caption`	Получает или устанавливает заголовок, который необходимо отображать для столбца. Это позволяет определить дружественную к пользователю версию для реального имени столбца в базе данных
`ColumnMapping`	Определяет представление объекта `DataColumn` при сохранении `DataSet` в документе XML посредством метода `DataSet.WriteXml()`. Можно указать, что столбец данных должен быть записан как элемент XML, атрибут XML, простое текстовое содержимое или вообще игнорироваться
`ColumnName`	Получает или устанавливает имя столбца в коллекции `Columns` (т.е. как он представлен внутри `DataTable`). Если свойство `ColumnName` не установлено явно, то стандартным значением будет слово `Column` с числовым суффиксом по формуле *n*+1 (`Column1`, `Column2`, `Column3` и т.д.)
`DataType`	Определяет тип (например, булевский, строковый или с плавающей точкой) данных, хранящихся в столбце
`DefaultValue`	Получает или устанавливает стандартное значение, присваиваемое столбцу при вставке новых строк
`Expression`	Получает или устанавливает выражение для фильтрации строк, вычисления значения столбца или создания агрегированного столбца
`Ordinal`	Получает или устанавливает числовую позицию столбца в коллекции `Columns`, поддерживаемой объектом `DataTable`
`ReadOnly`	Определяет, допускает ли заданный столбец только чтение, после добавления строки в таблицу. Стандартным значением является `false`
`Table`	Получает объект `DataTable`, который содержит текущий объект `DataColumn`
`Unique`	Получает или устанавливает значение, которое указывает, должны ли быть значения во всех строках столбца уникальными, или же разрешены повторяющиеся значения. Когда столбцу назначается ограничение первичного ключа, свойство `Unique` должно быть установлено в `true`

Построение объекта DataColumn

Чтобы продолжить работу с проектом SimpleDataSet (и проиллюстрировать применение типа DataColumn), предположим, что необходимо смоделировать столбцы таблицы Inventory. Поскольку столбец CarId будет первичным ключом таблицы, он должен быть сконфигурирован как предназначенный только для чтения, содержащий уникальные значения и не допускающий null (с использованием свойств ReadOnly, Unique и AllowDBNull). Добавим в класс Program новый метод по имени FillDataSet(), который будет применяться для построения четырех объектов DataColumn. Этот метод принимает в качестве единственного параметра объект DataSet.

```
static void FillDataSet(DataSet ds)
{
  // Создать столбцы данных, которые отображаются на "реальные"
  // столбцы в таблице Inventory из базы данных AutoLot.
  var carIDColumn = new DataColumn("CarID", typeof (int))
  {
    Caption = "Car ID",
    ReadOnly = true,
    AllowDBNull = false,
    Unique = true,
  };

  var carMakeColumn = new DataColumn("Make", typeof (string));
  var carColorColumn = new DataColumn("Color", typeof (string));
  var carPetNameColumn = new DataColumn("PetName", typeof (string))
    { Caption = "Pet Name"};
}
```

Обратите внимание, что при конфигурировании объекта carIDColumn свойству Caption было присвоено значение. Это свойство удобно, потому что позволяет определить строковое значение в целях отображения, которое может отличаться от реального имени столбца таблицы в базе данных (имена столбцов обычно больше подходят для целей программирования (например, au_fname), чем для отображения (скажем, Author First Name (Имя автора))). По той же причине был установлен заголовок для столбца PetName, т.к. Pet Name (Дружественное имя) выглядит для конечного пользователя лучше, чем PetName.

Включение автоинкрементных полей

Одним из аспектов типа DataColumn, который вы можете выбрать для конфигурирования, является возможность *автоинкремента*. Автоинкрементное поле используется для обеспечения того, что при добавлении в таблицу новой строки значение этого поля устанавливается автоматически на основе предыдущего значения и шага увеличения. Это удобно, когда нужно гарантировать, что в столбце отсутствуют повторяющиеся значения (например, в первичном ключе).

Такое поведение управляется свойствами AutoIncrement, AutoIncrementSeed и AutoIncrementStep. Значение AutoIncrementSeed применяется для определения начального значения в столбце, а AutoIncrementStep — для указания числа, которое прибавляется при вычислении каждого последующего значения. Взгляните на следующее обновление кода создания объекта carIDColumn:

```
static void FillDataSet(DataSet ds)
{
  var carIDColumn = new DataColumn("CarID", typeof (int))
  {
```

```
    Caption = "Car ID",
    ReadOnly = true,
    AllowDBNull = false,
    Unique = true,
    AutoIncrement = true,
    AutoIncrementSeed = 1,
    AutoIncrementStep = 1
  };
}
```

Здесь объект `carIDColumn` сконфигурирован так, что при добавлении новых строк в соответствующую таблицу значение в столбце увеличивается на 1. Начальное значение установлено в 1, поэтому в столбце будут содержаться числа 1, 2, 3, 4 и т.д.

Добавление объектов DataColumn в DataTable

Обычно объект типа `DataColumn` не существует как обособленная сущность, а вставляется в связанный объект `DataTable`. Для примера создадим новый объект `DataTable` и вставим все объекты `DataColumn` в коллекцию столбцов с использованием свойства `Columns`:

```
static void FillDataSet(DataSet ds):
{
  ...
  // Добавить объекты DataColumn в DataTable.
  var inventoryTable = new DataTable("Inventory");
  inventoryTable.Columns.AddRange(new[]
    {carIDColumn, carMakeColumn, carColorColumn, carPetNameColumn});
}
```

В настоящий момент объект `DataTable` содержит четыре объекта `DataColumn`, которые представляют схему находящейся в памяти таблицы `Inventory`. Тем не менее, пока что эта таблица не содержит данных и не входит в коллекцию таблиц, обслуживаемых `DataSet`. Мы восполним эти пробелы, начав с наполнения таблицы данными с применением объектов `DataRow`.

Работа с объектами DataRow

Вы видели, что коллекция объектов `DataColumn` представляет схему `DataTable`. В противовес этому коллекция объектов `DataRow` представляет действительные данные в таблице. Таким образом, если таблица `Inventory` базы данных `AutoLot` содержит 20 строк, то представить эти записи можно с использованием 20 объектов `DataRow`. В табл. 22.4 кратко описаны некоторые (но не все) члены типа `DataRow`.

Таблица 22.4. Основные члены типа DataRow

Член	Описание

`HasErrors`
`GetColumnsInError()`
`GetColumnError()`
`ClearErrors()`
`RowError`
	Свойство `HasErrors` возвращает булевское значение, указывающее на наличие ошибок в `DataRow`. Если ошибки есть, то с помощью метода `GetColumnsInError()` можно получить проблемные столбцы, а с помощью метода `GetColumnError()` — описание ошибки. Метод `ClearErrors()` позволяет очистить список ошибок для строки. Свойство `RowError` дает возможность сконфигурировать текстовое описание ошибки для заданной строки

`ItemArray`	Это свойство получает или устанавливает значения всех столбцов в строке с применением массива объектов

Окончание табл. 22.4

Член	Описание
`RowState`	Это свойство используется для определения текущего *состояния* объекта `DataRow` в содержащем его объекте `DataTable` посредством значений перечисления `RowState` (например, строка может быть помечена как новая, модифицированная, неизмененная или удаленная)
`Table`	Это свойство применяется для получения ссылки на объект `DataTable`, содержащий данный объект `DataRow`
`AcceptChanges()` `RejectChanges()`	Эти методы фиксируют или отклоняют все изменения, внесенные в данную строку с момента последнего вызова `AcceptChanges()`

`BeginEdit()`
`EndEdit()`
`CancelEdit()`
	Эти методы начинают, заканчивают или отменяют операцию редактирования для объекта `DataRow`

`Delete()`	Этот метод помечает строку, которую необходимо удалить при вызове метода `AcceptChanges()`
`IsNull()`	Этот метод возвращает значение, которое указывает, содержит ли заданный столбец `null`

Работа с объектом `DataRow` слегка отличается от работы с `DataColumn`; создавать экземпляр этого типа напрямую невозможно, т.к. открытые конструкторы в нем отсутствуют:

```
// Ошибка! Открытых конструкторов нет!
DataRow r = new DataRow();
```

Взамен новый объект `DataRow` получается из заданного объекта `DataTable`. Для примера предположим, что в таблицу `Inventory` нужно вставить две строки. Метод `DataTable.NewRow()` позволяет получить очередную область в таблице, после чего каждый столбец можно заполнить новыми данными, используя индексатор типа. При этом можно указывать либо строковое имя, назначенное объекту `DataColumn`, либо порядковый номер (начинающийся с нуля):

```
static void FillDataSet(DataSet ds)
{
  ...
  // Добавить несколько строк в таблицу Inventory.
  DataRow carRow = inventoryTable.NewRow();
  carRow["Make"] = "BMW";
  carRow["Color"] = "Black";
  carRow["PetName"] = "Hamlet";
  inventoryTable.Rows.Add(carRow);

  carRow = inventoryTable.NewRow();
  // Столбец 0 - это автоинкрементное поле идентификатора,
  // поэтому начать заполнение со столбца 1.
  carRow[1] = "Saab";
  carRow[2] = "Red";
  carRow[3] = "Sea Breeze";
  inventoryTable.Rows.Add(carRow);
}
```

На заметку! Если методу индексатора типа `DataRow` передается недействительное имя столбца или порядковый номер позиции, то во время выполнения сгенерируется исключение.

В настоящий момент имеется единственный объект DataTable, содержащий две строки. Разумеется, такой общий процесс можно повторить, чтобы определить схему и содержимое для нескольких объектов DataTable. Перед вставкой объекта inventoryTable в DataSet вы должны ознакомиться с крайне важным свойством RowState.

Свойство RowState

Свойство RowState применяется для программной идентификации набора строк таблицы, которые были изменены, заново вставлены и т.д. Этому свойство можно присваивать любое значение из перечисления DataRowState (табл. 22.5).

Таблица 22.5. Значения перечисления DataRowState

Значение	Описание
Added	Строка была добавлена в DataRowCollection, а метод AcceptChanges() пока не был вызван
Deleted	Строка была помечена для удаления с помощью метода Delete() класса DataRow, а метод AcceptChanges() пока не был вызван
Detached	Строка была создана, но не является частью какой-то коллекции DataRowCollection. Объект DataRow находится в этом состоянии непосредственно после создания, но перед добавлением в коллекцию. Он также будет в таком состоянии после удаления из коллекции
Modified	Строка была изменена, а метод AcceptChanges() пока не был вызван
Unchanged	Строка не изменялась с момента последнего вызова метода AcceptChanges()

При программном манипулировании строками заданного объекта DataTable свойство RowState устанавливается автоматически. Для примера добавим в класс Program новый метод, который работает с локальным объектом DataRow, попутно выводя на консоль состояние его строк:

```
private static void ManipulateDataRowState()
{
  // Создать объект temp типа DataTable для целей тестирования.
  var temp = new DataTable("Temp");
  temp.Columns.Add(new DataColumn("TempColumn", typeof(int)));

  // RowState = Detached.
  var row = temp.NewRow();
  WriteLine($"After calling NewRow(): {row.RowState}");

  // RowState = Added.
  temp.Rows.Add(row);
  WriteLine($"After calling Rows.Add(): {row.RowState}");

  // RowState = Added.
  row["TempColumn"] = 10;
  WriteLine($"After first assignment: {row.RowState}");

  // RowState = Unchanged.
  temp.AcceptChanges();
  WriteLine($"After calling AcceptChanges: {row.RowState}");

  // RowState = Modified.
  row["TempColumn"] = 11;
  WriteLine($"After first assignment: {row.RowState}");
```

```
    // RowState = Deleted.
    temp.Rows[0].Delete();
    WriteLine($"After calling Delete: {row.RowState}");
}
```

На заметку! Не забудьте добавить оператор `using static System.Console;` в начало файла кода в этом примере (и во всех других примерах, где используется консоль).

Тип `DataRow` в ADO.NET достаточно интеллектуален, чтобы запоминать свое текущее положение дел. В итоге владеющий им объект `DataTable` способен идентифицировать, какие строки были добавлены, обновлены или удалены. Это важная особенность `DataSet`, т.к. когда наступает время передачи обновленной информации хранилищу данных, отправляются только модифицированные данные.

Свойство DataRowVersion

Помимо отслеживания текущего состояния строки посредством свойства `RowState` объект `DataRow` поддерживает три возможных версии содержащихся в нем данных с помощью свойства `DataRowVersion`. Когда объект `DataRow` сконструирован впервые, он содержит только единственную копию данных, которая является текущей версией. Однако по мере программного манипулирования объектом `DataRow` (с применением разнообразных методов) появляются дополнительные версии данных. Свойство `DataRowVersion` может быть установлено в любое значение перечисления `DataRowVersion` (табл. 22.6).

Таблица 22.6. Значения перечисления DataRowVersion

Значение	Описание
`Current`	Представляет текущее значение строки, даже после внесения изменений
`Default`	Стандартная версия `DataRowState`. Если значение `DataRowState` равно `Added`, `Modified` или `Deleted`, то стандартной версией является `Current`. Для значения `DataRowState`, равного `Detached`, стандартной версией будет `Proposed`
`Original`	Представляет значение, первоначально вставленное в `DataRow`, или значение при последнем вызове метода `AcceptChanges()`
`Proposed`	Значение строки, редактируемой в текущий момент по причине вызова `BeginEdit()`

В табл. 22.6 видно, что значение свойства `DataRowVersion` во многих случаях зависит от значения свойства `DataRowState`. Как упоминалось ранее, свойство `DataRowVersion` будет автоматически изменяться при вызовах различных методов на объекте `DataRow` (или в ряде ситуаций на объекте `DataTable`). Ниже приведен обзор методов, которые могут повлиять на значение свойства `DataRowVersion` отдельной строки.

- Если вызывается метод `DataRow.BeginEdit()` и изменяется значение строки, то становятся доступными значения `Current` и `Proposed`.
- Если вызывается метод `DataRow.CancelEdit()`, то значение `Proposed` удаляется.
- После вызова метода `DataRow.EndEdit()` значение `Proposed` становится значением `Current`.
- После вызова метода `DataRow.AcceptChanges()` значение `Original` становится идентичным значению `Current`. То же самое происходит и при вызове метода `DataTable.AcceptChanges()`.
- После вызова метода `DataRow.RejectChanges()` значение `Proposed` отбрасывается, и версия становится `Current`.

Действительно, это немного запутанно, и не в последнюю очередь из-за того, что в любой момент времени объект DataRow может иметь или же не иметь все версии (при попытке получить версию строки, которая в текущий момент не отслеживается, возникают исключения времени выполнения). Несмотря на сложность, поскольку объект DataRow поддерживает три копии данных, становится более простым построение пользовательского интерфейса, который позволяет конечному пользователю изменить значения и затем отказаться от изменений или зафиксировать новые значения для постоянного сохранения. В оставшемся материале главы вы увидите разнообразные примеры манипулирования этими методами.

Работа с объектами DataTable

В типе DataTable определены многочисленные члены, многие из которых идентичны по именам и функциональности членам типа DataSet. В табл. 22.7 кратко описаны основные члены типа DataTable кроме Rows и Columns.

Таблица 22.7. Основные члены типа DataTable

Член	Описание
CaseSensitive	Указывает, должно ли сравнение строк внутри таблицы быть чувствительным к регистру символов. Стандартное значение равно false
ChildRelations	Возвращает коллекцию дочерних отношений для этого объекта DataTable (если они есть)
Constraints	Возвращает коллекцию ограничений, поддерживаемых таблицей
Copy()	Метод, который копирует схему и данные из объекта DataTable в новый такой объект
DataSet	Возвращает объект DataSet, содержащий эту таблицу (если он есть)
DefaultView	Возвращает настроенное представление таблицы, которое может включать отфильтрованное представление или позицию курсора
ParentRelations	Возвращает коллекцию родительских отношений для этого объекта DataTable
PrimaryKey	Получает или устанавливает массив столбцов, которые выступают в качестве первичных ключей для таблицы данных
TableName	Получает или устанавливает имя таблицы. Значение для этого свойства можно также указать в параметре конструктора

Чтобы продолжить текущий пример, присвоим свойству PrimaryKey в DataTable объект DataColumn по имени carIDColumn. Имейте в виду, что свойству PrimaryKey должна присваиваться коллекция объектов DataColumn, чтобы учесть ключ, состоящий из нескольких столбцов. Тем не менее, в рассматриваемом случае необходимо указать только столбец CarId (находящийся в самой первой позиции таблицы):

```
static void FillDataSet(DataSet ds)
{
  ...
  // Установить первичный ключ этой таблицы.
  inventoryTable.PrimaryKey = new [] { inventoryTable.Columns[0] };
}
```

Вставка объектов DataTable в DataSet

К настоящему моменту построение объекта DataTable завершено. Осталось вставить его в объект DataSet по имени carsInventoryDS, используя коллекцию Tables:

```
static void FillDataSet(DataSet ds)
{
  ...
  // Наконец, добавить таблицу в DataSet.
  ds.Tables.Add(inventoryTable);
}
```

Теперь обновим метод Main(), добавив вызов метода FillDataSet(), которому в качестве аргумента передается локальный объект DataSet. Затем этот же объект передадим новому (пока еще не написанному) вспомогательному методу по имени PrintDataSet():

```
static void Main(string[] args)
{
  WriteLine("***** Fun with DataSets *****\n");
  ...
  FillDataSet(carsInventoryDS);
  PrintDataSet(carsInventoryDS);
  ReadLine();
}
```

Получение данных из объекта DataSet

Метод PrintDataSet() просто проходит по метаданным DataSet (с применением коллекции ExtendedProperties) и по каждому объекту DataTable внутри DataSet, выводя на консоль имена столбцов и значения строк с помощью индексаторов типов. Добавим в начало файла оператор using для пространства имен System.Collections, чтобы получить доступ к типу DictionaryEntry:

```
static void PrintDataSet(DataSet ds)
{
  // Вывести имя DataSet и любые расширенные свойства.
  WriteLine($"DataSet is named: {ds.DataSetName}");
  foreach (DictionaryEntry de in ds.ExtendedProperties)
  {
    WriteLine($"Key = {de.Key}, Value = {de.Value}");
  }
  WriteLine();

  // Вывести содержимое каждой таблицы.
  foreach (DataTable dt in ds.Tables)
  {
    WriteLine($"=> {dt.TableName} Table:");

    // Вывести имена столбцов.
    for (var curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write($"{dt.Columns[curCol].ColumnName}\t");
    }
    WriteLine("\n---------------------------------");

    // Вывести содержимое DataTable.
    for (var curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
```

```
      for (var curCol = 0; curCol < dt.Columns.Count; curCol++)
      {
        Write($"{dt.Rows[curRow][curCol]}\t");
      }
      WriteLine();
    }
  }
}
```

Если теперь запустить программу, то будет получен следующий вывод (конечно, метка времени и значение GUID у вас будут другими):

```
***** Fun with DataSets *****

DataSet is named: Car Inventory
Key = TimeStamp, Value = 7/24/2015 6:41:09 AM
Key = DataSetID, Value = 11c533ed-d1aa-4c82-96d4-b0f88893ab21
Key = Company, Value = Mikko's Hot Tub Super Store

=> Inventory Table:
CarID   Make    Color   PetName
---------------------------------
1       BMW     Black   Hamlet
2       Saab    Red     Sea Breeze
```

Обработка данных в DataTable с использованием объектов DataTableReader

Учитывая проделанную в главе 21 работу, вы должны заметить, что способы обработки данных с применением подключенного уровня (объектов чтения данных) и автономного уровня (объектов `DataSet`) совершенно отличаются. Работа с объектом чтения данных обычно предусматривает организацию цикла `while`, вызов метода `Read()` и выборку пар "имя-значение" с использованием индексатора. С другой стороны, при обработке объекта `DataSet` обычно задействуется последовательность итерационных конструкций для обращения к данным внутри таблиц, строк и столбцов (вспомните, что объект `DataReader` требует открытого подключения к базе данных, чтобы он мог читать данные из действительной базы).

Объекты `DataTable` поддерживают метод по имени `CreateDataReader()`. Этот метод позволяет получать данные из `DataTable` с применением схемы навигации, которая похожа на схему, реализованную объектом чтения данных (теперь данные будут читаться из объекта `DataTable` в памяти, а не из действительной базы данных, так что подключение к базе данных здесь не участвует). Главное преимущество такого подхода связано с тем, что для обработки данных используется единая модель независимо от того, какой уровень ADO.NET применяется для получения данных. Предположим, что в класс `Program` добавлен новый метод по имени `PrintTable()`:

```
static void PrintTable(DataTable dt)
{
  // Получить объект DataTableReader.
  DataTableReader dtReader = dt.CreateDataReader();

  // DataTableReader работает в точности как DataReader.
  while (dtReader.Read())
  {
    for (var i = 0; i < dtReader.FieldCount; i++)
    {
      Write($"{dtReader.GetValue(i).ToString().Trim()}\t");
    }
```

```
    WriteLine();
  }
  dtReader.Close();
}
```

Обратите внимание, что DataTableReader работает идентично объекту чтения данных вашего поставщика данных. Тип DataTableReader может быть идеальным вариантом, когда нужно быстро извлечь данные из DataTable без необходимости в обходе внутренних коллекций строк и столбцов. Теперь предположим, что предыдущий метод PrintDataSet() модифицирован для вызова PrintTable() вместо работы с коллекциями Rows и Columns:

```
static void PrintDataSet(DataSet ds)
{
  // Вывести имя DataSet и любые расширенные свойства.
  WriteLine($"DataSet is named: {ds.DataSetName}");
  foreach (DictionaryEntry de in ds.ExtendedProperties)
  {
    WriteLine($"Key = {de.Key}, Value = {de.Value}");
  }
  WriteLine();

  // Вывести содержимое каждой таблицы, используя объект чтения данных
  foreach (DataTable dt in ds.Tables)
  {
    WriteLine($"=> {dt.TableName} Table:");

    // Вывести имена столбцов.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write($"{dt.Columns[curCol].ColumnName.Trim()}\t");
    }
    WriteLine("\n---------------------------------");

    // Вызвать новый вспомогательный метод.
    PrintTable(dt);
  }
}
```

В результате запуска приложения будет получен вывод, идентичный показанному ранее. Единственное отличие в способе доступа к содержимому DataTable.

Сериализация объектов DataTable и DataSet в формате XML

Типы DataSet и DataTable поддерживают методы WriteXml() и ReadXml(). Метод WriteXml() позволяет сохранять содержимое объекта внутри локального файла (а также в любом типе, производном от System.IO.Stream) в виде документа XML. Метод ReadXml() дает возможность восстановить состояние объекта DataSet (или DataTable) из указанного документа XML. Вдобавок типы DataSet и DataTable поддерживают методы WriteXmlSchema() и ReadXmlSchema(), предназначенные для сохранения или загрузки файла *.xsd.

В целях тестирования модифицируем код Main(), чтобы в нем вызывался следующий вспомогательный метод (которому передается единственный параметр типа DataSet):

```
static void SaveAndLoadAsXml(DataSet carsInventoryDS)
{
  // Сохранить этот объект DataSet в формате XML.
  carsInventoryDS.WriteXml("carsDataSet.xml");
  carsInventoryDS.WriteXmlSchema("carsDataSet.xsd");
```

```
  // Очистить объект DataSet.
  carsInventoryDS.Clear();

  // Загрузить объект DataSet из файла XML.
  carsInventoryDS.ReadXml("carsDataSet.xml");
}
```

Если открыть файл `carsDataSet.xml` (который находится в папке `bin\Debug` проекта), то можно обнаружить, что каждый столбец таблицы закодирован как элемент XML:

```
<?xml version="1.0" standalone="yes"?>
<Car_x0020_Inventory>
  <Inventory>
    <CarID>1</CarID>
    <Make>BMW</Make>
    <Color>Black</Color>
    <PetName>Hamlet</PetName>
  </Inventory>
  <Inventory>
    <CarID>2</CarID>
    <Make>Saab</Make>
    <Color>Red</Color>
    <PetName>Sea Breeze</PetName>
  </Inventory>
</Car_x0020_Inventory>
```

Двойной щелчок на сгенерированном файле `.xsd` (также находящемся в папке `bin\Debug`) внутри Visual Studio приводит к открытию встроенного редактора схемы XML (рис. 22.3).

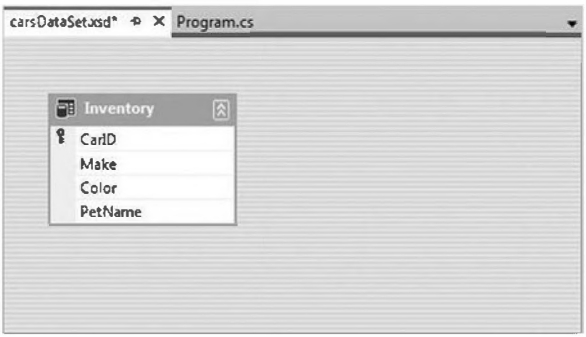


Рис. 22.3. Редактор XSD в Visual Studio

На заметку! В главе 24 будет представлен API-интерфейс LINQ to XML, который является рекомендуемым способом манипулирования XML-данными в рамках платформы .NET.

Сериализация объектов `DataTable` и `DataSet` в двоичном формате

Содержимое объекта `DataSet` (или отдельного `DataTable`) можно также сохранять в компактном двоичном формате. Это особенно полезно, когда объект `DataSet` должен передаваться за границы машины (в случае распределенного приложения). Один из недостатков представления данных в виде XML связан с тем, что его дескриптивная природа может привести к высоким накладным расходам при передаче.

Чтобы сохранить объекты `DataTable` или `DataSet` в двоичном формате, понадобится установить свойство `RemotingFormat` в `SerializationFormat.Binary`. После этого вполне ожидаемо можно использовать тип `BinaryFormatter` (см. главу 20). Рассмотрим следующий финальный метод проекта `SimpleDataSet` (не забудьте импортировать пространства имен `System.IO` и `System.Runtime.Serialization.Formatters.Binary`):

```
static void SaveAndLoadAsBinary(DataSet carsInventoryDS)
{
  // Установить флаг двоичной сериализации.
  carsInventoryDS.RemotingFormat = SerializationFormat.Binary;

  // Сохранить этот объект DataSet в двоичном виде.
  var fs = new FileStream("BinaryCars.bin", FileMode.Create);
  var bFormat = new BinaryFormatter();
  bFormat.Serialize(fs, carsInventoryDS);
  fs.Close();

  // Очистить объект DataSet.
  carsInventoryDS.Clear();

  // Загрузить объект DataSet из двоичного файла.
  fs = new FileStream("BinaryCars.bin", FileMode.Open);
  var data = (DataSet)bFormat.Deserialize(fs);
}
```

После вызова метода `SaveAndLoadAsBinary()` внутри `Main()` в папке `bin\Debug` можно будет найти файл `*.bin`. На рис. 22.4 показано содержимое файла `BinaryCars.bin`.

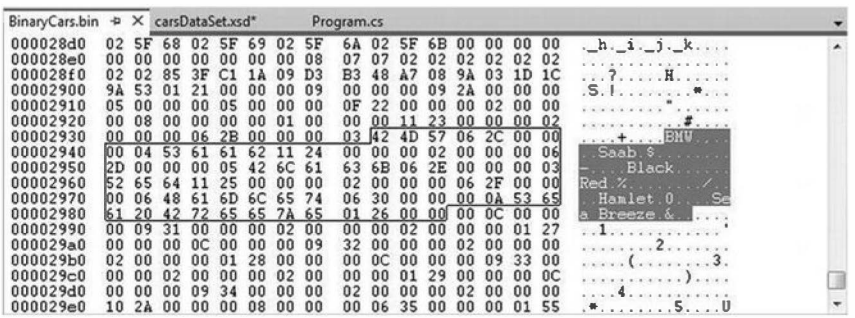


Рис. 22.4. Объект `DataSet` сохранен в двоичном формате

Исходный код. Проект `SimpleDataSet` доступен в подкаталоге `Chapter_22`.

Привязка объектов `DataTable` к графическому пользовательскому интерфейсу Windows Forms

До сих пор вы видели, каким образом создавать, заполнять и проходить по содержимому объекта `DataSet` вручную с применением унаследованной объектной модели ADO.NET. Хотя знать, как это делается, довольно-таки важно, платформа .NET поставляется с многочисленными API-интерфейсами, которые обладают возможностью автоматической *привязки* данных к элементам пользовательского интерфейса.

Например, первоначальный набор инструментов для построения графических пользовательских интерфейсов .NET, Windows Forms, предлагает элемент управления по имени `DataGridView`, который обладает встроенной возможностью отображения содержимого объекта `DataSet` или `DataTable` с использованием всего нескольких строк кода.

Инфраструктуры ASP.NET (API-интерфейс для разработки веб-приложений в .NET) и Windows Presentation Foundation также поддерживают понятие привязки данных. Вы научитесь привязывать данные к графическим элементам WPF и ASP.NET позже в книге; однако в настоящей главе будет применяться инфраструктура Windows Forms, т.к. ей присуща довольно простая и понятная модель программирования.

На заметку! В следующем примере предполагается наличие у вас некоторого опыта использования Windows Forms для построения графических пользовательских интерфейсов. Если это не так, то можете просто открыть готовое решение (доступное в загружаемом коде примеров) и продолжить чтение.

Теперь нам предстоит построить приложение Windows Forms, которое будет отображать содержимое объекта `DataTable` внутри своего пользовательского интерфейса. Попутно вы увидите, каким образом фильтровать и изменять табличные данные. Вы также ознакомитесь с ролью объекта `DataView`.

Начнем с создания нового проекта приложения Windows Forms по имени `WindowsFormsDataBinding`. В окне Solution Explorer назначим начальному файлу вместо `Form1.cs` более подходящее имя `MainForm.cs`. В окне свойств изменим заголовок формы на `Windows Forms Data Binding` (Привязка данных в Windows Forms). Затем перетащим из панели инструментов Visual Studio элемент `DataGridView` на поверхность визуального конструктора (и переименуем его в `carInventoryGridView` с применением свойства `(Name)` в окне свойств). Можно заметить, что при добавлении элемента `DataGridView` на поверхность визуального конструктора в первый раз активизируется контекстное меню, которое позволяет подключиться к физическому источнику данных. В настоящий момент этот аспект визуального конструктора следует проигнорировать, т.к. привязка объекта `DataTable` будет осуществляться программно. Наконец, поместим на поверхность визуального конструктора элемент `Label` с описательным текстом. Возможный внешний вид пользовательского интерфейса представлен на рис. 22.5.

Рис. 22.5. Первоначальный графический пользовательский интерфейс приложения Windows Forms

Заполнение `DataTable` из обобщенного `List<T>`

Подобно предыдущему примеру `SimpleDataSet` в приложении `WindowsForms DataBinding` будет конструироваться объект `DataTable`, который содержит набор объектов `DataColumn`, представляющих разнообразные столбцы и строки данных. Но на этот раз строки будут заполняться с использованием переменной-члена обобщенного

типа List<T>. Для начала вставим в проект новый класс C# по имени Car, определенный следующим образом:

```
public class Car
{
  public int Id { get; set; }
  public string PetName { get; set; }
  public string Make { get; set; }
  public string Color { get; set; }
}
```

Внутри стандартного конструктора главной формы заполним переменную-член типа List<T> (listCars) набором новых объектов Car:

```
public partial class MainForm : Form
{
 // Коллекция объектов Car.
 List<Car> listCars = null;
 public MainForm()
 {
  InitializeComponent();
  // Заполнить список объектами Car.
  listCars = new List<Car>
  {
    new Car { Id = 1, PetName = "Chucky", Make = "BMW", Color = "Green" },
    new Car { Id = 2, PetName = "Tiny", Make = "Yugo", Color = "White" },},
    new Car { Id = 3, PetName = "Ami", Make = "Jeep", Color = "Tan" },
    new Car { Id = 4, PetName = "Pain Inducer", Make = "Caravan", Color = "Pink" },
    new Car { Id = 5, PetName = "Fred", Make = "BMW", Color = "Green" },
    new Car { Id = 6, PetName = "Sidd", Make = "BMW", Color = "Black" },
    new Car { Id = 7, PetName = "Mel", Make = "Firebird", Color = "Red" },
    new Car { Id = 8, PetName = "Sarah", Make = "Colt", Color = "Black" },
  };
 }
}
```

Добавим в класс MainForm новую переменную-член типа DataTable по имени inventoryTable:

```
public partial class MainForm : Form
{
  // Коллекция объектов Car.
  List<Car> listCars = null;
  // Складская информация.
  DataTable inventoryTable = new DataTable();
  ...
}
```

Теперь добавим в класс MainForm новый вспомогательный метод CreateDataTable() и вызовем его в стандартном конструкторе класса MainForm:

```
void CreateDataTable()
{
  // Создать схему таблицы.
  var carIDColumn = new DataColumn("Id", typeof(int));
  var carMakeColumn = new DataColumn("Make", typeof(string));
  var carColorColumn = new DataColumn("Color", typeof(string));
  var carPetNameColumn = new DataColumn("PetName", typeof (string))
    { Caption = "Pet Name"};
  inventoryTable.Columns.AddRange(
    new[] { carIDColumn, carMakeColumn,carColorColumn, carPetNameColumn });
```

```
    // Пройти по элементам List<Car> для создания строк.
    foreach (var c in listCars)
    {
      var newRow = inventoryTable.NewRow();
      newRow["Id"] = c.Id;
      newRow["Make"] = c.Make;
      newRow["Color"] = c.Color;
      newRow["PetName"] = c.PetName;
      inventoryTable.Rows.Add(newRow);
    }
    // Привязать объект DataTable к carInventoryGridView.
    carInventoryGridView.DataSource = inventoryTable;
  }
```

Реализация метода начинается с построения схемы `DataTable` путем создания четырех объектов `DataColumn` (ради простоты можно не заботиться об автоинкременте для поля `CarId` или об установке его как первичного ключа). Затем их можно добавить в коллекцию столбцов, доступную через переменную-член класса `DataTable`. Данные строк из коллекции `List<T>` сопоставляются с `DataTable` посредством итерационной конструкции `foreach` и собственной объектной модели ADO.NET.

Тем не менее, обратите внимание, что в последнем операторе внутри метода `CreateDataTable()` объект `inventoryTable` присваивается свойству `DataSource` объекта `DataGridView`. Установка этого единственного свойства — все, что требуется сделать для привязки `DataTable` к объекту `DataGridView` из Windows Forms. "За кулисами" указанный элемент управления графического пользовательского интерфейса читает коллекции строк и столбцов во многом подобно тому, что происходило в методе `PrintDataSet()` из примера `SimpleDataSet`. Запустив приложение, можно просматривать содержимое объекта `DataTable` внутри элемента управления `DataGridView` (рис. 22.6).

Рис. 22.6. Привязка объекта `DataTable` к элементу `DataGridView` из Windows Forms

Удаление строк из `DataTable`

Теперь предположим, что необходимо обновить графический интерфейс, предоставив пользователю возможность удаления строки из находящегося в памяти объекта `DataTable`, который привязан к элементу управления `DataGridView`. Один из подходов предусматривает вызов метода `Delete()` объекта `DataRow`, который представляет удаляемую строку. В таком случае указывается индекс (или объект `DataRow`), соответствующий этой строке. Чтобы позволить пользователю выбрать строку для удаления, добавим на поверхность визуального конструктора элементы управления `TextBox` (по

имени txtRowToRemove) и Button (по имени btnRemoveRow). На рис. 22.7 показан возможный внешний вид интерфейса после обновления (обратите внимание на помещение этих двух элементов управления внутрь GroupBox для демонстрации их взаимосвязи).

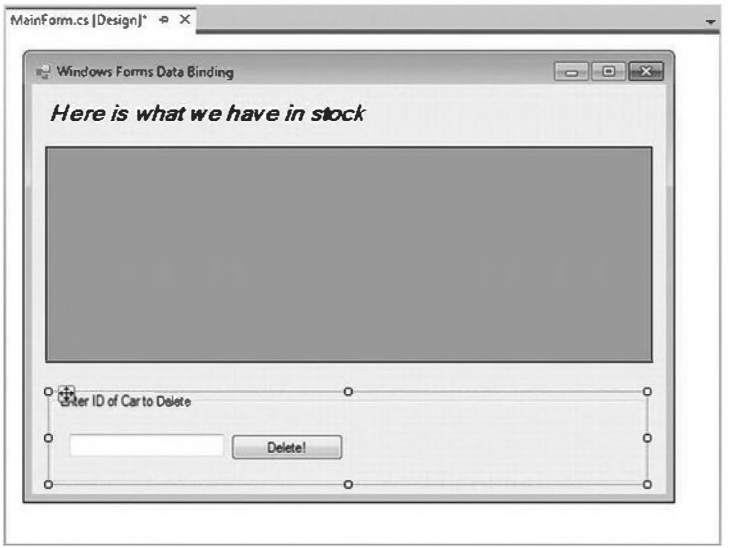


Рис. 22.7. Обновление пользовательского интерфейса для включения возможности удаления строк из DataTable

Ниже приведен код обработчика события Click новой кнопки, в котором из находящегося в памяти объекта DataTable удаляется строка согласно введенного пользователем идентификатора автомобиля. Метод Select() класса DataTable позволяет указывать критерий поиска, моделируемый посредством обычного синтаксиса SQL. Возвращаемым значением является массив объектов DataRow, которые удовлетворяют критерию поиска.

```
// Удалить эту строку из DataRowCollection.
private void btnRemoveCar_Click (object sender, EventArgs e)
{
  try
  {
    // Найти корректную строку для удаления.
    DataRow[] rowToDelete =
       inventoryTable.Select($"Id={int.Parse(txtCarToRemove.Text)}");

    // Удалить ее.
    rowToDelete[0].Delete();
    inventoryTable.AcceptChanges();
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

Теперь можно запустить приложение и указать идентификатор удаляемого автомобиля. При удалении объектов DataRow из DataTable пользовательский интерфейс DataGridView обновляется немедленно, т.к. он привязан к состоянию объекта DataTable.

Выборка строк на основе критерия фильтрации

Многие приложения обработки данных требуют возможности просмотра небольшого подмножества данных из `DataTable` на основе указанного критерия фильтрации. Например, предположим, что необходимо отобразить только автомобили определенной марки (скажем, BMW) из объекта `DataTable`, хранящегося в памяти. Вы уже видели, как с помощью метода `Select()` класса `DataTable` находилась строка, подлежащая удалению, но его можно также применять для выборки подмножества записей в целях отображения.

Чтобы взглянуть на это в действии, снова изменим пользовательский интерфейс, предоставив на этот раз пользователям возможность указывать строку, которая представляет интересующую модель автомобиля (рис. 22.8), используя новые элементы управления `TextBox` (`txtMakeToView`) и `Button` (`btnDisplayMakes`).

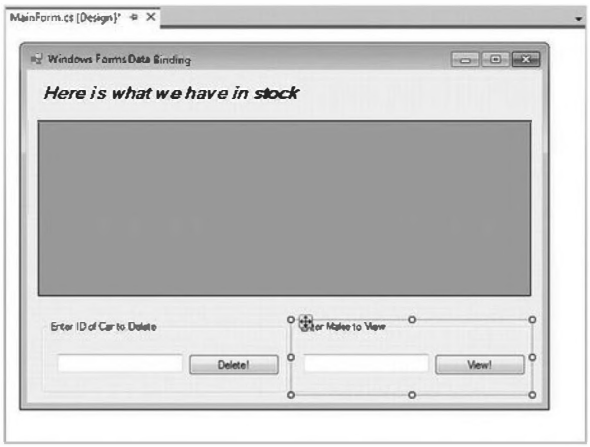


Рис. 22.8. Обновление пользовательского интерфейса для включения фильтрации строк

Метод `Select()` имеет несколько перегруженных версий, которые предлагают разную семантику выборки. В простейшем случае передаваемый `Select()` параметр является строкой, содержащей условное выражение. Для начала рассмотрим логику обработчика события `Click` для новой кнопки:

```
private void btnDisplayMakes_Click(object sender, EventArgs e)
{
  // Построить фильтр на основе пользовательского ввода.
  string filterStr = $"Make='{txtMakeToView.Text}'";
  // Найти все строки, удовлетворяющие фильтру.
  DataRow[] makes = inventoryTable.Select(filterStr);
  // Показать полученные результаты.
  if (makes.Length == 0)
    MessageBox.Show("Sorry, no cars...", "Selection error!");
  else
  {
    string strMake = null;
    for (var i = 0; i < makes.Length; i++)
    {
      strMake += makes[i]["PetName"] + "\n";
    }
    // Вывести все результаты в окне сообщений.
    MessageBox.Show(strMake,$"We have {txtMakeToView.Text}s named:");
  }
}
```

Здесь сначала создается простой фильтр на основе значения из связанного элемента управления TextBox. Если в текстовом поле фильтра указано BMW, то получится фильтр вида Make = 'BMW'. Передача этого фильтра методу Select() приводит к возвращению массива объектов DataRow, который представляет строки, удовлетворяющие данному фильтру (рис. 22.9).

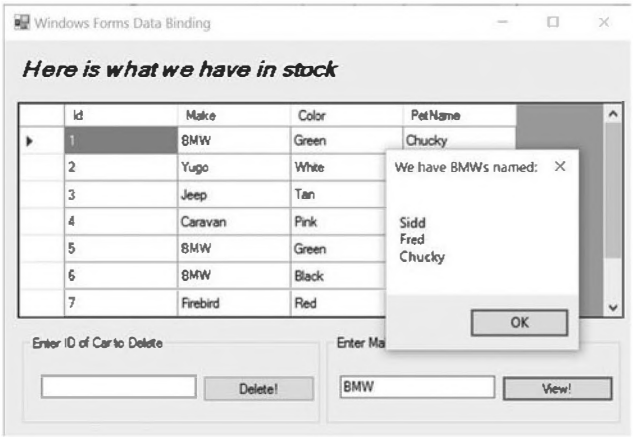


Рис. 22.9. Отображение фильтрованных данных

Логика фильтрации основана на стандартном синтаксисе SQL. Предположим что результаты предыдущего вызова Select() нужно получить в алфавитном порядке по столбцу PetName. К счастью, доступна перегруженная версия метода Select(), которая позволяет указывать критерий сортировки:

```
// Сортировать по PetName.
makes = inventoryTable.Select(filterStr, "PetName");
```

Если необходимо вывести результаты в убывающем порядке, то понадобится вызвать метод Select() следующим образом:

```
// Возвратить результаты в убывающем порядке.
makes = inventoryTable.Select(filterStr, "PetName DESC");
```

В общем случае строка с критерием сортировки содержит имя столбца, за которым указана конструкция ASC (по возрастанию; принимается по умолчанию) или DESC (по убыванию). Можно также указывать несколько столбцов, разделяя их запятыми. И, наконец, строка с критерием фильтрации может содержать любое количество операций отношения. Например, ниже показан вспомогательный метод, который выполняет поиск всех автомобилей со значением идентификатора больше 5:

```
private void ShowCarsWithIdGreaterThanFive()
{
  // Вывести дружественные имена всех автомобилей со значением ID больше 5.
  DataRow[] properIDs;
  string newFilterStr = "ID > 5";
  properIDs = inventoryTable.Select(newFilterStr);
  string strIDs = null;
  for(int i = 0; i < properIDs.Length; i++)
  {
    DataRow temp = properIDs[i];
    strIDs += $"{temp["PetName"]} is ID {temp["ID"]}\n";
  }
  MessageBox.Show(strIDs, "Pet names of cars where ID > 5");
}
```

Обновление строк в DataTable

Последний аспект DataTable, о котором вы должны знать — это процесс обновления существующих строк новыми значениями. Один из подходов предусматривает сначала получение строки, удовлетворяющей заданному критерию фильтрации (или нескольких таких строк), с помощью метода Select(). После получения интересующего объекта (или объектов) DataRow необходимо соответствующим образом модифицировать содержимое. Пусть на форме имеется новый элемент Button по имени btnChangeMakes, щелчок на котором приводит к поиску в DataTable всех строк, содержащих значение BMW в столбце Make. После получения таких строк значение Make изменяется на Yugo:

```
// Найти с помощью фильтра все строки, которые нужно отредактировать.
private void btnChangeMakes_Click(object sender, EventArgs e)
{
  // Удостовериться, что пользователь не изменил выбор.
  if (DialogResult.Yes !=
    MessageBox.Show("Are you sure?? BMWs are much nicer than Yugos!",
    "Please Confirm!", MessageBoxButtons.YesNo)) return;
  // Построить фильтр.
  string filterStr = "Make='BMW'";
  // Найти все строки, соответствующие фильтру.
  DataRow[] makes = inventoryTable.Select(filterStr);
  // Заменить BMW на Yugo.
  for (int i = 0; i < makes.Length; i++)
  {
    makes[i]["Make"] = "Yugo";
  }
}
```

Работа с типом DataView

Объект представления — это альтернативный вид таблицы (или набора таблиц). Например, с помощью Microsoft SQL Server можно создать представление для таблицы Inventory, которое возвращает новую таблицу, содержащую автомобили только указанного цвета. В ADO.NET тип DataView позволяет программным образом извлекать подмножество данных из DataTable в отдельный объект.

Серьезное преимущество наличия множества представлений одной и той же таблицы заключается в том, что их можно привязывать к разнообразным элементам управления графического пользовательского интерфейса (таким как DataGridView). Например, один DataGridView может быть привязан к объекту DataView, показывающему все автомобили из таблицы Inventory, тогда как другой можно сконфигурировать на отображение только автомобилей зеленого цвета.

Чтобы увидеть это в действии, добавим в текущий пользовательский интерфейс элемент управления DataGridView под названием dataGridColtsView и элемент Label с описанием. Затем определим переменную-член типа DataView по имени coltsOnlyView:

```
public partial class MainForm : Form
{
  // Представление DataTable.
  DataView yugosOnlyView;
  ...
}
```

Теперь создадим новый вспомогательный метод CreateDataView() и вызовем его в стандартном конструкторе главной формы сразу после того, как объект DataTable полностью сконструирован:

```
public MainForm()
{
  ...
  // Создать таблицу данных.
  CreateDataTable();
  // Создать представление.
  CreateDataView();
}
```

Ниже показана реализация нового вспомогательного метода. Обратите внимание, что конструктору `DataView` передается объект `DataTable`, который будет применяться для построения специального набора строк данных.

```
private void CreateDataView()()
{
  // Установить таблицу, которая используется для создания этого представления.
  yugosOnlyView = new DataView(inventoryTable);
  // Сконфигурировать представление с применением фильтра.
  yugosOnlyView.RowFilter = "Make = 'Yugo'";
  // Привязать к новому элементу DataGridView.
  dataGridYugosView.DataSource = yugosOnlyView;
}
```

Как видите, класс `DataView` поддерживает свойство `RowFilter`, которое содержит строку с критерием фильтрации, используемым для извлечения интересующих строк. После создания представления установим соответствующим образом свойство `DataSource` элемента управления `DataGridView`. Завершенное приложение в действии показано на рис. 22.10.

Рис. 22.10. Отображение уникального представления данных

Исходный код. Проект `WindowsFormsDataBinding` доступен в подкаталоге `Chapter_22`.

Работа с адаптерами данных

Теперь, когда вы понимаете особенности ручного манипулирования объектами `DataSet` в ADO.NET, самое время переключить внимание на тему *объектов адаптеров данных*. Адаптер данных — это класс, применяемый для заполнения `DataSet` объектами `DataTable`; он также может отправлять модифицированные объекты `DataTable` обратно базе данных на обработку. В табл. 22.8 кратко описаны основные члены базового класса `DbDataAdapter`, который является общим родительским классом для всех объектов адаптеров данных (скажем, `SqlDataAdapter` и `OdbcDataAdapter`).

Таблица 22.8. Основные члены класса `DbDataAdapter`

Член	Описание
`Fill()`	Выполняет SQL-команду `SELECT` (указанную в свойстве `SelectCommand`) для выдачи запроса к базе данных и загрузки результирующих данных в объект `DataTable`

`SelectCommand`
`InsertCommand`
`UpdateCommand`
`DeleteCommand`
	Устанавливают SQL-команды, которые будут отправляться хранилищу данных, когда вызываются методы `Fill()` и `Update()`

`Update()`	Выполняет SQL-команды `INSERT`, `UPDATE` и `DELETE` (указанные в свойствах `InsertCommand`, `UpdateCommand` и `DeleteCommand`) для сохранения в базе данных изменений, произведенных в объекте `DataTable`

Обратите внимание, что адаптер данных определяет четыре свойства: `SelectCommand`, `InsertCommand`, `UpdateCommand` и `DeleteCommand`. При создании объекта адаптера данных для конкретного поставщика (например, `SqlDataAdapter`) можно передавать строку с текстом команды, которая будет применяться объектом команды `SelectCommand`.

Предполагая, что каждый из четырех объектов команд должным образом сконфигурирован, можно вызвать метод `Fill()` для получения объекта `DataSet` (или при желании одиночного объекта `DataTable`). Чтобы сделать это, адаптер данных должен выполнить SQL-оператор `SELECT`, указанный в свойстве `SelectCommand`.

Аналогично, если необходимо сохранить модифицированный объект `DataSet` (или `DataTable`) в базе данных, можно вызвать метод `Update()`, который будет использовать какой-то из оставшихся объектов команд в зависимости от состояния каждой строки в `DataTable` (вскоре мы обсудим это более подробно).

Одним из самых необычных аспектов работы с объектом адаптера данных является отсутствие потребности в открытии или закрытии подключения к базе данных. Взамен управление подключением к базе данных происходит автоматически. Однако адаптеру данных по-прежнему необходимо предоставить действительный объект подключения или строку подключения (на основе которой внутренне строится объект подключения), чтобы сообщить, с какой базой данных нужно взаимодействовать.

На заметку! Адаптер данных независим по своей природе. К нему можно присоединять на лету разные объекты подключения и объекты команд и извлекать информацию из широкого разнообразия баз данных. Например, единственный объект `DataSet` может содержать табличные данные, полученные от поставщиков данных SQL Server, Oracle и MySQL.

Простой пример адаптера данных

Следующий шаг заключается в добавлении новой функциональности в сборку библиотеки доступа к данным (AutoLotDAL.dll), которая была создана в главе 21. Начнем с простого примера, в котором объект DataSet заполняется одной таблицей с применением объекта адаптера данных ADO.NET.

Создадим новый проект консольного приложения по имени FillDataSetUsingSqlDataAdapter и импортируем в первоначальный файл кода C# пространства имен System.Data, System.Data.SqlClient и System.Collections. Далее модифицируем метод Main() следующим образом (в зависимости от того, как создавалась база данных AutoLot в главе 21, может понадобиться изменить строку подключения):

```
static void Main(string[] args)
{
  WriteLine("***** Fun with Data Adapters *****\n");

  // Жестко закодированная строка подключения.
  string connectionString = "Integrated Security = SSPI;Initial
Catalog=AutoLot;" +
    @"Data Source=(local)\SQLEXPRESS2014";

  // Объект DataSet создается вызывающим кодом.
  DataSet ds = new DataSet("AutoLot");

  // Указать адаптеру текст команды Select и строку подключения.
  SqlDataAdapter adapter =
    new SqlDataAdapter("Select * From Inventory", connectionString);

  // Заполнить DataSet новой таблицей по имени Inventory.
  adapter.Fill(ds, "Inventory");

  // Отобразить содержимое DataSet.
  PrintDataSet(ds);
  ReadLine();
}
```

Обратите внимание, что адаптер данных конструируется с указанием строкового литерала, который будет отображен на SQL-оператор SELECT. Это значение будет использоваться для внутреннего построения объекта команды, который позже можно получить через свойство SelectCommand.

Кроме того, создание экземпляра класса DataSet, который затем передается методу Fill(), является работой вызывающего кода. Дополнительно методу Fill() можно передать во втором аргументе строковое имя, которое будет применяться для установки свойства TableName нового объекта DataTable (если не указать имя таблицы, то адаптер данных назовет таблицу просто Table). В большинстве случаев имя, назначаемое DataTable, будет идентичным имени физической таблицы в реляционной базе данных; тем не менее, это не обязательно.

На заметку! Метод Fill() возвращает целое число, которое представляет количество строк, возвращенных запросом SQL.

Наконец, в методе Main() отсутствует явное открытие или закрытие подключения к базе данных. В методе Fill() любого адаптера данных заложена возможность открытия и закрытия подключения перед выходом из метода. Следовательно, когда объект DataSet передается методу PrintDataSet(), реализованному ранее в главе и приведенному здесь в справочных целях, работа осуществляется с локальной копией автономных данных, не требуя обращений к СУБД для извлечения данных:

```
static void PrintDataSet(DataSet ds)
{
  // Вывести имя DataSet и любые расширенные свойства.
  WriteLine($"DataSet is named: {ds.DataSetName}");
  foreach (DictionaryEntry de in ds.ExtendedProperties)
  {
    WriteLine($"Key = {de.Key}, Value = {de.Value}");
  }
  WriteLine();
  foreach (DataTable dt in ds.Tables)
  {
    WriteLine($"=> {dt.TableName} Table:");
    // Вывести имена столбцов.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write(dt.Columns[curCol].ColumnName + "\t");
    }
    WriteLine("\n---------------------------------");
    // Вывести содержимое DataTable.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
      for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
      {
        Write(dt.Rows[curRow][curCol].ToString().Trim() + "\t");
      }
      WriteLine();
    }
  }
}
```

Отображение имен из базы данных на дружественные к пользователю имена

Как упоминалось ранее, администраторы баз данных обычно создают такие имена таблиц и столбцов, которые редко бывают дружественными в отношении конечных пользователей (скажем, `au_id`, `au_fname`, `au_lname` и т.д.). Хорошая новость в том, что объекты адаптеров данных поддерживают внутреннюю строго типизированную коллекцию (по имени `DataTableMappingCollection`) объектов `System.Data.Common.DataTableMapping`, к которой можно получать доступ через свойство `TableMappings` объекта адаптера данных.

При желании с помощью этой коллекции объект `DataTable` можно информировать о том, какие *отображаемые имена* должны использоваться при выводе его содержимого. Предположим, что во время вывода вместо имени таблицы `Inventory` необходимо отображать `Current Inventory`, вместо имени столбца `CarId` — название `Car Id` (с пробелом), а вместо имени столбца `PetName` — строку `Name of Car`. Для этого перед вызовом метода `Fill()` объекта адаптера данных понадобится поместить показанный ниже код (а также импортировать пространство имен `System.Data.Common`, чтобы определение типа `DataTableMapping` стало доступным):

```
static void Main(string[] args)
{
  ...
  // Отобразить имена столбцов базы данных на дружественные к пользователю имена.
  DataTableMapping tableMapping =
    adapter.TableMappings.Add("Inventory", "Current Inventory");
```

```
  tableMapping.ColumnMappings.Add("CarId", "Car Id");
  tableMapping.ColumnMappings.Add("PetName", "Name of Car");
  dAdapt.Fill(ds, "Inventory");
  ...
}
```

Если снова запустить программу, то обнаружится, что метод `PrintDataSet()` теперь отображает дружественные имена объектов `DataTable` и `DataRow`, а не имена из схемы базы данных:

```
***** Fun with Data Adapters *****

DataSet is named: AutoLot

=> Current Inventory Table:
Car ID  Make    Color    Name of Car
---------------------------------
1       VW      Black    Zippy
2       Ford    Rust     Rusty
3       Saab    Black    Mel
4       Yugo    Yellow   Clunker
5       BMW     Black    Bimmer
6       BMW     Green    Hank
7       BMW     Pink     Pinkey
```

Исходный код. Проект `FillDataSetUsingSqlDataAdapter` доступен в подкаталоге `Chapter_22`.

Добавление функциональности автономного уровня в `AutoLotDAL.dll`

Чтобы продемонстрировать применение адаптера данных для передачи изменений в объекте `DataTable` базе данных, мы модифицируем созданную в главе 21 сборку `AutoLotDAL.dll`, включив в нее новое пространство имен (`AutoLotDisconnectedLayer`). Оно будет содержать новый класс `InventoryDALDC`, который использует адаптер данных для взаимодействия с объектом `DataTable`. Можно продолжить работу с проектом `AutoLotDAL`. В загружаемом коде примеров рассматриваемый пример находится в проекте `AutoLotDAL2`.

Определение начального класса

Добавим новую папку с помощью пункта меню Project⇨New Folder (Проект⇨Новая папка) и назначим ей имя `DisconnectedLayer`. Затем добавим в новую папку класс по имени `InventoryDALDC` (от **D**is**C**onnected), выбрав пункт меню Project⇨Add Class (Проект⇨Добавить класс). Далее снабдим тип класса в новом файле кода модификатором `public` и импортируем пространство имен `System.Data.SqlClient`.

В отличие от типа `InventoryDAL`, ориентированного на работу с подключением, новому классу не нужны специальные методы открытия/закрытия, поскольку адаптер данных обрабатывает все детали автоматически.

Начнем с добавления специального конструктора, который устанавливает закрытую переменную `string`, представляющую строку подключения. Кроме того, определим закрытую переменную-член `SqlDataAdapter`, которая будет конфигурироваться посредством вызова (пока еще не созданного) вспомогательного метода по имени `ConfigureAdapter()`, принимающего выходной параметр `SqlDataAdapter`:

```
namespace AutoLotDAL2.DisconnectedLayer
{
  public class InventoryDALDC
  {
    // Поля данных.
    private string _connectionString;
    private SqlDataAdapter _adapter = null;

    public InventoryDALDC(string connectionString)
    {
      _connectionString = connectionString;

      // Конфигурировать объект SqlDataAdapter.
      ConfigureAdapter(out _adapter);
    }

  }
}
```

Конфигурирование адаптера данных с использованием SqlCommandBuilder

Перед применением адаптера данных для модификации таблиц в `DataSet` сначала необходимо присвоить свойствам `UpdateCommand`, `DeleteCommand` и `InsertCommand` допустимые объекты команд (до этого указанные свойства возвращают ссылки `null`).

Ручное конфигурирование объектов команд для свойств `InsertCommand`, `UpdateCommand` и `DeleteCommand` может повлечь за собой написание значительного объема кода, особенно если используются параметризированные запросы. Вспомните из главы 21, что параметризированные запросы позволяют строить операторы SQL с применением набора объектов параметров. Таким образом, если избран долгий путь, то метод `ConfigureAdapter()` можно было бы реализовать так, чтобы в нем создавались три новых объекта `SqlCommand`, каждый из которых содержал бы набор объектов `SqlParameter`. Затем готовые объекты можно было бы присвоить свойствам `UpdateCommand`, `DeleteCommand` и `InsertCommand` адаптера.

Среда Visual Studio предлагает несколько визуальных конструкторов, которые позаботятся о создании этого утомительного кода. Визуальные конструкторы немного отличаются в зависимости от используемого API-интерфейса (например, Windows Forms, WPF или ASP.NET), но общая функциональность у них похожа. Примеры применения таких визуальных конструкторов будут неоднократно встречаться далее в книге, а в настоящей главе используются некоторые визуальные конструкторы Windows Forms.

В этот момент не придется писать многочисленные операторы кода для полного конфигурирования адаптера данных; взамен работу можно значительно сократить, реализовав метод `ConfigureAdapter()` следующим образом:

```
private void ConfigureAdapter(out SqlDataAdapter adapter)
{
  // Создать адаптер и настроить SelectCommand.
  adapter =
     new SqlDataAdapter("Select * From Inventory", _connectionString);

  // Динамически получить остальные объекты команд
  // во время выполнения, используя SqlCommandBuilder.
  var builder = new SqlCommandBuilder(adapter);
}
```

Чтобы упростить конструирование объектов адаптеров данных, каждый поставщик данных ADO.NET от компании Microsoft предоставляет тип *построителя команд*.

Тип SqlCommandBuilder автоматически генерирует значения для свойств InsertCommand, UpdateCommand и DeleteCommand объекта SqlDataAdapter на основе начального объекта SelectCommand. Преимущество в том, что отпадает необходимость вручную создавать все объекты SqlCommand и SqlParameter.

В этот момент возникает очевидный вопрос: благодаря чему построитель команд способен создавать эти объекты команд на лету? Краткий ответ таков: благодаря метаданным. Когда во время выполнения вызывается метод Update() адаптера данных, связанный построитель команд читает информацию схемы базы данных и автоматически генерирует внутренние объекты команд вставки, удаления и обновления.

Понятно, что такие действия требуют дополнительных обращений к удаленной базе данных, т.е. при многократном применении типа SqlCommandBuilder в одном приложении его производительность ухудшится. Здесь мы минимизируем негативный эффект за счет вызова метода ConfigureAdapter() во время конструирования объекта InventoryDALDC и сохранения сконфигурированного объекта SqlDataAdapter для использования на протяжении всего времени существования InventoryDALDC.

В приведенном ранее коде объект построителя команд (SqlCommandBuilder) не использовался помимо того, что его конструктору был передан в качестве параметра объект адаптера данных. Как ни странно, это все, что должно быть сделано (в минимальном варианте). "За кулисами" тип SqlCommandBuilder конфигурирует адаптер данных с помощью остальных объектов команд.

Хотя вам может нравиться идея получить кое-что просто так, вы должны иметь в виду, что построители команд обладают рядом серьезных ограничений. В частности, построитель команд способен автоматически генерировать команды SQL для их применения адаптером данных, если удовлетворены все перечисленные ниже условия:

- SQL-команда SELECT взаимодействует только с одной таблицей (т.е. никаких соединений);
- единственная таблица имеет первичный ключ;
- таблица должна иметь столбец или столбцы, представляющие первичный ключ, который включается в SQL-оператор SELECT.

Учитывая способ построения базы данных AutoLot, эти ограничения не доставляют никаких проблем. Однако в производственной базе данных вам придется хорошо обдумать, будет ли этот тип хоть как-то полезен (если нет, то вспомните, что Visual Studio автоматически генерирует большой объем необходимого кода, используя разнообразные инструменты визуального конструирования баз данных, как вы увидите позже).

Реализация метода GetAllInventory()

Наш адаптер данных готов к применению. Первый метод нового класса будет просто вызывать метод Fill() объекта SqlDataAdapter для получения объекта DataTable, представляющего все записи в таблице Inventory базы данных AutoLot:

```
public DataTable GetAllInventory()
{
  DataTable inv = new DataTable("Inventory");
  _adapter.Fill(inv);
  return inv;
}
```

Реализация метода UpdateInventory()

Метод UpdateInventory() очень прост:

```
public void UpdateInventory(DataTable modifiedTable)
{
  _adapter.Update(modifiedTable);
}
```

Здесь объект адаптера данных проверяет значение RowState у каждой строки входного объекта DataTable. В зависимости от его значения (RowState.Added, RowState.Deleted или RowState.Modified) "за кулисами" задействуется подходящий объект команды.

Установка номера версии

Итак, создание логики второй версии библиотеки доступа к данным завершено. Хотя делать это необязательно, установите номер версии библиотеки в 2.0.0.0 просто ради аккуратного ведения учета. Как объяснялось в главе 14, чтобы изменить версию сборки .NET, нужно дважды щелкнуть на значке Properties (Свойства) в окне Solution Explorer и затем щелкнуть на кнопке Assembly Information (Информация о сборке), расположенной на вкладке Application (Приложение). В открывшемся диалоговом окне укажите 2 для старшего номера версии сборки (за дополнительными сведениями обращайтесь в главу 14). Затем перекомпилируйте приложение для обновления манифеста сборки.

Исходный код. Проект AutoLotDAL2 доступен в подкаталоге Chapter_22.

Тестирование функциональности автономного уровня

В этот момент можно построить клиентское приложение для тестирования нового класса InventoryDALDC. Мы снова будем использовать API-интерфейс Windows Forms, чтобы отображать данные в графическом пользовательском интерфейсе. Создадим новый проект приложения Windows Forms по имени InventoryDALDisconnectedGUI, изменим в окне Solution Explorer первоначальное имя файла Form1.cs на MainForm.cs и установим свойство Text формы в Simple GUI Front End to the Inventory Table (Простое клиентское приложение с графическим пользовательским интерфейсом для таблицы Inventory). После создания проекта установим ссылку на обновленную сборку AutoLotDAL.dll (обязательно должна быть выбрана версия 2.0.0.0 сборки) и импортируем следующее пространство имен:

```
using AutoLotDAL2.DisconnectedLayer;
```

Главная форма приложения содержит элементы управления Label, DataGridView (по имени inventoryGrid) и Button (с именем btnUpdateInventory), который конфигурируется для обработки события Click. Вот определение формы:

```
public partial class MainForm : Form
{
  InventoryDALDC _dal = null;
  public MainForm()
  {
    InitializeComponent();
    string cnStr =
    @"Data Source=(local)\SQLEXPRESS2014;Initial Catalog=AutoLot;" +
     "Integrated Security=True;Pooling=False";
    // Создать объект доступа к данным.
    _dal = new InventoryDALDC(cnStr);
    // Заполнить элемент управления DataGridView.
    inventoryGrid.DataSource = _dal.GetAllInventory();
  }
```

```
private void btnUpdateInventory_Click(object sender, EventArgs e)
{
  // Получить модифицированные данные из DataGridView.
  DataTable changedDT = (DataTable)inventoryGrid.DataSource;

  try
  {
    // Зафиксировать изменения.
    _dal.UpdateInventory(changedDT);
    inventoryGrid.DataSource = _dal.GetAllInventory();
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
 }
}
```

После создания объекта `InventoryDALDC` можно привязать объект `DataTable`, возвращенный в результате вызова метода `GetAllInventory()`, к элементу управления `DataGridView`. Когда пользователь щелкает на кнопке Update Database (Обновить базу данных), из `DataGridView` извлекается модифицированный объект `DataTable` (с помощью свойства `DataSource`) и передается методу `UpdateInventory()`.

Вот и все! Запустите приложение, добавьте на сетке несколько новых строк и обновите/удалите ряд других строк. После щелчка на кнопке Update Database все изменения сохранятся в базе данных `AutoLot`. Из-за особенностей работы привязки данных в Windows Forms понадобится сбросить свойство `DataSource` сетки, чтобы изменения отобразились немедленно. При построении приложения с помощью Windows Presentation Foundation (WPF) вы заметите, что шаблон проектирования Observer (Наблюдатель) исправляет такое поведение.

Исходный код. Проект `InventoryDALDisconnectedGUI` доступен в подкаталоге `Chapter_22`.

Объекты `DataSet` с несколькими таблицами и отношения между данными

До сих пор все примеры, приведенные в главе, имели дело с единственным объектом `DataTable`. Тем не менее, настоящая мощь автономного уровня проявляется, когда объект `DataSet` содержит многочисленные взаимосвязанные объекты `DataTable`. В этом случае в коллекции `DataRelation` объекта `DataSet` можно определять любое количество объектов `DataRelation`, чтобы учесть все взаимозависимости между таблицами. На клиентском уровне такие объекты можно применять для навигации между табличными данными, не обращаясь к сети.

На заметку! Вместо обновления сборки `AutoLotDAL.dll`, чтобы учесть таблицы `Customers` и `Orders`, в этом примере вся логика доступа к данным выносится в новый проект Windows Forms. Однако смешивать логику пользовательского интерфейса с логикой обработки данных в производственном приложении не рекомендуется. В финальных примерах главы будут задействованы разнообразные инструменты проектирования баз данных для отделения кода пользовательского интерфейса от кода взаимодействия с данными.

Начнем пример с создания нового проекта приложения Windows Forms по имени `MultitabledDataSetApp`. Графический пользовательский интерфейс приложения довольно прост (обратите внимание, что имя первоначального файла `Form1.cs` изменено на `MainForm.cs`, а свойство `Text` формы установлено в `AutoLot Database Manipulator` (Средство манипулирования базой данных `AutoLot`)). На рис. 22.11 можно видеть три элемента управления `DataGridView` (`dataGridViewInventory`, `dataGridViewCustomers` и `dataGridViewOrders`), которые содержат данные, извлеченные из таблиц `Inventory`, `Orders` и `Customers` базы данных `AutoLot`. Кроме того, в интерфейсе предусмотрен элемент `Button` (`btnUpdateDatabase`), который позволяет отправить все внесенные в сетки изменения базе данных для обработки, используя объекты адаптеров данных.

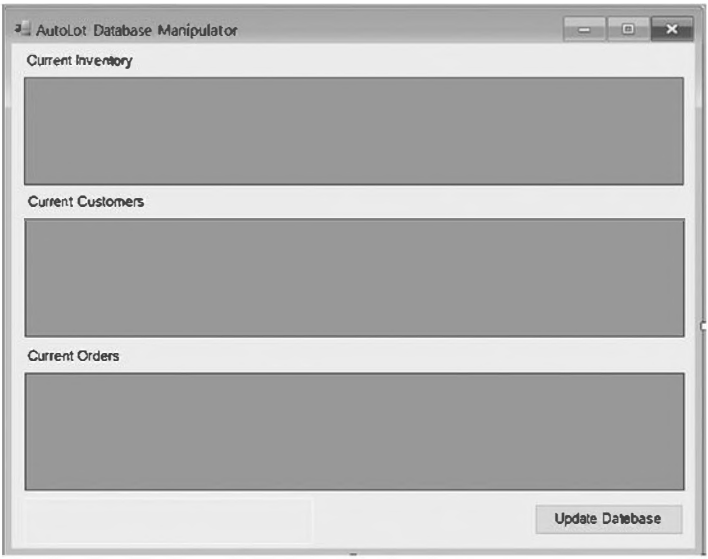


Рис. 22.11. Первоначальный пользовательский интерфейс будет отображать содержимое всех таблиц базы данных `AutoLot`

Подготовка адаптеров данных

Для как можно большего упрощения кода доступа к данным в классе `MainForm` будут применяться объекты построителей команд, которые автоматически сгенерируют команды SQL для всех трех объектов `SqlDataAdapter` (по одному на каждую таблицу). Ниже показаны начальные изменения типа, производного от `Form` (важно не забыть об импортировании пространства имен `System.Data.SqlClient`):

```
public partial class MainForm : Form
{
  // Объект DataSet уровня формы.
  private DataSet _autoLotDs = new DataSet("AutoLot");

  // Использовать построители команд для упрощения
  // конфигурирования адаптеров данных.
  private SqlCommandBuilder _sqlCbInventory;
  private SqlCommandBuilder _sqlCbCustomers;
  private SqlCommandBuilder _sqlCbOrders;

  // Адаптеры данных (для каждой таблицы).
  private SqlDataAdapter _invTableAdapter;
```

```
  private SqlDataAdapter _custTableAdapter;
  private SqlDataAdapter _ordersTableAdapter;
  // Строка подключения уровня формы.
  private string _connectionString;
  ...
}
```

Конструктор делает всю черновую работу по созданию переменных-членов, относящихся к данным, и заполнению `DataSet`. **В этом примере предполагается, что был создан файл** `App.config`, **который содержит корректную строку подключения (а также добавлена ссылка на сборку** `System.Configuration.dll` **и импортировано пространство имен** `System.Configuration`):

```
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
  </startup>
  <connectionStrings>
    <add name ="AutoLotSqlProvider" connectionString =
      "Data Source=(local)\SQLEXPRESS2014;
       Integrated Security=SSPI;Initial Catalog=AutoLot"
    />
  </connectionStrings>
</configuration>
```

Кроме того, обратите внимание на добавление вызова закрытого вспомогательного метода `BuildTableRelationship()`:

```
public MainForm()
{
  InitializeComponent();
  // Получить строку подключения.
  _connectionString =
    ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"]
      .ConnectionString;

  // Создать объекты адаптеров.
  _invTableAdapter = new SqlDataAdapter(
    "Select * from Inventory", _connectionString);
  _custTableAdapter = new SqlDataAdapter(
    "Select * from Customers", _connectionString);
  _ordersTableAdapter = new SqlDataAdapter(
    "Select * from Orders", _connectionString);

  // Автоматически сгенерировать команды.
  _sqlCbInventory = new SqlCommandBuilder(_invTableAdapter);
  _sqlCbOrders = new SqlCommandBuilder(_ordersTableAdapter);
  _sqlCbCustomers = new SqlCommandBuilder(_custTableAdapter);

  // Заполнить таблицы в DataSet.
  _invTableAdapter.Fill(_autoLotDs, "Inventory");
  _custTableAdapter.Fill(_autoLotDs, "Customers");
  _ordersTableAdapter.Fill(_autoLotDs, "Orders");

  // Построить отношения между таблицами.
  BuildTableRelationship();

  // Привязаться к сеткам.
  dataGridViewInventory.DataSource = _autoLotDs.Tables["Inventory"];
  dataGridViewCustomers.DataSource = _autoLotDs.Tables["Customers"];
  dataGridViewOrders.DataSource = _autoLotDs.Tables["Orders"];
}
```

Построение отношений между таблицами

Вспомогательный метод `BuildTableRelationship()` выполняет всю рутинную работу по добавлению двух объектов `DataRelation` в объект `autoLotDS`. Вспомните из главы 21, что в базе данных `AutoLot` имеется несколько отношений "родительский–дочерний", которые можно учесть с помощью следующего кода:

```
private void BuildTableRelationship()
{
  // Создать объект отношения между данными CustomerOrder.
  DataRelation dr = new DataRelation("CustomerOrder",
     _autoLotDs.Tables["Customers"].Columns["CustID"],
     _autoLotDs.Tables["Orders"].Columns["CustID"]);
  _autoLotDs.Relations.Add(dr);

  // Создать объект отношения между данными InventoryOrder.
  dr = new DataRelation("InventoryOrder",
     _autoLotDs.Tables["Inventory"].Columns["CarID"],
     _autoLotDs.Tables["Orders"].Columns["CarID"]);
  _autoLotDs.Relations.Add(dr);
}
```

Обратите внимание, что при создании объекта `DataRelation` в первом параметре указывается дружественный строковый псевдоним (вы вскоре узнаете, почему это удобно). Кроме того, устанавливаются ключи, используемые для построения самого отношения. В коде видно, что родительская таблица (второй параметр конструктора) указывается перед дочерней таблицей (третий параметр конструктора).

Обновление таблиц базы данных

Теперь, когда объект `DataSet` заполнен информацией из источника данных, каждым объектом `DataTable` можно манипулировать локально. Для этого запустите приложение и вставляйте, обновляйте или удаляйте значения в любом из трех элементов управления `DataGridView`. По готовности отправки изменений базе данных на обработку щелкните на кнопке Update Database. К этому времени вы должны легко отслеживать код в обработчике события `Click` указанной кнопки:

```
private void btnUpdateDatabase_Click(object sender, EventArgs e)
{
  try
  {
    _invTableAdapter.Update(_autoLotDs, "Inventory");
    _custTableAdapter.Update(_autoLotDs, "Customers");
    _ordersTableAdapter.Update(_autoLotDs, "Orders");
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

Запустите приложение и проведите различные обновления данных. При следующем запуске приложения вы должны обнаружить, что содержимое сеток отражает все последние изменения.

Навигация между связанными таблицами

Теперь давайте посмотрим, как объект `DataRelation` позволяет программно перемещаться между связанными таблицами. Расширим наш пользовательский интерфейс путем добавления в него нового элемента управления `Button` (`btnGetOrderInfo`), связанного с ним `TextBox` (`txtCustID`) и `Label` с описательным текстом (для улучшения внешнего вида эти элементы управления можно сгруппировать внутри `GroupBox`).

На рис. 22.12 показан возможный вид графического интерфейса приложения.

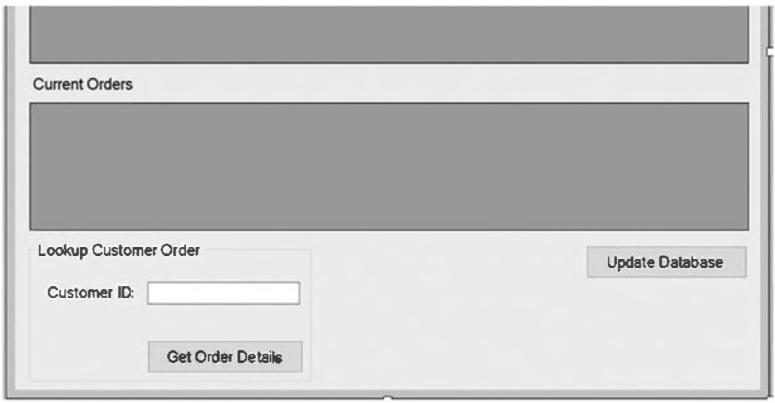


Рис. 22.12. Измененный пользовательский интерфейс предоставляет пользователю возможность поиска информации о заказах клиента

Обновленный пользовательский интерфейс позволяет пользователю ввести идентификатор клиента и извлечь связанную информацию о заказе этого клиента (имя, номер заказа и автомобиль). Информация форматируется в виде значения типа `string`, которое в конечном итоге отображается в окне сообщений. Взгляните на код обработчика события `Click` только что добавленной кнопки:

```
private void btnGetOrderInfo_Click(object sender, EventArgs e)
{
  string strOrderInfo = string.Empty;
  // Получить идентификатор клиента из текстового поля.
  int custID = int.Parse(txtCustID.Text);

  // На основе custID получить подходящую строку из таблицы Customers.
  var drsCust = _autoLotDs.Tables["Customers"].Select($"CustID =
{custID}");
  strOrderInfo +=
    $"Customer {drsCust[0]["CustID"]}:
    {drsCust[0]["FirstName"].ToString().Trim()}
    {drsCust[0]["LastName"].ToString().Trim()}\n";

  // Перейти из таблицы Customers в таблицу Orders.
  var drsOrder = drsCust[0].GetChildRows(_autoLotDs.Relations["CustomerOrder"]);

  // Проход в цикле по всем заказам этого клиента.
  foreach (DataRow order in drsOrder)
  {
    strOrderInfo += $"----\nOrder Number: {order["OrderID"]}\n";

    // Получить автомобиль, на который ссылается этот заказ.
    DataRow[] drsInv = order.GetParentRows(_autoLotDs.Relations[„InventoryOrder"]);

    // Получить информацию для (ОДНОГО) автомобиля из этого заказа.
    DataRow car = drsInv[0];
```

```
    strOrderInfo += $"Make: {car[„Make"]}\n";
    strOrderInfo += $"Color: {car[„Color"]}\n";
    strOrderInfo += $"Pet Name: {car[„PetName"]}\n";
  }
  MessageBox.Show(strOrderInfo, „Order Details");
}
```

На рис. 22.13 показан один из возможных результатов работы, когда задан идентификатор клиента 3 (ваш вывод может отличаться в зависимости от содержимого таблиц базы данных `AutoLot`).

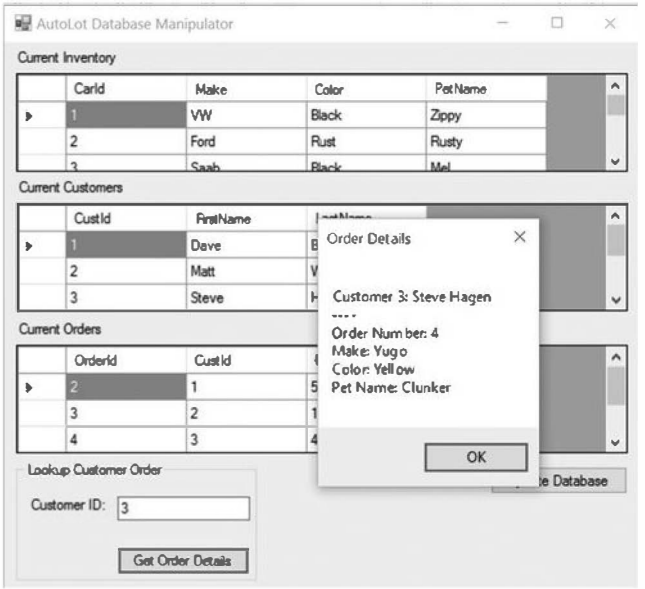


Рис. 22.13. Навигация с помощью отношений между данными

Предыдущий пример должен был убедить вас в полезности класса `DataSet`. Учитывая, что объект `DataSet` полностью отключен от лежащего в основе источника данных, вы можете работать с находящейся в памяти копией данных и переходить от таблицы к таблице для выполнения необходимых обновлений, удалений или вставок, не обращаясь к базе данных. Завершив модификацию, вы можете отправить изменения хранилищу данных на обработку. Конечным результатом является масштабируемое и надежное приложение.

Исходный код. Проект `MultitabledDataSetApp` доступен в подкаталоге `Chapter_22`.

Инструменты Windows Forms для визуального конструирования баз данных

Все приведенные до сих пор примеры предусматривали приличный объем непростой работы в том смысле, что всю логику доступа к данным приходилось писать вручную. В то время как значительная часть этого кода была вынесена в библиотеку .NET (`AutoLotDAL.dll`) для повторного использования в последующих главах книги, перед взаимодействием с реляционной базой данных по-прежнему приходится вручную

создавать разнообразные объекты имеющегося поставщика данных. Следующая задача заключается в исследовании инструментов визуального конструирования баз данных Windows Forms, которые могут создать за вас значительный объем кода доступа к данным.

Одним из способов применения этих интегрированных инструментов является использование визуальных конструкторов, поддерживаемых элементом управления `DataGridView` из Windows Forms. Проблема такого подхода в том, что инструменты визуального конструирования баз данных будут вставлять весь код доступа к данным прямо в кодовую базу графического пользовательского интерфейса! В идеале код, сгенерированный визуальным конструктором, лучше изолировать в выделенной библиотеке кода .NET, чтобы логику доступа к данным можно было многократно использовать во множестве проектов.

Тем не менее, полезно начать с выяснения того, как с помощью элемента `DataGridView` генерировать требуемый код доступа к данным, поскольку этот прием может быть полезен в небольших проектах и прототипах приложений. Затем вы узнаете, каким образом изолировать сгенерированный визуальным конструктором код в третьей версии библиотеки `AutoLotDAL.dll`.

Визуальное проектирование элемента управления `DataGridView`

С элементом управления `DataGridView` ассоциирован мастер, который может генерировать код доступа к данным. Начнем с создания нового проекта приложения Windows Forms по имени `DataGridViewDataDesigner`. В окне Solution Explorer переименуем первоначальную форму в `MainForm.cs`, установим ее свойство `Text` в `Windows Forms Data Wizards` (Мастера данных Windows Forms) и добавим к форме элемент управления `DataGridView` (по имени `inventoryDataGridView`). Когда элемент управления `DataGridView` выбран, справа от него должен открыться встроенный редактор (если он не открылся, нужно щелкнуть на кнопке с изображением треугольника в правом верхнем углу элемента управления). В раскрывающемся списке Choose Data Source (Выберите источник данных) щелкнем на ссылке Add Project Data Source (Добавить источник данных для проекта), как показано на рис. 22.14.

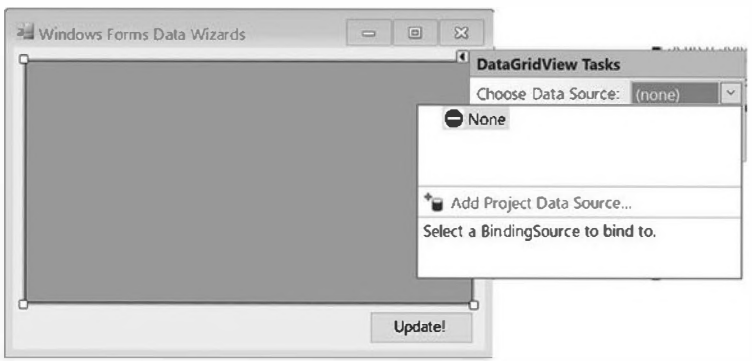


Рис. 22.14. Редактор `DataGridView`

Запустится мастер конфигурирования источников данных (Data Source Configuration Wizard). Он проведет через последовательность шагов, позволяющих выбрать и сконфигурировать источник данных, который затем будет привязан к `DataGridView`. На первом шаге мастер запрашивает тип источника данных, с которым необходимо взаимодействовать. Выберем вариант Database (База данных), как видно на рис. 22.15, и щелкнем на кнопке Next (Далее).

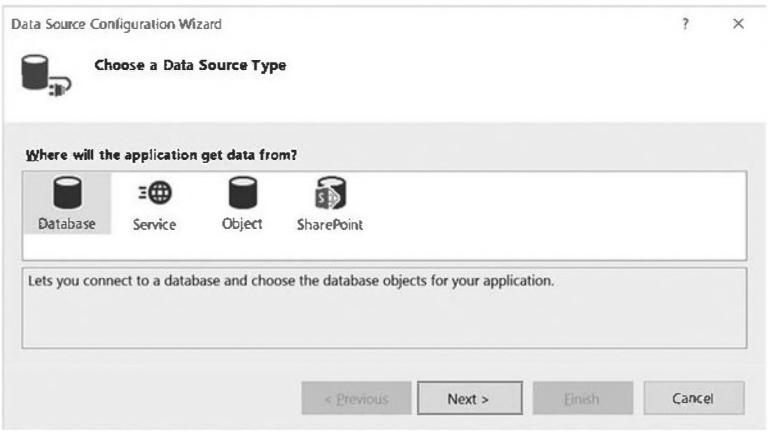


Рис. 22.15. Выбор типа источника данных

На следующем шаге (который будет слегка отличаться в зависимости от выбора, произведенного на первом шаге) мастер запрашивает тип применяемой модели базы данных. Модель базы данных `Dataset` будет видна, только если к проекту была добавлена инфраструктура Entity Framework. Выберем модель `Dataset` (рис. 22.16).

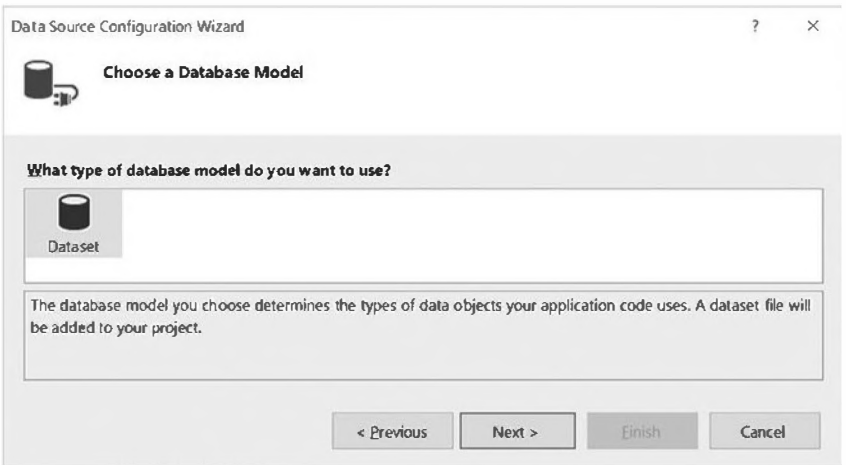


Рис. 22.16. Выбор модели базы данных

Следующий шаг мастера позволяет сконфигурировать подключение к базе данных. Если база данных в текущий момент добавлена в окно Server Explorer, то она должна автоматически отображаться в раскрывающемся списке. В противном случае (или если необходимо подключиться к базе данных, ранее не добавленной в Server Explorer) понадобится щелкнуть на кнопке New Connection (Новое подключение). Результат выбора локального экземпляра базы данных `AutoLot` представлен на рис. 22.17.

На следующем шаге мастера выдается запрос о том, нужно ли сохранить строку подключения в конфигурационном файле приложения (рис. 22.18). Отметим флажок, чтобы сохранить строку подключения, и щелкнем на кнопке Next.

На последнем шаге мастера выбираются объекты базы данных, которые будут учтены в автоматически сгенерированном классе `DataSet` и связанных адаптерах данных. Хотя можно было бы выбрать все объекты базы данных `AutoLot`, нас интересует только таблица `Inventory`.

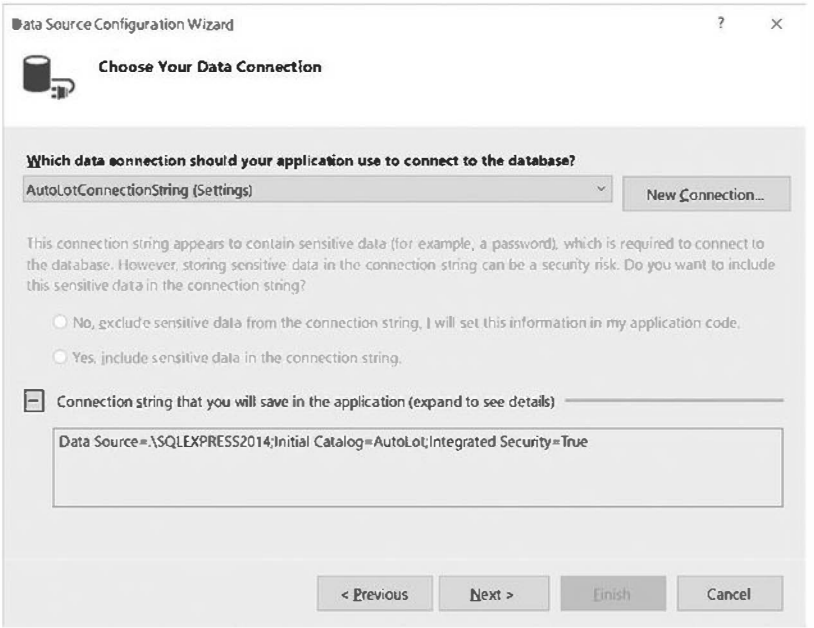


Рис. 22.17. Выбор базы данных

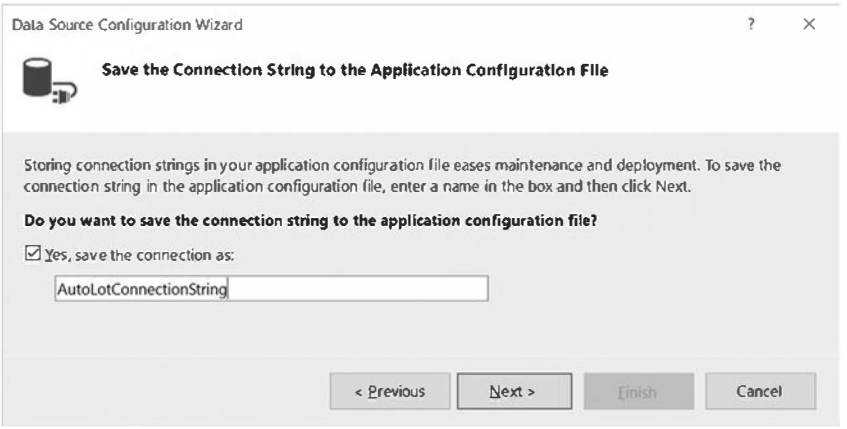


Рис. 22.18. Сохранение строки подключения в файле `App.config`

Изменим предлагаемое имя `DataSet` на `InventoryDataSet` (рис. 22.19), отметим флажок возле таблицы `Inventory` и щелкнем на кнопке Finish (Готово).

После этого визуальный конструктор обновится во многих отношениях. Самое заметное изменение связано с тем, что элемент управления `DataGridView` теперь отображает схему таблицы `Inventory`, что видно по заголовкам столбцов. Кроме того, в нижней части визуального конструктора формы (в области, называемой *лотком с компонентами*) находятся три компонента: `DataSet`, `BindingSource` и `TableAdapter` (рис. 22.20).

В этот момент можно запустить приложение и сетка заполнится записями из таблицы `Inventory`. Конечно же, никакой магии здесь нет. Просто IDE-среда создала за вас порядочный объем кода и настроила элемент управления типа сетки для последующего использования. Давайте проанализируем этот автоматически сгенерированный код.

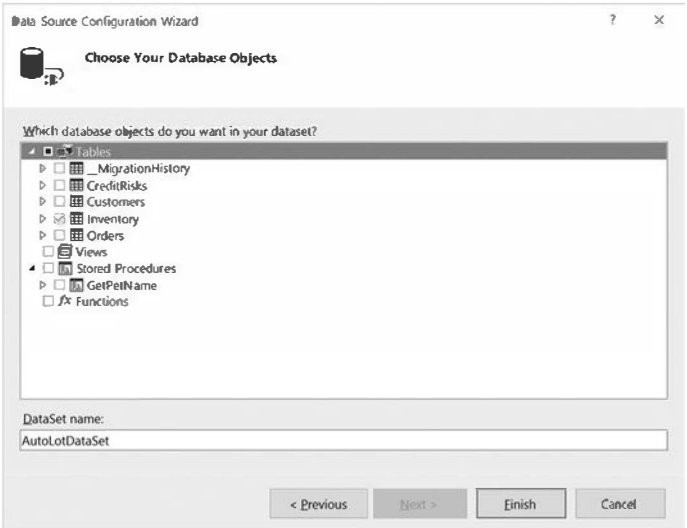


Рис. 22.19. Выбор таблицы `Inventory`

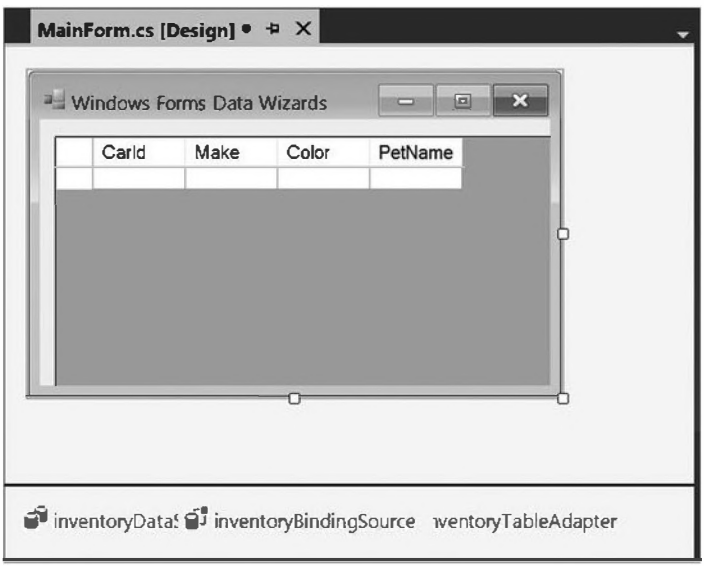


Рис. 22.20. Проект Windows Forms после завершения мастера конфигурирования источников данных

Сгенерированный файл `App.config`

В окне Solution Explorer можно заметить, что проект теперь содержит файл `App.config`, в котором имеется элемент `<connectionStrings>` с несколько необычным именем:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
  </configSections>
```

```
  <connectionStrings>
    <add name=
     "DataGridViewDataDesigner.Properties.Settings.AutoLotConnectionString"
        connectionString="Data Source=.\SQLEXPRESS2014;
                          Initial Catalog=AutoLot;Integrated Security=True"
        providerName="System.Data.SqlClient" />
  </connectionStrings>
    <startup>
      <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
    </startup>
</configuration>
```

Автоматически сгенерированный объект адаптера данных (о котором будет рассказано ниже) применяет длинное значение `"DataGridViewDataDesigner.Properties.Settings.AutoLotConnectionString"`.

Исследование строго типизированного класса `DataSet`

В дополнение к конфигурационному файлу мастер генерирует так называемый *строго типизированный класс* `DataSet`. Этим термином обозначается специальный класс, расширяющий `DataSet` и открывающий доступ к нескольким членам, которые позволяют взаимодействовать с базой данных, используя интуитивно понятную объектную модель. Например, строго типизированные объекты `DataSet` содержат свойства, которые отображаются непосредственно на имена таблиц базы данных. Таким образом, свойство `Inventory` можно применять для обращения к строкам и столбцам базы напрямую, не углубляясь в коллекцию таблиц с использованием свойства `Tables`.

Рис. 22.21. Мастер конфигурирования источников данных создал строго типизированный класс `DataSet`

Если вставить в проект новый файл диаграммы классов, то можно заметить, что мастер создал класс по имени `InventoryDataSet`. В нем определяется набор членов, из которых наиболее важным является свойство `Inventory` (рис. 22.21).

Двойной щелчок на файле `InventoryDataSet.xsd` в окне Solution Explorer приводит к загрузке визуального конструктора наборов данных Visual Studio (который более подробно рассматривается далее в главе). Если щелкнуть правой кнопкой мыши на поверхности этого конструктора и выбрать в контекстном меню пункт View Code (Просмотреть код), то отобразится следующее практически пустое определение частичного класса:

```
partial class InventoryDataSet {
}
```

При необходимости к этому определению частичного класса можно добавить специальные члены, однако реальное действие происходит в обслуживаемом конструктором файле `InventoryDataSet.Designer.cs`. Открыв этот файл в Solution Explorer, вы увидите, что `InventoryDataSet` расширяет родительский класс `DataSet`. Взгляните на показанный ниже фрагмент кода с комментариями, добавленными для ясности:

```
// Весь этот код сгенерирован конструктором!
public partial class InventoryDataSet : global::System.Data.DataSet
{
  // Переменная-член типа InventoryDataTable.
  private InventoryDataTable tableInventory;
```

```
  // Каждый конструктор вызывает вспомогательный метод по имени InitClass().
  public InventoryDataSet()
  {
    ...
    this.InitClass();
    ...
  }
  // Метод InitClass() подготавливает DataSet и добавляет
  // InventoryDataTable в коллекцию Tables.
  private void InitClass()
  {
    this.DataSetName = "InventoryDataSet";
    this.Prefix = "";
    this.Namespace = "http://tempuri.org/InventoryDataSet.xsd";
    this.EnforceConstraints = true;
    this.SchemaSerializationMode =
      global::System.Data.SchemaSerializationMode.IncludeSchema;
    this.tableInventory = new InventoryDataTable();
    base.Tables.Add(this.tableInventory);
  }
  // Свойство Inventory, предназначенное только для чтения,
  // возвращает переменную-член InventoryDataTable.
  public InventoryDataTable Inventory
  {
    get { return this.tableInventory; }
  }
}
```

Обратите внимание, что в показанном строго типизированном `DataSet` есть переменная-член, которая относится к *строго типизированному классу* `DataTable` — классу по имени `InventoryDataTable` в этом случае. Конструктор строго типизированного класса `DataSet` вызывает закрытый метод инициализации `InitClass()`, который добавляет экземпляр строго типизированного `DataTable` в коллекцию `Tables` объекта `DataSet`. Кроме того, реализация свойства `Inventory` возвращает переменную-член `InventoryDataTable`.

Исследование строго типизированного класса `DataTable`

Возвратимся к файлу диаграммы классов и раскроем узел Nested Types (Вложенные типы) для класса `InventoryDataSet`. Здесь будет находиться строго типизированный класс `DataTable` по имени `InventoryDataTable` и *строго типизированный* класс `DataRow` по имени `InventoryRow`.

В классе `InventoryDataTable` (который имеет тот же самый тип, что и рассмотренная переменная-член строго типизированного `DataSet`) определен набор свойств, основанных на именах столбцов физической таблицы `Inventory` (`CarIdColumn`, `ColorColumn`, `MakeColumn` и `PetNameColumn`), а также специальный индексатор и свойство `Count` для получения текущего количества записей.

Более интересно то, что в этом строго типизированном классе `DataTable` определены методы, которые позволяют вставлять, находить и удалять строки в таблице с применением строго типизированных членов (удобная альтернатива ручной навигации по индексаторам `Rows` и `Columns`). Например, метод `AddInventoryRow()` предназначен для добавления новой строки к находящейся в памяти таблице, `FindByCarId()` — для поиска в таблице по первичному ключу, а `RemoveInventoryRow()` — для удаления строки из строго типизированной таблицы (рис. 22.22).

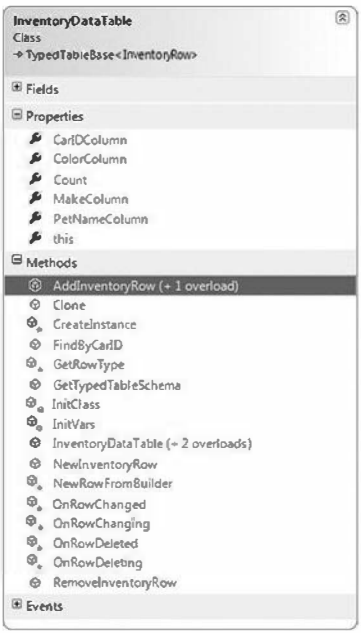


Рис. 22.22. Строго типизированный класс `DataTable`, вложенный в строго типизированный класс `DataSet`

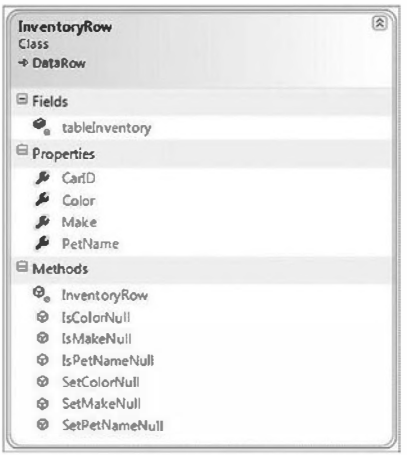


Рис. 22.23. Строго типизированный класс `DataRow`

Исследование строго типизированного класса `DataRow`

Строго типизированный класс `DataRow`, также вложенный внутрь строго типизированного класса `DataSet`, расширяет класс `DataRow` и открывает доступ к свойствам, которые отображаются прямо на схему таблицы `Inventory`. Кроме того, визуальный конструктор баз данных создал метод `IsPetNameNull()`, который проверяет, содержит ли этот столбец значение (рис. 22.23).

Исследование строго типизированного адаптера данных

Строгая типизация для автономных типов является серьезным преимуществом, которое дает использование мастера конфигурирования источников данных, т.к. создание этих классов вручную может оказаться утомительным (хотя и вполне посильным) занятием. Тот же самый мастер способен даже генерировать объект специального адаптера данных, который может заполнять и обновлять объекты `InventoryDataSet` и `InventoryDataTable` в строго типизированной манере. Найдите в окне визуального конструктора классов класс `InventoryTableAdapter` и просмотрите сгенерированные для него члены (рис. 22.24).

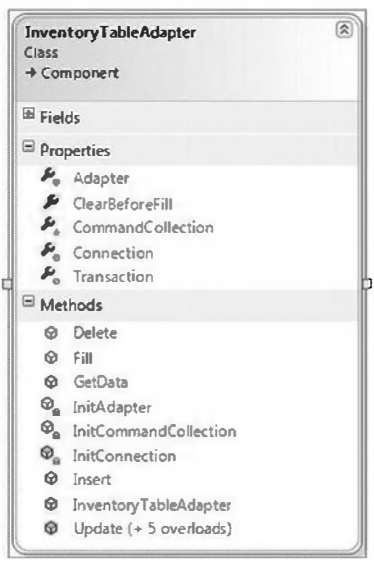


Рис. 22.24. Настроенный адаптер данных, который оперирует со строго типизированными классами `DataSet` и `DataTable`

Автоматически сгенерированный тип InventoryTableAdapter поддерживает коллекцию объектов SqlCommand (с доступом к ним через свойство CommandCollection), каждый из которых имеет полностью заполненный набор объектов SqlParameter. Вдобавок этот специальный адаптер данных предлагает набор свойств для извлечения лежащих в основе объектов подключения, транзакции и адаптера данных, а также свойство для получения массива, представляющего все типы команд.

Завершение приложения Windows Forms

Если внимательно изучить обработчик события Load в производном от формы типе (другими словами, если отобразить код MainForm.cs и найти метод MainForm_Load()), то обнаружится, что в самом начале вызывается метод Fill() специального адаптера данных таблицы с передачей ему специального объекта DataTable, поддерживаемого специальным DataSet:

```
private void MainForm_Load(object sender, EventArgs e)
{
  this.inventoryTableAdapter.Fill(this.inventoryDataSet.Inventory);
}
```

Тот же самый специальный объект адаптера данных можно применять для обновления сетки изменениями, произошедшими в данных. Добавим к пользовательскому интерфейсу формы элемент управления Button (по имени btnUpdateInventory). Затем создадим для него обработчик события Click со следующим кодом:

```
private void btnUpdateInventory_Click(object sender, EventArgs e)
{
  try
  {
    // Сохранить в базе данных изменения, внесенные в таблицу Inventory.
    this.inventoryTableAdapter.Update(this.inventoryDataSet.Inventory);
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
  // Получить актуальную копию данных для сетки.
  this.inventoryTableAdapter.Fill(this.inventoryDataSet.Inventory);
}
```

Снова запустите приложение; добавьте, удалите или обновите записи, отображаемые в сетке, а затем щелкните на кнопке Update Database (Обновить базу данных). При следующем запуске программы вы увидите, что все изменения были учтены.

Итак, рассмотренный пример продемонстрировал, насколько полезным может быть визуальный конструктор элемента управления DataGridView. Он позволяет работать со строго типизированными данными и генерирует большую часть логики, необходимой для взаимодействия с базой данных. Очевидная проблема заключается в том, что результирующий код тесно связан с окном, в котором он используется. В идеальном случае такая разновидность кода должна находиться в сборке AutoLotDAL.dll (или в какой-то другой библиотеке доступа к данным). Тем не менее, вас может интересовать, каким образом задействовать код, который генерируется мастером, ассоциированным с элементом управления DataGridView, в проекте библиотеки классов, поскольку по умолчанию визуальный конструктор форм в нем отсутствует.

Исходный код. Проект DataGridViewDataDesigner доступен в подкаталоге Chapter_22.

Изоляция строго типизированного кода работы с базой данных в библиотеке классов

К счастью, активизировать инструменты визуального проектирования данных среды Visual Studio можно в любой разновидности проекта (с пользовательским интерфейсом или нет) без необходимости копирования и вставки крупных фрагментов кода между проектами. Чтобы увидеть это в действии, мы добавим в `AutoLotDAL.dll` дополнительную функциональность. Можно продолжить работу с существующим проектом. В загружаемом коде примеров доступен отдельный проект под названием `AutoLotDAL3`.

Создадим внутри папки проекта новую папку `DataSets` и вставим в нее строго типизированный класс `DataSet` (по имени `AutoLotDataSet.xsd`) с применением пункта меню Project⇨Add New Item (Проект⇨Добавить новый элемент). Чтобы быстро найти тип элемента `DataSet`, в диалоговом окне New Item (Новый элемент) понадобится выбрать раздел Data (Данные), как показано на рис. 22.25.

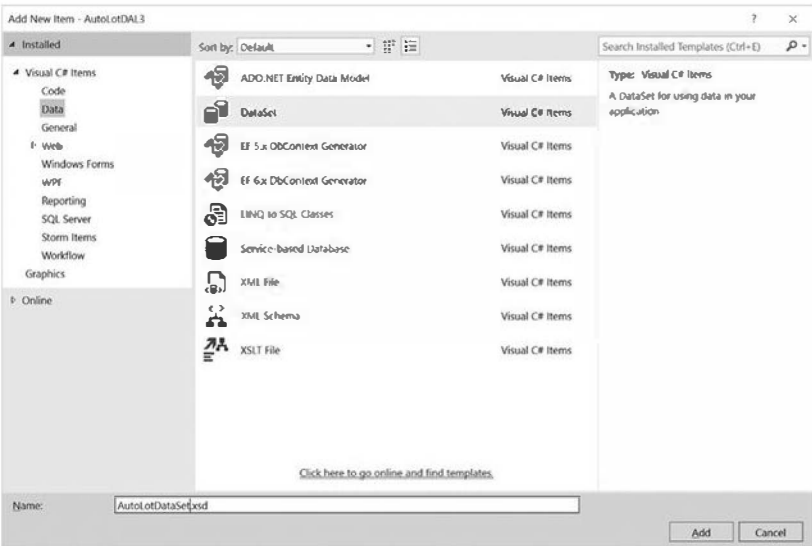


Рис. 22.25. Вставка нового строго типизированного класса `DataSet`

Откроется пустая поверхность визуального конструктора наборов данных. Теперь с помощью окна Server Explorer можно подключиться к нужной базе данных (подключение к `AutoLot` уже должно существовать) и перетащить на поверхность все таблицы и хранимые процедуры, которые требуются в `DataSet`. На рис. 22.26 видно, что учтены все специфические аспекты базы `AutoLot`, а отношения между таблицами реализованы автоматически (в этом примере таблица `CreditRisk` не перетаскивалась).

Просмотр сгенерированного кода

Визуальный конструктор наборов данных создал ту же самую разновидность кода, что и мастер `DataGridView` в предыдущем примере приложения Windows Forms. Однако на этот раз были задействованы таблицы `Inventory`, `Customers` и `Orders`, а также хранимая процедура `GetPetName`, потому сгенерированных классов получилось намного больше. По существу каждая таблица базы данных, помещенная на поверхность визуального конструктора, дала в результате классы `DataTable`, `DataRow` и адаптера данных, которые содержатся в строго типизированном `DataSet`.

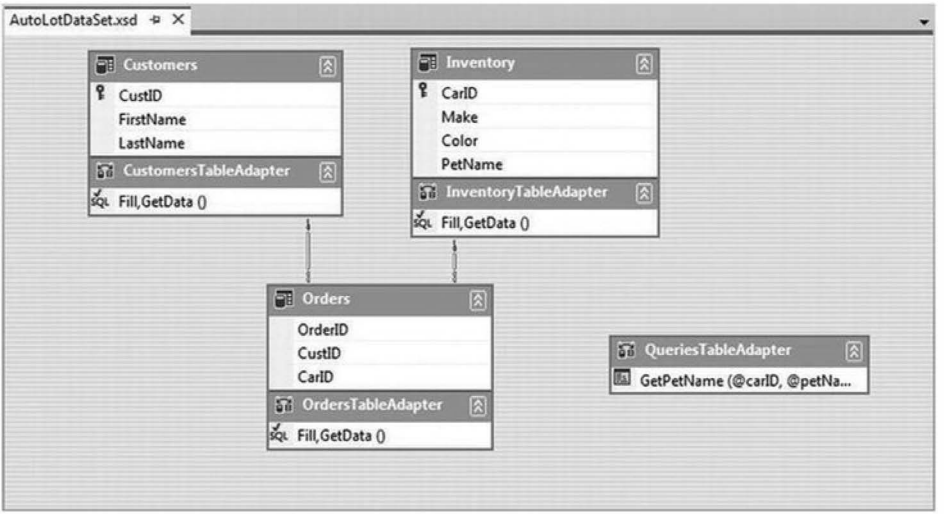


Рис. 22.26. Специальные строго типизированные классы, на этот раз внутри проекта библиотеки классов

Рис. 22.27. Автоматически сгенерированные строго типизированные классы для базы данных `AutoLot`

Строго типизированные классы `DataSet`, `DataTable` и `DataRow` будут помещены в корневое пространство имен проекта (`AutoLotDAL`). Специальные адаптеры таблиц будут находиться во вложенном пространстве имен. Просмотреть все сгенерированные типы проще всего с использованием окна Class View (Представление классов), которое открывается через меню View (Вид) в Visual Studio (рис. 22.27).

Ради завершенности можно открыть окно Properties (Свойства) в Visual Studio (см. главу 14) и изменить версию последней модификации сборки `AutoLotDAL.dll` на `3.0.0.0`.

Исходный код. Проект `AutoLotDAL3` доступен в подкаталоге `Chapter_22`.

Выборка данных с помощью сгенерированного кода

Полученные к настоящему моменту строго типизированные классы можно применять в любом приложении .NET, которому необходимо взаимодействовать с базой данных `AutoLot`.

Чтобы удостовериться в понимании всех основных механизмов, создадим консольное приложение по имени StronglyTypedDataSetConsoleClient. Добавим в него ссылку на последнюю версию сборки AutoLotDAL3.dll, импортируем в первоначальный файл кода C# пространства имен AutoLotDAL3.DataSets и AutoLotDAL3.DataSets.AutoLotDataSetTableAdapters, а также добавим оператор using static System.Console;.

Ниже приведен код метода Main(), в котором объект InventoryTableAdapter используется для выборки всех данных из таблицы Inventory. Обратите внимание, что здесь нет необходимости указывать строку подключения, т.к. эта информация теперь является частью строго типизированной объектной модели. После заполнения таблицы результаты выводятся с применением вспомогательного метода по имени PrintInventory(). Манипулировать строго типизированным DataTable можно точно так же, как это делается с "обычным" объектом DataTable, используя коллекции Rows и Columns.

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Strongly Typed DataSets *****\n");

    // Вызываемый код создает объект DataSet.
    var table = new AutoLotDataSet.InventoryDataTable();

    // Информировать адаптер о команде Select и подключении.
    var adapter = new InventoryTableAdapter();

    // Заполнить объект DataSet новой таблицей по имени Inventory.
    adapter.Fill(table);
    PrintInventory(table);
    Console.ReadLine();
  }

  static void PrintInventory(AutoLotDataSet.InventoryDataTable dt)
  {
    // Вывести имена столбцов.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write(dt.Columns[curCol].ColumnName + "\t");
    }
    WriteLine("\n----------------------------------");

    // Вывести содержимое DataTable.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
      for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
      {
        Write(dt.Rows[curRow][curCol] + "\t");
      }
      WriteLine();
    }
  }
}
```

Вставка данных с помощью сгенерированного кода

Предположим, что теперь нужно вставить новые записи с применением этой строго типизированной объектной модели. Показанный далее вспомогательный метод добавляет две новых строки в текущий объект InventoryDataTable и затем обновляет содержимое базы данных через адаптер данных. Первая строка добавляется вручную

за счет конфигурирования строго типизированного DataRow, а вторая — путем передачи необходимых данных столбцов, что позволяет создать DataRow автоматически "за кулисами".

```
public static void AddRecords(
  AutoLotDataSet.InventoryDataTable table,
  InventoryTableAdapter adapter)
{
  try
  {
    // Получить из таблицы новую строго типизированную строку.
    AutoLotDataSet.InventoryRow newRow = table.NewInventoryRow();

    // Заполнить строку данными.
    newRow.Color = "Purple";
    newRow.Make = "BMW";
    newRow.PetName = "Saku";

    // Вставить новую строку.
    table.AddInventoryRow(newRow);

    // Добавить еще одну строку, используя перегруженный метод добавления.
    table.AddInventoryRow("Yugo", "Green", "Zippy");

    // Обновить базу данных.
    adapter.Update(table);
  }
  catch (Exception ex)
  {
    WriteLine(ex.Message);
  }
}
```

Этот метод можно вызвать в Main(), в результате чего таблица базы данных обновится новыми записями:

```
static void Main(string[] args)
{
  ...
  // Добавить строки, обновить и вывести повторно.
  AddRecords(table, adapter);
  table.Clear();
  adapter.Fill(table);
  PrintInventory(table);
  Console.ReadLine();
}
```

Удаление данных с помощью сгенерированного кода

Удаление записей с помощью этой строго типизированной объектной модели также реализуется просто. Автоматически сгенерированный метод FindBy*XXXX*() (где *XXXX* — имя столбца первичного ключа) строго типизированного класса DataTable возвращает корректный (строго типизированный) объект DataRow, используя первичный ключ. Взгляните на вспомогательный метод, который удаляет две только что созданных записи:

```
private static void RemoveRecords(
  AutoLotDataSet.InventoryDataTable table, InventoryTableAdapter adapter)
{
```

```
    try
    {
      AutoLotDataSet.InventoryRow rowToDelete = table.FindByCarId(1);
      adapter.Delete(rowToDelete.CarId, rowToDelete.Make, rowToDelete.Color,
        rowToDelete.PetName);
      rowToDelete = table.FindByCarId(2);
      adapter.Delete(rowToDelete.CarId, rowToDelete.Make, rowToDelete.Color,
      rowToDelete.PetName);
    }
    catch (Exception ex)
    {
      WriteLine(ex.Message);
    }
  }
```

После вызова этого метода в `Main()` и повторного вывода содержимого таблицы вы должны заметить, что две ранее вставленных тестовых записи больше не отображаются.

На заметку! При желании этот пример можно сделать более гибким, запрашивая данные у пользователя с применением класса `Console`.

Вызов хранимой процедуры с помощью сгенерированного кода

Давайте рассмотрим еще один пример использования строго типизированной объектной модели. Мы создадим последний метод, который вызывает хранимую процедуру `GetPetName`. Когда строились адаптеры данных для базы `AutoLot`, был сгенерирован специальный класс по имени `QueriesTableAdapter`, который инкапсулирует процесс вызова хранимых процедур реляционной базы данных. Ниже приведен код финального вспомогательного метода, отображающего название указанного автомобиля:

```
public static void CallStoredProc()
{
  try
  {
    var queriesTableAdapter = new QueriesTableAdapter();
    Write("Enter ID of car to look up: ");
    string carID = ReadLine() ?? "0";
    string carName = "";
    queriesTableAdapter.GetPetName(int.Parse(carID), ref carName);
    WriteLine($"CarID {carID} has the name of {carName}");
  }
  catch(Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

К настоящему моменту вы знаете, как работать со строго типизированными классами базы данных и упаковывать их в отдельную библиотеку классов. В этой объектной модели вы обнаружите и другие аспекты, с которыми можно поэкспериментировать, но вам уже известно достаточно, чтобы самостоятельно разобраться в них. В завершение главы будет показано, как применять запросы LINQ к объекту `DataSet` из ADO.NET.

Исходный код. Проект `StronglyTypedDataSetConsoleClient` доступен в подкаталоге `Chapter_22`.

Программирование с помощью LINQ to DataSet

В этой главе вы узнали, что данными внутри объекта DataSet можно манипулировать тремя различными способами:

- с использованием коллекций Tables, Rows и Columns;
- с использованием объектов чтения таблиц данных;
- с использованием строго типизированных классов данных.

Разнообразные индексаторы типов DataSet и DataTable позволяют взаимодействовать с содержащимися данными в прямолинейной, но слабо типизированной манере. Вспомните, что такой подход требует трактовки данных как табличного блока ячеек, как показано в следующем примере:

```
static void PrintDataWithIndxers(DataTable dt)
{
  // Вывести содержимое DataTable.
  for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
  {
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write(dt.Rows[curRow][curCol + "\t");
    }
    WriteLine();
  }
}
```

Метод CreateDataReader() класса DataTable предлагает другой подход, при котором данные, содержащиеся в DataSet, трактуются как линейный набор строк, предназначенный для обработки последовательным образом. Это позволяет применять к автономному объекту DataSet модель программирования с подключенными объектами чтения данных.

```
static void PrintDataWithDataTableReader(DataTable dt)
{
  // Получить объект DataTableReader.
  DataTableReader dtReader = dt.CreateDataReader();
  while (dtReader.Read())
  {
    for (int i = 0; i < dtReader.FieldCount; i++)
    {
      Write($"{dtReader.GetValue(i)}\t");
    }
    WriteLine();
  }
  dtReader.Close();
}
```

И, наконец, можно использовать строго типизированный DataSet, чтобы получить кодовую базу, которая позволяет взаимодействовать с данными в объекте через свойства, отображающиеся на имена столбцов в реляционной базе данных. Строго типизированные объекты позволяют писать код следующего вида:

```
static void AddRowWithTypedDataSet()
{
  InventoryTableAdapter invDA = new InventoryTableAdapter();
```

```
  AutoLotDataSet.InventoryDataTable inv = invDA.GetData();
  inv.AddInventoryRow("Ford", "Yellow", "Sal");
  invDA.Update(inv);
}
```

Хотя все эти подходы имеют свои случаи применения, API-интерфейс LINQ to DataSet предоставляет еще одну возможность манипулирования данными DataSet с использованием выражений запросов LINQ.

На заметку! С помощью API-интерфейса LINQ to DataSet запросы LINQ применяются только к объектам DataSet, возвращаемым адаптерами данных, но это никак не связано с применением запросов LINQ напрямую к механизму базы данных. В главе 23 вы ознакомитесь с платформой ADO.NET Entity Framework, которая предлагает способ представления запросов SQL как запросов LINQ.

В первоначальном виде объект DataSet из ADO.NET (и связанные с ним типы, такие как DataTable и DataView) не имеет необходимой инфраструктуры, чтобы служить непосредственной целью для запроса LINQ. Например, показанный ниже метод (в котором используются типы из пространства имен AutoLotDisconnectedLayer) в результате приводит к ошибке на этапе компиляции:

```
static void LinqOverDataTable()
{
  // Получить объект DataTable с данными.
  InventoryDALDC dal = new InventoryDALDC(
    @"Data Source=(local)\SQLEXPRESS2014;Initial Catalog=AutoLot;
      Integrated Security=True");
  DataTable data = dal.GetAllInventory();

  // Применить запрос LINQ к DataSet?
  var moreData = from c in data where (int)c["CarID"] > 5 select c;
}
```

При попытке компиляции метода LinqOverDataTable() компилятор сообщит, что тип DataTable предоставляет *реализацию шаблона запросов*. Аналогично процессу применения запросов LINQ к объектам, которые не реализуют интерфейс IEnumerable<T>, объекты ADO.NET должны быть трансформированы в совместимый тип. Чтобы понять, как это делается, потребуется исследовать типы из сборки System.Data.DataSetExtensions.dll.

Роль библиотеки расширений DataSet

Сборка System.Data.DataSetExtensions.dll, ссылка на которую по умолчанию присутствует во всех проектах Visual Studio, дополняет пространство имен System.Data рядом новых типов (рис. 22.28).

Двумя наиболее полезными типами являются DataTableExtensions и DataRowExtensions. Эти классы расширяют функциональность типов DataTable и DataRow за счет использования набора расширяющих методов (см. главу 12). Еще один важный класс, TypedTableBaseExtensions, определяет расширяющие методы, которые можно применять к строго типизированным объектам DataSet, чтобы обеспечить поддержку LINQ для внутренних объектов DataTable. Все остальные члены сборки System.Data.DataSetExtensions.dll относятся к чистой инфраструктуре и не предназначены для непосредственного использования в кодовой базе.

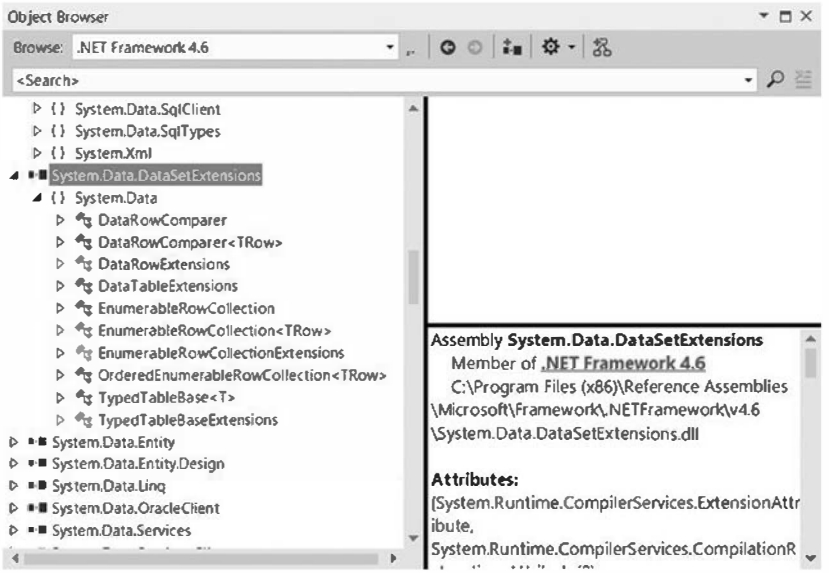


Рис. 22.28. Сборка `System.Data.DataSetExtensions.dll`

Получение объекта `DataTable`, совместимого с LINQ

А теперь давайте посмотрим, как работать с расширениями `DataSet`. Предположим, что имеется новый проект консольного приложения C# по имени `LinqToDataSetApp`. Добавим в него ссылку на последнюю версию (`3.0.0.0`) сборки `AutoLotDAL.dll` и модифицируем первоначальный файл кода следующим образом:

```
using System;
...
// Местоположение строго типизированных контейнеров данных.
using AutoLotDAL3.DataSets;

// Местоположение строго типизированных адаптеров данных.
using AutoLotDAL3.DataSets.AutoLotDataSetTableAdapters;
using static System.Console;

namespace LinqToDataSetApp
{
  class Program
  {
    static void Main(string[] args)
    {
      WriteLine("***** LINQ over DataSet *****\n");

      // Получить строго типизированный объект DataTable, содержащий
      // текущие данные таблицы Inventory из базы данных AutoLot.
      AutoLotDataSet dal = new AutoLotDataSet();
      InventoryTableAdapter tableAdapter = new InventoryTableAdapter();
      AutoLotDataSet.InventoryDataTable data = tableAdapter.GetData();

      // Вызвать описанные ниже методы.
      ReadLine();
    }
  }
}
```

Для преобразования объекта DataTable (включая строго типизированный DataTable) из ADO.NET в совместимый с LINQ объект должен быть вызван расширяющий метод AsEnumerable(), определенный в классе DataTableExtensions. Этот метод возвращает объект EnumerableRowCollection, который содержит коллекцию объектов DataRow.

После этого тип EnumerableRowCollection можно использовать для обработки каждой строки с помощью основного синтаксиса DataRow (например, синтаксиса индексатора). Рассмотрим показанный ниже новый метод класса Program, который принимает строго типизированный DataTable, получает перечислимую копию данных и выводит все значения CarId:

```
static void PrintAllCarIDs(DataTable data)
{
  // Получить перечислимую версию DataTable.
  EnumerableRowCollection enumData = data.AsEnumerable();
  // Вывести значения идентификаторов автомобилей.
  foreach (DataRow r in enumData)
  {
    WriteLine($"Car ID = {r["CarID"]}");
  }
}
```

Здесь запрос LINQ еще не применялся, но суть в том, что объект enumData теперь может служить целью выражения запроса LINQ. Обратите внимание, что объект EnumerableRowCollection содержит коллекцию объектов DataRow, т.к. для вывода значений столбца CarId к каждому подобъекту применяется индексатор типа.

В большинстве случаев нет необходимости объявлять переменную типа EnumerableRowCollection для хранения возвращаемого значения AsEnumerable(). Взамен этот метод можно вызывать внутри самого выражения запроса. Далее приведен код более интересного метода класса Program, который получает проекцию CarId и Make из всех записей в DataTable, представляющих автомобили черного цвета (если в вашей таблице Inventory нет записей для черных автомобилей, то соответствующим образом измените цвет в запросе LINQ):

```
static void ShowRedCars(DataTable data)
{
  // Спроецировать новый результирующий набор, содержащий
  // идентификатор/цвет для строк, в которых Color = Black.
  var cars = from car in data.AsEnumerable()
             where
               (string)car["Color"] == "Black"
             select new
             {
               ID = (int)car["CarID"],
               Make = (string)car["Make"]
             };
  WriteLine("Here are the red cars we have in stock:");
  foreach (var item in cars)
  {
    WriteLine($"-> CarID = {item.ID} is {item.Make}");
  }
}
```

Роль расширяющего метода DataRowExtensions.Field<T>()

Один из нежелательных аспектов текущего выражения запроса LINQ связан с тем, что для получения результирующего набора используются многочисленные операции приведения и индексаторы DataRow. Это может привести к генерации исключений времени выполнения, если будет предпринята попытка приведения к несовместимому типу данных. Чтобы привнести в запрос некоторую строгую типизацию, можно применить расширяющий метод Field<T>() типа DataRow. Такой прием позволяет увеличить безопасность к типам запроса, поскольку совместимость типов данных проверяется на этапе компиляции. Взгляните на следующее изменение:

```
var cars = from car in data.AsEnumerable()
  where
    car.Field<string>("Color") == "Black"
  select new
  {
    ID = car.Field<int>("CarID"),
    Make = car.Field<string>("Make")
  };
```

В этом случае можно вызвать метод Field<T>() и указать параметр типа для представления лежащего в основе типа данных столбца. В качестве аргумента методу передается имя столбца. Учитывая дополнительную проверку на этапе компиляции, при обработке элементов EnumerableRowCollection рекомендуется использовать метод Field<T>() (а не индексатор DataRow).

Помимо факта вызова метода AsEnumerable() общий формат запроса LINQ идентичен тому, что вы уже видели в главе 13, повторять здесь детали разнообразных операций LINQ не имеет смысла. Дополнительные примеры можно найти в разделе "LINQ to DataSet Examples" ("Примеры LINQ to DataSet") документации .NET Framework 4.6 SDK.

Заполнение новых объектов DataTable из запросов LINQ

Заполнить данными новый объект DataTable можно также на основе результатов запроса LINQ при условии, что в нем *не* применяются проекции. Если есть результирующий набор, тип которого может быть представлен как IEnumerable<T>, то на нем можно вызвать расширяющий метод CopyToDataTable<T>(), как показано ниже:

```
static void BuildDataTableFromQuery(DataTable data)
{
  var cars = from car in data.AsEnumerable()
             where car.Field<int>("CarID") > 5
             select car;

  // Использовать этот результирующий набор для построения
  // нового объекта DataTable.
  DataTable newTable = cars.CopyToDataTable();

  // Вывести содержимое DataTable.
  for (int curRow = 0; curRow < newTable.Rows.Count; curRow++)
  {
    for (int curCol = 0; curCol < newTable.Columns.Count; curCol++)
    {
      Write(newTable.Rows[curRow][curCol].ToString().Trim() + "\t");
    }
    WriteLine();
  }
}
```

На заметку! С использованием расширяющего метода `AsDataView<T>()` запрос LINQ можно также трансформировать в тип `DataView`.

Продемонстрированный прием может оказаться удобным, когда результат запроса LINQ нужно задействовать в качестве источника для операции привязки к данным. Вспомните, что элемент управления `DataGridView` в Windows Forms (а также элемент управления типа сетки в ASP.NET или WPF) поддерживает свойство по имени `DataSource`. Привязать результат запроса LINQ к сетке можно было бы следующим образом:

```
// Предположим, что myDataGrid - объект сетки графического
// пользовательского интерфейса.
myDataGrid.DataSource = (from car in data.AsEnumerable()
            where car.Field<int>("CarID") > 5
            select car).CopyToDataTable();
```

Итак, исследование автономного уровня ADO.NET завершено. С применением этого аспекта API-интерфейса можно извлекать данные из реляционной базы, обрабатывать их и возвращать обратно в базу, удерживая подключение к базе данных открытым на протяжении минимально возможного промежутка времени.

Исходный код. Проект `LinqToDataSetApp` доступен в подкаталоге `Chapter_22`.

Резюме

В этой главе подробно рассматривался автономный уровень ADO.NET. Как вы видели, центральной частью автономного уровня является тип `DataSet` — размещаемое в памяти представление любого числа таблиц и дополнительно любого количества отношений, ограничений и выражений. Преимущество установления отношений между локальными таблицами в том, что можно программно перемещаться между ними без подключения к удаленному хранилищу данных.

В главе также была исследована роль типа адаптера данных. С использованием этого типа (и связанных свойств `SelectCommand`, `InsertCommand`, `UpdateCommand` и `DeleteCommand`) адаптер может переносить изменения из `DataSet` в исходное хранилище данных. Кроме того, вы научились осуществлять навигацию по объектной модели `DataSet` прямолинейным ручным способом, а также с помощью строго типизированных объектов, которые обычно генерируют инструменты визуального конструктора наборов данных среды Visual Studio.

Наконец, вы взглянули на один из аспектов набора технологий LINQ под названием LINQ to DataSet. Он позволяет получать поддерживающую запросы копию `DataSet`, которую могут принимать правильно сформированные запросы LINQ.

ГЛАВА 23

ADO.NET, часть III: Entity Framework

В предшествующих двух главах исследовались фундаментальные программные модели ADO.NET — подключенный и автономный уровни. Эти подходы позволяли разработчикам приложений .NET взаимодействовать с реляционными данными (в относительно прямолинейной манере) с самого первого выпуска платформы. Однако в версии .NET 3.5 Service Pack 1 был введен новый компонент API-интерфейса ADO.NET под названием *Entity Framework* (*EF*).

На заметку! Несмотря на то что вышедшая первая версия была подвергнута широкой критике, команда разработчиков EF в Microsoft усердно трудилась над выпуском новых версий инфраструктуры. В настоящее время (на момент написания этой главы) среда Visual Studio 2015 поставляется с версией EF 6.1.3, которая предлагает множество средств и улучшений производительности по сравнению с более ранними выпусками. Версия Entity Framework 7 по-прежнему находится на этапе бета-тестирования (опять-таки, на момент написания главы). В итоге мы решили сосредоточить внимание в книге на версии EF 6.*x*, т.к. с версией EF 7 пока еще связано слишком много проблем.

Главная цель EF заключается в том, чтобы предоставить возможность взаимодействия с данными из реляционных баз данных с использованием объектной модели, которая отображается прямо на бизнес-объекты (или объекты предметной области) в приложении. Например, вместо трактовки пакета данных как коллекции строк и столбцов можно оперировать коллекцией строго типизированных объектов, которые называются *сущностями* (entity). Такие сущности также естественным образом поддерживают LINQ, и к ним можно применять запросы, используя ту же самую грамматику LINQ, которая была представлена в главе 12. Исполняющая среда EF самостоятельно транслирует запросы LINQ в подходящие запросы SQL.

В настоящей главе вы узнаете о доступе к данным с применением инфраструктуры Entity Framework. Вы научитесь создавать модель предметной области и отображать классы модели на базу данных, а также ознакомитесь с ролью класса `DbContext`. Кроме того, будет рассказано о навигационных свойствах, транзакциях и проверке параллелизма.

К концу главы мы построим финальную версию сборки `AutoLotDAL.dll`. Эта версия `AutoLotDAL.dll` будет использоваться в оставшихся главах книги, посвященных инфраструктурам Windows Presentation Foundation (WPF), ASP.NET Web Forms и ASP.NET MVC.

На заметку! Все версии Entity Framework (включая EF 6.x) поддерживают визуальный конструктор сущностей, предназначенный для создания XML-файла модели сущностных данных (EDMX). Начиная с версии 4.1, в инфраструктуре EF доступна поддержка простых старых объектов CLR (POCO) с применением приема, называемого Code First (Сначала код). Версия EF 7 будет поддерживать только парадигму Code First, но не визуальный конструктор сущностей. По этой причине основное внимание в главе уделяется Code First.

Роль Entity Framework

Подключенный и автономный уровни ADO.NET, описанные в главах 21 и 22, предоставляют фабрику, которая позволяет выбирать, вставлять, обновлять и удалять данные с помощью объектов подключений, команд, чтения данных, адаптеров данных и `DataSet`. Хотя все это замечательно, такие аспекты ADO.NET заставляют трактовать полученные данные в манере, которая тесно связана со схемой физической базы данных. Например, при использовании подключенного уровня обычно производится итерация по всем записям за счет указания объекту чтения данных имен столбцов. С другой стороны, в случае работы с автономным уровнем осуществляется обход коллекций строк и столбцов объекта `DataTable` внутри контейнера `DataSet`.

Если применяется автономный уровень в сочетании со строго типизированными классами `DataSet` или адаптерами данных, то получается программная абстракция, которая обеспечивает ряд преимуществ. Во-первых, строго типизированный класс `DataSet` открывает доступ к данным таблицы через свойства класса. Во-вторых, строго типизированный адаптер таблицы поддерживает методы, которые инкапсулируют конструирование лежащих в основе операторов SQL. Вспомните следующий метод `AddRecords()` из главы 22:

```
public static void AddRecords(AutoLotDataSet.InventoryDataTable table,
                               InventoryTableAdapter adapter)
{
  // Получить из таблицы новую строго типизированную строку.
  AutoLotDataSet.InventoryRow newRow = table.NewInventoryRow();

  // Заполнить строку данными.
  newRow.Color = "Purple";
  newRow.Make = "BMW";
  newRow.PetName = "Saku";

  // Вставить новую строку.
  table.AddInventoryRow(newRow);

  // Добавить еще одну строку, используя перегруженный метод добавления.
  table.AddInventoryRow("Yugo", "Green", "Zippy");

  // Обновить базу данных.
  adapter.Update(table);
}
```

Ситуация еще больше улучшится, если автономный уровень скомбинировать с LINQ to DataSet. В рассматриваемом примере запросы LINQ применяются к находящимся в памяти данным для получения нового результирующего набора, который затем можно дополнительно отобразить на автономный объект, такой как новый `DataTable`, `List<T>`, `Dictionary<K, V>` или массив данных:

```
static void BuildDataTableFromQuery(DataTable data)
{
  var cars = from car in data.AsEnumerable()
             where car.Field<int>("CarID") > 5 select car;
```

```
    // Использовать этот результирующий набор для построения нового объекта DataTable.
    DataTable newTable = cars.CopyToDataTable();

    // Работать с объектом DataTable...
}
```

Хотя интерфейс LINQ to DataSet удобен, вы должны помнить, что целью запроса LINQ являются *данные, возвращаемые из базы данных*, а не сам механизм базы данных. Это значит, что вы помещаете все данные из базы внутрь клиента и *затем* используете LINQ для получения подмножества данных либо их трансформирования. В идеале вы могли бы строить запрос LINQ, отправлять его напрямую механизму базы данных на обработку и получать обратно некоторые строго типизированные данные (именно это позволяет достичь инфраструктура ADO.NET Entity Framework).

Во время применения подключенного и автономного уровней ADO.NET вы всегда должны заботиться о физической структуре лежащей в основе базы данных. Необходимо знать схему каждой таблицы данных, писать потенциально сложные запросы SQL для взаимодействия с табличными данными и т.д. Это может требовать написания довольно громоздкого кода C#, т.к. сам язык C# не позволяет взаимодействовать непосредственно со схемой базы данных.

Хуже того, способ построения физической базы данных (администратором баз данных) полностью сосредоточен на таких конструкциях базы данных, как внешние ключи, представления и хранимые процедуры. Сложность создаваемых администратором баз данных может еще более возрастать, если администратор прилагает усилия по обеспечению безопасности и масштабируемости. В итоге также усложняется код C#, который приходится писать для взаимодействия с хранилищем данных.

Инфраструктура ADO.NET Entity Framework (EF) — это программная модель, которая пытается сократить разрыв между конструкциями базы данных и конструкциями объектно-ориентированного программирования. Используя EF, можно взаимодействовать с реляционными базами данных, (при желании) не сталкиваясь ни с одной строкой кода SQL. Исполняющая среда EF самостоятельно генерирует подходящие операторы SQL, когда вы применяете запросы LINQ к строго типизированным классам.

На заметку! *LINQ to Entities* — термин, описывающий действие по применению запросов LINQ к сущностным объектам ADO.NET.

Другой возможный подход состоит в том, чтобы вместо обновления базы данных посредством нахождения строки, ее обновления и отправки обратно на обработку с помощью пакета запросов SQL просто изменять свойства объекта и сохранять его состояние. В таком случае исполняющая среда EF снова будет обновлять базу данных автоматически.

В Microsoft считают инфраструктуру ADO.NET Entity Framework просто еще одним подходом к реализации API-интерфейса доступа к данным, который не планировался в качестве полной замены использованию ADO.NET прямо в коде C#. Однако, поработав какое-то время с EF, вы очень скоро можете обнаружить, что иметь дело с этой развитой объектной моделью предпочтительнее, чем с более примитивным миром запросов SQL и коллекций строк/столбцов.

Тем не менее, иногда в проектах .NET находится место для применения всех трех подходов из-за того, что модель EF может усложнять кодовую базу. Например, при построении внутреннего приложения, которому нужно взаимодействовать только с единственной таблицей базы данных, вы можете отдать предпочтение использованию подключенного уровня для обращения к пакету связанных хранимых процедур. Существенно выиграть от применения EF могут более крупные приложения, особенно

если команда разработчиков уверено работает с LINQ. Как и с любой новой технологией, понадобится выяснять, каким образом (и когда) имеет смысл выбирать ADO.NET EF для решения текущих задач.

На заметку! Вы можете вспомнить, что в версии .NET 3.5 появился API-интерфейс программирования для баз данных под названием LINQ to SQL. Концептуально этот API-интерфейс близок (а в плане конструкций программирования очень близок) к ADO.NET EF. Инфраструктура LINQ to SQL поддерживается в режиме сопровождения, а это означает, что она будет получать только исправления критических ошибок. Располагая приложением, в котором используется LINQ to SQL, имейте в виду, что официальная политика Microsoft предусматривает поддержку всего программного обеспечения на протяжении минимум десяти лет после прекращения его существования. Таким образом, хотя инфраструктура LINQ to SQL не будет удалена из машины какими-то средствами защиты программного обеспечения, официальное мнение в Microsoft заключается в том, что вы должны сконцентрировать усилия на EF, а не на LINQ to SQL. Это определенно того стоит.

Роль сущностей

Упомянутые ранее строго типизированные классы называются *сущностями*. Сущности являются концептуальной моделью физической базы данных, которая отображается на предметную область. Формально говоря, эта модель называется *моделью сущностных данных* (Entity Data Model — EDM). Модель EDM представляет собой набор классов клиентской стороны, которые отображаются на физическую базу данных посредством соглашений и конфигурации Entity Framework. Вы должны понимать, что сущности не обязаны напрямую отображаться на схему базы данных точно в соответствии с соглашениями об именовании. Вы вольны реструктурировать сущностные классы для удовлетворения своим нуждам, а исполняющая среда EF отобразит имеющиеся уникальные имена на корректную схему базы данных.

На заметку! В мире Code First большинство разработчиков ссылаются на классы POCO как на *модели*, а на коллекцию таких классов — как на *модель*. После создания экземпляров классов модели с помощью данных из хранилища на них ссылаются как на *сущности*. В действительности перечисленные термины чаще всего применяются взаимозаменяемо.

Например, вспомним ранее созданную простую таблицу `Inventory` из базы данных `AutoLot` со схемой, показанной на рис. 23.1.

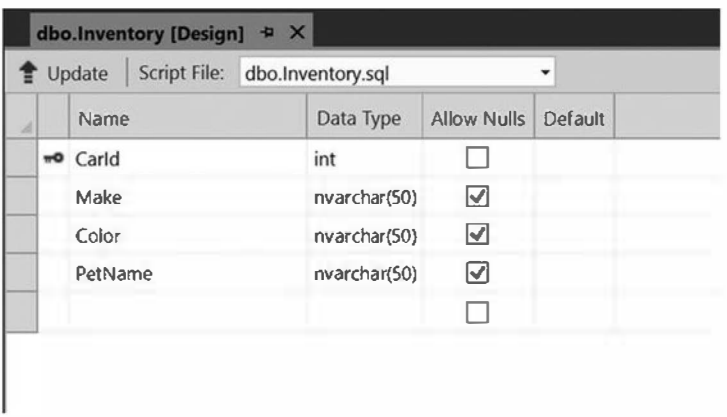


Рис. 23.1. Структура таблицы `Inventory` базы данных `AutoLot`

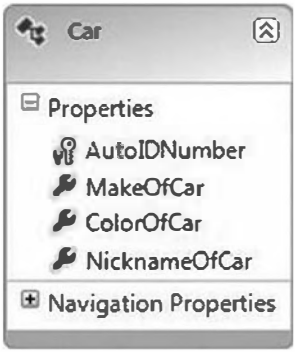


Рис. 23.2. Модель `Car` — это клиентская форма схемы `Inventory`

Если вы создадите модель для таблицы `Inventory` базы данных `AutoLot` (вскоре вы увидите, как это делается), то по умолчанию класс будет называться `Inventory`. Однако класс можно переименовать в `Car` и определить уникально именованные свойства по своему выбору, которые будут отображены на столбцы таблицы `Inventory`. Слабая привязка подобного рода означает возможность формирования сущностей так, чтобы они близко моделировали предметную область. На рис. 23.2 представлен пример сущностного класса.

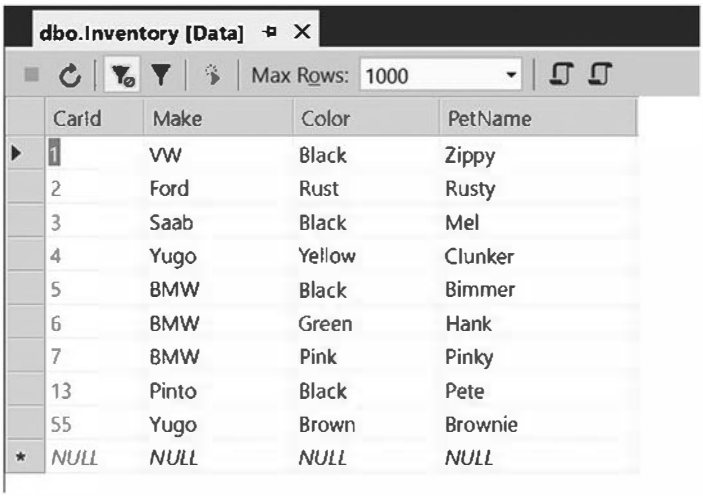
На заметку! Во многих случаях классы модели будут именоваться идентично связанным с ними таблицам базы данных. Тем не менее, не забывайте, что вы всегда можете изменить форму модели для соответствия конкретной ситуации.

Вскоре мы построим полноценный пример с использованием EF. Однако пока что взгляните на следующий класс `Program`, в котором применяется класс модели `Car` (и связанный контекстный класс по имени `AutoLotEntities`) для добавления новой строки в таблицу `Inventory` базы данных `AutoLot`. Контекстный класс `AutoLotEntities` является производным от `DbContext`. Его работа заключается во взаимодействии с физической базой данных (детали рассматриваются ниже).

```
class Program
{
  static void Main(string[] args)
  {
    // Строка подключения читается из
    // конфигурационного файла автоматически.
    using (AutoLotEntities context = new AutoLotEntities())
    {
      // Добавить новую строку в таблицу Inventory, используя нашу модель.
      context.Cars.Add(new Car() { ColorOfCar = "Black",
                                   MakeOfCar = "Pinto",
                                   NicknameOfCar = "Pete" });
      context.SaveChanges();
    }
  }
}
```

Исполняющая среда EF отвечает за получение клиентского представления таблицы `Inventory` (в данном случае это класс по имени `Car`) и отображение его обратно на корректные столбцы таблицы `Inventory`. Обратите внимание, что здесь нет никаких следов SQL-оператора `INSERT`. Мы просто добавляем новый объект `Car` в коллекцию, поддерживаемую подходящее именованным свойством `Cars` контекстного объекта, и сохраняем изменения. Естественно, заглянув в таблицу данных с использованием окна Server Explorer в Visual Studio, можно увидеть новую запись (рис. 23.3).

В приведенном примере нет никакой магии. "За кулисами" производится подключение к базе данных, генерируется подходящий оператор SQL и т.д. Преимущество инфраструктуры EF в том, что все детали обрабатываются без вашего участия. Теперь давайте рассмотрим базовые службы EF, которые делают это возможным.



CarId	Make	Color	PetName
1	VW	Black	Zippy
2	Ford	Rust	Rusty
3	Saab	Black	Mel
4	Yugo	Yellow	Clunker
5	BMW	Black	Bimmer
6	BMW	Green	Hank
7	BMW	Pink	Pinky
13	Pinto	Black	Pete
55	Yugo	Brown	Brownie
NULL	*NULL*	*NULL*	*NULL*

Рис. 23.3. Результат сохранения контекста

Строительные блоки Entity Framework

API-интерфейс EF находится на вершине существующей инфраструктуры ADO.NET, которая была описана в предшествующих двух главах. Подобно любому взаимодействию ADO.NET, для взаимодействия с хранилищем данных в Entity Framework применяется поставщик данных ADO.NET. Тем не менее, поставщик данных должен быть модернизирован, чтобы поддерживать новый набор служб, прежде чем он сможет работать с API-интерфейсом EF. Как и можно было ожидать, поставщик данных Microsoft SQL Server был обновлен необходимой инфраструктурой, которая учитывается при использовании сборки `System.Data.Entity.dll`.

На заметку! Многие СУБД от сторонних производителей (скажем, Oracle и MySQL) предлагают совместимые с EF поставщики данных. Детальную информацию можно узнать у производителя СУБД или просмотреть список известных поставщиков данных ADO.NET по адресу `https://msdn.microsoft.com/en-us/library/dd363565.aspx`. Инфраструктура EF построена поверх модели поставщиков данных ADO.NET и будет работать с любым источником данных, для которого доступен поставщик данных.

В дополнение к добавлению необходимых аспектов к поставщику данных Microsoft SQL Server сборка `System.Data.Entity.dll` содержит разнообразные пространства имен, которые предназначены для самих служб EF. В настоящий момент мы сосредоточим внимание на двух основных частях API-интерфейса EF — классе `DbContext` и производном контексте, специфичном для модели.

Роль класса `DbContext`

Класс `DbContext` представляет комбинацию шаблонов проектирования Unit of Work (Единица работы) и Repository (Хранилище), которые могут применяться для запрашивания базы данных и объединения изменений, допускающих запись обратно в базу данных в виде одиночной единицы работы. Класс `DbContext` предоставляет дочерним классам набор основных служб, включая возможность сохранения всех изменений (результатом чего является обновление базы данных), настройку строки подключения, удаление объектов, вызов хранимых процедур и обработку других фундаментальных деталей. В табл. 23.1 описаны некоторые часто используемые члены класса `DbContext`.

Таблица 23.1. Часто используемые члены DbContext

Член	Описание
`DbContext()`	Конструктор, применяемый по умолчанию в производном контекстном классе. В строковом параметре указывается либо имя базы данных, либо строка подключения, хранящаяся в файле `*.config`
`Entry()` `Entry<TEntity>()`	Извлекает объект `System.Data.Entity.Infrastructure.DbEntityEntry`, который предоставляет доступ к информации и возможность выполнения действий над сущностью
`GetValidationErrors()`	Проверяет достоверность обработанных сущностей и возвращает коллекцию объектов `System.Data.Entity.Validation.DbEntityValidationResult`
`SaveChanges()` `SaveChangesAsync()`	Сохраняет в базе данных все изменения, внесенные в этот контекст. Возвращает количество затронутых сущностей
`Configuration`	Предоставляет доступ к свойствам конфигурации контекста
`Database`	Предлагает механизм для создания/удаления/проверки существования лежащей в основе базы данных, для выполнения хранимых процедур и низкоуровневых операторов SQL в отношении внутреннего хранилища данных, а также для доступа к функциональности транзакций

Класс `DbContext` также реализует интерфейс `IObjectContextAdapter`, поэтому доступна также любая функциональность, имеющаяся в классе `ObjectContext`. Несмотря на то что класс `DbContext` позаботится о большинстве ваших потребностей, есть два события, которые могут оказаться исключительно удобными, как вы увидите позже в главе. События класса `DbContext` описаны в табл. 23.2.

Таблица 23.2. События DbContext

Событие	Описание
`ObjectMaterialized`	Инициируется при создании нового сущностного объекта из хранилища данных как части операции запроса или загрузки
`SavingChanges`	Происходит, когда изменения сохраняются в хранилище данных, но перед тем, как данные станут постоянными

Роль производных контекстных классов

Как упоминалось ранее, класс `DbContext` предоставляет основную функциональность во время работы со средством Code First инфраструктуры EF. В наших проектах будет создаваться производный от `DbContext` класс для специфической предметной области. В конструкторе понадобится передать конструктору базового класса имя строки подключения для этого контекстного класса, как показано ниже:

```
public class AutoLotEntities : DbContext
{
  public AutoLotEntities() base("name=AutoLotConnection")
  {
  }
  protected override void Dispose(bool disposing)
  {
  }
}
```

Роль DbSet<T>

Чтобы добавить в контекст таблицы, для каждой таблицы в объектной модели понадобится предусмотреть свойство типа DbSet<T>. Для включения ленивой загрузки свойства в контексте должны быть виртуальными, например:

```
public virtual DbSet<CreditRisk> CreditRisks { get; set; }
public virtual DbSet<Customer> Customers { get; set; }
public virtual DbSet<Inventory> Inventory { get; set; }
public virtual DbSet<Order> Orders { get; set; }
```

Каждый объект DbSet<T> предлагает для каждой коллекции несколько основных служб, таких как создание, удаление и нахождение записей в представленной таблице. Некоторые основные члены класса DbSet<T> описаны в табл. 23.3.

Таблица 23.3. Избранные основные члены DbSet<T>

Член	Описание
Add() AddRange()	Позволяют вставлять в коллекцию новый объект (или диапазон объектов). Объекты получают состояние EntityState.Added и будут вставлены в базу данных в результате вызова метода SaveChanges() (или SaveChangesAsync()) на объекте DbContext
Attach()	Ассоциирует объект с DbContext. Широко применяется в автономных приложениях вроде ASP.NET/MVC
Create() Create<T>()	Создает новый экземпляр указанного сущностного типа
Find() FindAsync()	Находит строку данных по первичному ключу и возвращает объект, представляющий эту строку
Remove() RemoveRange()	Помечает объект (или диапазон объектов) для последующего удаления
SqlQuery()	Создает низкоуровневый запрос SQL, который возвратит сущности в этом наборе

Добравшись до нужного свойства контекста, вы можете обратиться к любому члену DbSet<T>. Еще раз взгляните на пример кода, приведенный в начале главы:

```
using (AutoLotEntities context = new AutoLotEntities())
{
  // Добавить новую строку в таблицу Inventory, используя нашу модель.
  context.Cars.Add(new Car() { ColorOfCar = "Black",
                               MakeOfCar = "Pinto",
                               NicknameOfCar = "Pete" });
  context.SaveChanges();
}
```

Здесь AutoLotEntities — производный контекстный класс. Свойство Cars предоставляет доступ к переменной DbSet<Car>. Эта ссылка используется для вставки нового сущностного объекта Car и сообщения экземпляру DbContext о необходимости сохранения всех изменений в базе данных.

Целью запросов LINQ to Entities обычно является объект DbSet<T>; по существу класс DbSet<T> поддерживает те же самые расширяющие методы, о которых вы узнали в главе 12, такие как ForEach(), Select() и All(). Более того, DbSet<T> получает приличный объем функциональности от своего непосредственного родительского класса, DbQuery<T>, который представляет строго типизированный запрос LINQ (или Entity SQL).

Empty Code First Model или Code First from Database

Прежде чем приступить к исследованию первого примера Entity Framework, необходимо обсудить еще один момент. Модель данных Entity Framework можно построить с нуля или воспроизвести ее из существующей базы данных (подобно применению визуального конструктора Entity Framework). Оба приема будут рассмотрены в последующих разделах.

Поддержка транзакций

Все версии EF помещают каждый вызов `SaveChanges()`/`SaveChangesAsync()` внутрь транзакции. Уровень изоляции таких автоматических транзакций будет таким же, как стандартный уровень изоляции для базы данных (`READ COMMITTED` в случае SQL Server). При необходимости можно обеспечить более высокий контроль над поддержкой транзакций в EF. Дополнительные сведения ищите по адресу:

`https://msdn.microsoft.com/en-us/data/dn456843.aspx`.

На заметку! Хотя в книге данный факт не раскрывается, но вызовы метода `ExecuteSqlCommand()` объекта базы данных `DbContext` для выполнения операторов SQL теперь помещаются внутрь неявной транзакции. Это нововведение версии EF 6.

Состояние сущности

Экземпляр класса `DbContext` автоматически отслеживает состояние любого объекта в рамках своего контекста. В предшествующих примерах модификации производятся внутри оператора `using`, но любые изменения в данных будут отслеживаться и сохраняться при вызове метода `SaveChanges()` класса `AutoLotEntities`. Возможные значения состояния объекта описаны в табл. 23.4.

Таблица 23.4. Значения перечисления `EntityState`

Значение	Описание
`Detached`	Объект существует, но не отслеживается. Сущность находится в этом состоянии немедленно после создания и перед добавлением к объектному контексту
`Unchanged`	Объект не изменялся с момента присоединения к контексту или с момента последнего вызова метода `SaveChanges()`
`Added`	Объект является новым и добавлен в объектный контекст, но метод `SaveChanges()` не вызывался
`Deleted`	Объект был удален из объектного контекста, но пока еще не удален из хранилища данных
`Modified`	Одно из скалярных свойств объекта было модифицировано, но метод `SaveChanges()` не вызывался

Для проверки состояния объекта используется следующий код:

```
EntityState state = context.Entry(entity).State;
```

Обычно переживать по поводу состояния объектов нет никакой необходимости. Однако в случае удаления объекта можно установить состояние объекта в `EntityState.Deleted` и выполнить обращение к базе данных. Позже в главе будет показано, как это делается.

Прием Code First from Database

Теперь, когда вы лучше понимаете, что собой представляет инфраструктура ADO.NET Entity Framework и как она работает на высоком уровне, самое время рассмотреть первый полноценный пример. Мы разработаем простое консольное приложение с применением приема Code First from Database для создания классов модели, представляющих существующую базу данных `AutoLot`, которая была построена в главах 21 и 22. Внутри консольного приложения мы напишем код, выполняющий типичные операции создания, чтения, обновления и удаления (create, read, update, delete — CRUD), после чего исследуем все, что было автоматически сгенерировано.

Генерация модели

Начнем с создания нового проекта консольного приложения по имени `AutoLotConsoleApp`. С помощью пункта меню Project⇨New Folder (Проект⇨Новая папка) добавим в проект папку и назовем ее `EF`. При выбранной папке `EF` выберем пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент), удостоверимся в том, что выделен узел Data (Данные), и вставим элемент ADO.NET Entity Data Model (Модель сущностных данных ADO.NET) по имени `AutoLotEntities` (рис. 23.4).

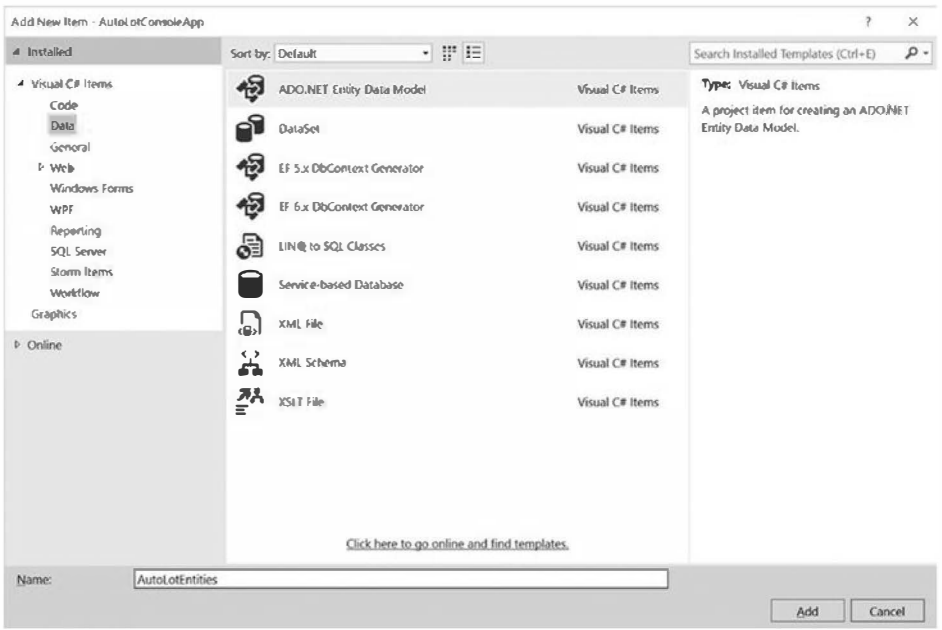


Рис. 23.4. Вставка в проект нового элемента ADO.NET EDM

Щелчок на кнопке Add (Добавить) приводит к запуску мастера создания модели сущностных данных (Entity Data Model Wizard). На первом шаге мастер позволяет выбрать вариант генерации модели EDM с использованием визуального конструктора Entity Framework (из существующей базы данных или с созданием пустой поверхности конструктора) либо с применением приема Code First (из существующей базы данных или с созданием пустого объекта `DbContext`). Выберите вариант Code First from database (Code First из базы данных) и щелкните на кнопке Next (Далее), как показано на рис. 23.5.

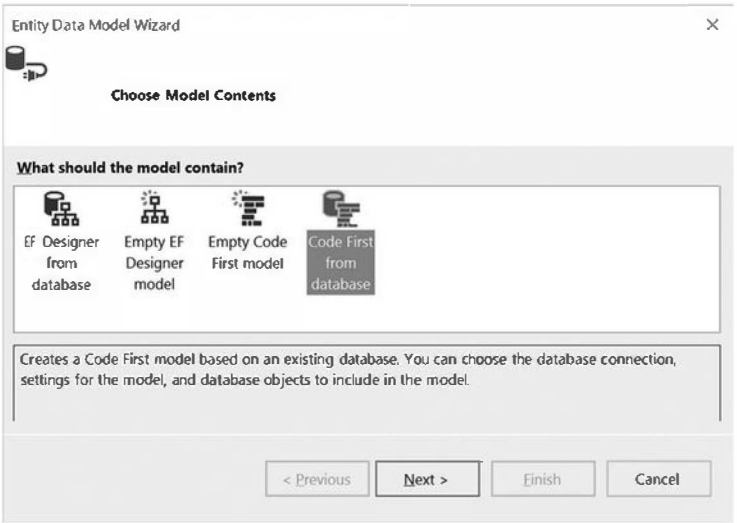


Рис. 23.5. Генерация модели EDM из существующей базы данных

На втором шаге мастера можно выбрать базу данных. Если подключение к базе данных в окне Server Explorer среды Visual Studio уже имеется, то оно будет присутствовать в раскрывающемся списке. В противном случае понадобится щелкнуть на кнопке New Connection (Создать подключение). В любом случае затем следует выбрать базу данных `AutoLot`, отметить флажок Save connection settings in App.config as (Сохранить настройки подключения в файле `App.config` как) и указать в поле под флажком имя `AutoLotConnection` (рис. 23.6).

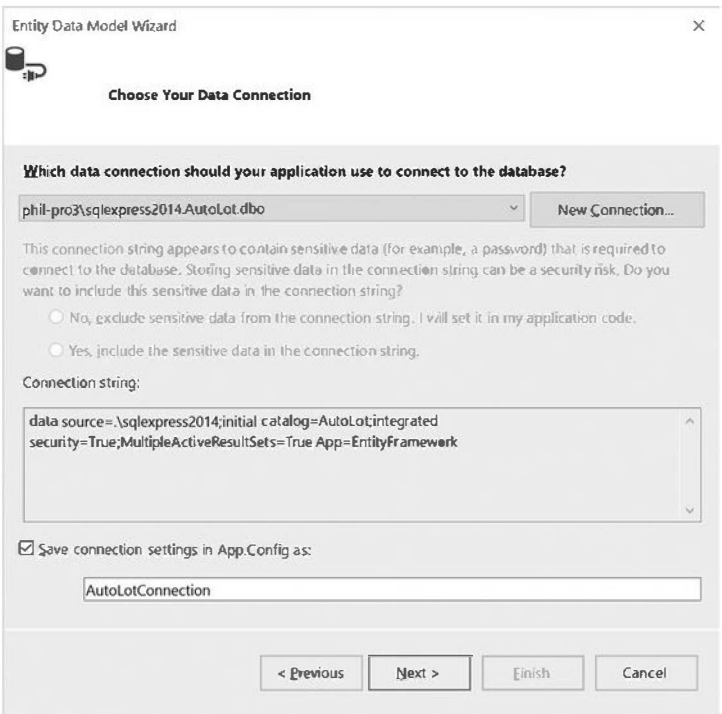


Рис. 23.6. Выбор базы данных для генерации модели

Прежде чем щелкать на кнопке Next, полезно взглянуть на формат строки подключения:

```
Data source= .\SQLEXPRESS2014;Initial Catalog=AutoLot;Integrated Security=
True;MultipleActiveResultSets=true;App=EntityFramework
```

Строка подключения очень похожа на ту, что использовалась в главах 21 и 22, с добавлением пары "имя-значение" `App=EntityFramework`. Здесь `App` является сокращением для имени приложения, которое можно применять при поиске и устранении проблем с базой данных SQL Server.

На последнем шаге мастера можно выбрать элементы из базы данных, которые должны использоваться при генерации модели EDM. Выберем все таблицы приложения, удостоверившись в том, что флажок для таблицы `sysdiagrams` не отмечен (если эта таблица существует в базе данных). Результирующее окно выглядит подобным показанному на рис. 23.7.

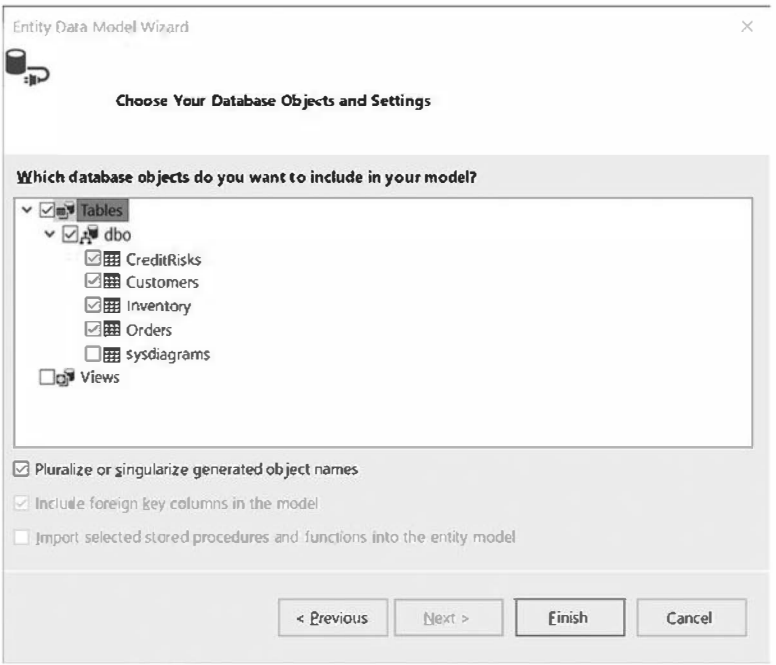


Рис. 23.7. Выбор элементов базы данных

Теперь можно щелкнуть на кнопке Finish (Готово), чтобы сгенерировать модель EDM.

Что мы получили?

После завершения работы с мастером в проекте появится несколько новых классов: по одному классу для каждой таблицы, выбранной в мастере, и еще один класс по имени `AutoLotEntities` (то же самое имя, которое вводилось на первом шаге мастера). По умолчанию имена сущностей будут основаны на именах исходных объектов базы данных; тем не менее, вспомните, что имена сущностей в концептуальной модели могут быть какими угодно. Изменить имя сущности, а также имена свойств сущности, можно за счет применения специальных атрибутов .NET, которые называются *аннотациями данных*. Аннотации данных будут использоваться для внесения ряда модификаций в модель.

На заметку! Другой способ конфигурирования классов модели и их свойств для отображения на базу данных предлагает интерфейс Fluent API. Все, что можно делать с помощью аннотаций данных, можно также реализовать в коде посредством Fluent API. Из-за ограничений в объеме в этой главе будут подробно раскрыты только аннотации данных и приведены лишь краткие упоминания об интерфейсе Fluent API.

Откроем код класса `Inventory`. Первое, что можно заметить — это последовательность атрибутов, декорирующих класс и его свойства. Они называются *аннотациями данных* и предназначены для инструктирования инфраструктуры EF о том, как строить таблицы и свойства при генерации базы данных. Аннотации данных также указывают EF способ отображения данных из базы на классы модели. На уровне классов атрибут `Table` определяет, на какую таблицу отображается конкретный класс. На уровне свойств применяются два атрибута. Первый из них — атрибут `Key`, который задает первичный ключ для таблицы. Второй атрибут, `StringLength`, указывает длину строки при генерации кода DDL для поля. Как вы увидите далее в главе, этот атрибут также используется во время проверки достоверности.

На заметку! Имеются также два атрибута `SuppressMessage`. Они инструктируют статические анализаторы, такие как FXCop и новые анализаторы кода Roslyn, о необходимости отключения специфических правил, перечисленных в конструкторе.

```
[Table("Inventory")]
public partial class Inventory
{
  [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
    "CA2214:DoNotCallOverridableMethodsInConstructors")]
  public Inventory()
  {
    Orders = new HashSet<Order>();
  }
  [Key]
  public int CarId { get; set; }
  [StringLength(50)]
  public string Make { get; set; }
  [StringLength(50)]
  public string Color { get; set; }
  [StringLength(50)]
  public string PetName { get; set; }
  [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
    "CA2227:CollectionPropertiesShouldBeReadOnly")]
  public virtual ICollection<Order> Orders { get; set; }
}
```

Можно также заметить, что класс `Inventory` содержит коллекцию объектов `Order`. В результате получается отношение "один ко многим" между `Inventory` и `Order`. На другом конце этого отношения в классе `Order` указаны свойства `CarId` и `Car`:

```
public partial class Order
{
  public int OrderId { get; set; }
  public int CustId { get; set; }
  public int CarId { get; set; }
  public virtual Customer Customer { get; set; }
  public virtual Inventory Inventory { get; set; }
}
```

Теперь откроем код класса AutoLotEntities. Этот класс является производным от DbContext и содержит свойство типа DbSet<TEntity> для каждой таблицы, которая была выбрана в мастере. В нем переопределен метод OnModelCreating(), чтобы с помощью Fluent API определить отношения между Customer и Orders, а также между Orders и Inventory.

```
public partial class AutoLotEntities : DbContext
{
  public AutoLotEntities() : base("name=AutoLotConnection")
  {
  }
  public virtual DbSet<CreditRisk> CreditRisks { get; set; }
  public virtual DbSet<Customer> Customers { get; set; }
  public virtual DbSet<Inventory> Inventories { get; set; }
  public virtual DbSet<Order> Orders { get; set; }
  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Customer>()
      .HasMany(e => e.Orders)
      .WithRequired(e => e.Customer)
      .WillCascadeOnDelete(false);
    modelBuilder.Entity<Inventory>()
      .HasMany(e => e.Orders)
      .WithRequired(e => e.Inventory)
      .WillCascadeOnDelete(false);
  }
}
```

Наконец, откроем файл App.config. Здесь легко заметить новый раздел конфигурации (по имени entityFramework) и строку подключения, сгенерированную мастером. Большую ее часть можно проигнорировать, но если вы измените базу данных, то имейте в виду, что единственной информацией, которую понадобится модифицировать, будут значения строки подключения AutoLotConnection (это имя было указано в мастере).

```
<configuration>
  <configSections>
    <!-- Дополнительные сведения о конфигурации Entity Framework
      доступны по адресу http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.
Internal.ConfigFile.EntityFrameworkSection, EntityFramework,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
  </startup>
  <entityFramework>
    <defaultConnectionFactory type=
"System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFramework" />
    <providers>
      <provider invariantName="System.Data.SqlClient" type=
"System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
  <connectionStrings>
    <add name="AutoLotConnection" connectionString=
"data source=.\SQLEXPRESS2014;initial catalog=AutoLot;
```

```
integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Изменение стандартных отображений

Как обсуждалось в предыдущем разделе, атрибут `[Table("Inventory")]` указывает, что класс отображается на таблицу `Inventory`. При наличии этого атрибута имя класса можно изменять любым желаемым образом. Изменим имя класса (и конструктора) на `Car`. В дополнение к атрибуту `Table` инфраструктура EF также работает с атрибутом `Column`. Добавив к свойству `PetName` атрибут `[Column("PetName")]`, можно изменить имя этого свойства на `CarNickName`. Вот как должен выглядеть итоговый код:

```
[Table("Inventory")]
public partial class Car
{
  public Car()
  {
    Orders = new HashSet<Order>();
  }
  [StringLength(50), Column("PetName")]
  public string CarNickName { get; set; }
  // Для краткости оставшийся код не показан.
}
```

В случае изменения какого-то имени без применения инструментов рефакторинга Visual Studio код приложения не скомпилируется. Если так и вышло, то необходимо открыть код класса `Order` и изменить тип и имя свойства `Inventory` на `Car`. Вот результирующий код:

```
public partial class Order
{
  public virtual Car Car { get; set; }
  // Для краткости оставшийся код не показан.
}
```

Последнее изменение потребуется внести в класс `AutoLotEntities` (если только не использовались инструменты рефакторинга, доступные в версии Visual Studio 2015). Откроем код класса `AutoLotEntities` и изменим два вхождения `Inventory` на `Car`, а `DbSet<Car>` — на `Cars`. Ниже показан обновленный код:

```
public partial class AutoLotEntities : DbContext
{
  public AutoLotEntities() : base("name=AutoLotConnection")
  {
  }
  // Для краткости оставшийся код не показан.
  public virtual DbSet<Car> Cars { get; set; }
  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Car>()
      .HasMany(e => e.Orders)
      .WithRequired(e => e.Car
      .WillCascadeOnDelete(false);
    // Для краткости оставшийся код не показан.
  }
}
```

На заметку! Команда разработчиков EF выпустила набор мощных инструментов (под названием Entity Framework Power Tools) для Visual Studio. Эти инструменты предлагают разнообразные способы составления диаграмм моделей EDM, а также дополнительную функциональность. К сожалению, на момент написания главы указанные инструменты не были доступны для Visual Studio 2015. Следите за обновлениями по адресу `https://visualstudiogallery.msdn.microsoft.com/72a60b14-1581-4b9b-89f2-846072eff19d/`.

Добавление сгенерированных классов модели

Все классы, сгенерированные визуальным конструктором, объявлены с ключевым словом `partial`, которое позволяет разнести реализацию класса по нескольким файлам кода C#. Это особенно удобно при работе с программной моделью EF, поскольку означает возможность добавления к сущностным классам дополнительных методов, которые помогают лучше моделировать предметную область.

Например, можно переопределить метод `ToString()` сущностного класса `Car`, чтобы он возвращал состояние сущности в виде удобно сформатированной строки. Если добавить код переопределенного метода `ToString()` к сгенерированному классу, то возникнет риск утери этого специального кода каждый раз, когда классы модели будут генерироваться повторно. Взамен определим следующий частичный класс в новом файле по имени `CarPartial.cs`:

```
public partial class Car
{
  public override string ToString()
  {
    // Поскольку столбец PetName может быть пустым,
    // предоставить стандартное имя **No Name**.
    return $"{this.CarNickName ?? "**No Name**"} is a {this.Color}
      {this.Make} with ID {this.CarId}.";
  }
}
```

Использование классов модели в коде

Теперь, располагая классами модели, можно написать код, который взаимодействует с ними и, следовательно, с базой данных. Начнем с добавления к классу `Program` оператора `using` для пространства имен `AutoLotConsoleApp.EF` и оператора `using static System.Console;`.

Вставка записи

Добавим вспомогательный метод по имени `AddNewRecord()`, вызываемый из `Main()`, который будет вставлять новую запись в таблицу `Inventory`:

```
private static int AddNewRecord()
{
  // Добавить запись в таблицу Inventory базы данных AutoLot.
  using (var context = new AutoLotEntities())
  {
    try
    {
      // В целях тестирования жестко закодировать данные для новой записи.
      var car = new Car() { Make = "Yugo", Color = "Brown", CarNickName="Brownie"};
      context.Cars.Add(car);
      context.SaveChanges();
```

```
      // В случае успешного сохранения EF заполняет поле идентификатора
      // значением, сгенерированным базой данных.
      return car.CarId;
    }
    catch(Exception ex)
    {
      WriteLine(ex.InnerException.Message);
      return 0;
    }
  }
}
```

В этом коде применяется метод `Add()` класса `DbSet<Car>`. Метод `Add()` принимает объект типа `Car` и добавляет его в коллекцию `Cars` контекстного класса `AutoLotEntities`. Вставка нового объекта `Car` посредством метода `Add()` класса `DbSet<Car>` и последующий вызов метода `SaveChanges()` на контекстном объекте приводит к выполнению SQL-оператора `INSERT`. Когда вызывается метод `SaveChanges()`, все ожидающие обработки изменения (в этом случае только одна дополнительная запись) сохраняются в базе данных. Если никаких ошибок не произошло, то запись добавится, а объект `Car` обновится любыми сгенерированными базой данных значениями (`CarId` в этом случае).

Чтобы увидеть все это в действии, модифицируем метод `Main()`, как показано ниже:

```
static void Main(string[] args)
{
  WriteLine("***** Fun with ADO.NET EF *****\n");
  int carId = AddNewRecord();
  WriteLine(carId);
  ReadLine();
}
```

На консоль действительно выводится значение поля `CarId` новой записи. Важно отметить, что хотя для получения сгенерированных базой данных идентификаторов ничего специального предпринимать не требуется, чтобы извлечь значение `CarId`, инфраструктура EF выполняет оператор `SELECT`. В большинстве приложений это не особо крупная проблема, но важно о ней помнить, когда возникают сложности с производительностью или масштабированием и приходится заняться сокращением числа вызовов.

Выборка записей

Существует несколько способов получения записей из базы данных с использованием EF. Простейший способ предусматривает проход по коллекции `DbSet<Car>`. Для его демонстрации добавим новый метод по имени `PrintAllInventory()`. Реализуем в нем цикл `foreach` для свойства `Cars` объекта `DbContext` (которое имеет тип `DbSet<Car>`) и выведем сведения о каждом объекте автомобиля:

```
private static void PrintAllInventory()
{
  // Выбрать все элементы из таблицы Inventory базы данных AutoLot
  // и вывести данные с применением специального метода ToString()
  // сущностного класса Car.
  using (var context = new AutoLotEntities())
  {
```

```
    foreach (Car c in context.Cars)
    {
      WriteLine(c);
    }
  }
}
```

Чтобы протестировать метод `PrintAllInventory()`, обновим метод `Main()` следующим образом:

```
class Program
{
  static void Main(string[] args)
  {
    WriteLine("***** Fun with ADO.NET EF *****\n");
    // int carId = AddNewRecord();
    // WriteLIne(carId);
    PrintAllInventory();
    ReadLine();
  }
```

Перечисление элементов, доступных через свойство `Cars`, приводит к неявной отправке SQL-оператора `SELECT` лежащему в основе поставщику данных ADO.NET. Важно отметить, что задачей EF является создание объекта `DataReader` для загрузки записей из базы данных и затем трансформация записей из `DataReader` в объекты типа `Car`.

Запрашивание с помощью SQL

Инфраструктура EF также поддерживает заполнение объектов `DbSet` с помощью SQL (либо операторами внутри кода, либо хранимыми процедурами). Чтобы проверить это, модифицируем метод `PrintInventory()`, как показано ниже:

```
private static void PrintAllInventory()
{
  // Выбрать все элементы из таблицы Inventory базы данных AutoLot
  // и вывести данные с применением специального метода ToString()
  // сущностного класса Car.
  using (var context = new AutoLotEntities())
  {
    // foreach (Car c in context.Cars)
    // {
    //   WriteLine(c);
    // }
    foreach (Car c in context.Cars.SqlQuery("Select CarId,Make,Color,PetName
as CarNickName from Inventory where Make=@p0", "BMW"))
    {
      WriteLine(c);
    }
  }
}
```

Хорошая новость в том, что этот код заполняет список отслеживаемыми сущностями, т.е. любые изменения или удаления будут переданы в базу данных после вызова метода `SaveChanges()`. Плохая новость (как можно заметить по тексту команды SQL) в том, что метод `SqlQuery()` не воспринимает изменения в отображении, которые были сделаны ранее. Потребуется не только применять имена таблицы базы данных и ее полей, но любые изменения в именах полей (такие как изменение на `PetName`) должны сопровождаться сопоставлением имени поля базы данных с именем свойства модели.

Запрашивание с помощью LINQ

Инфраструктура EF становится намного более мощной, когда объединяется с запросами LINQ. Рассмотрим следующее обновление метода `PrintInventory()`, в котором для получения записей из базы данных используется LINQ:

```
private static void PrintAllInventory()
{
  // Выбрать все элементы из таблицы Inventory базы данных AutoLot
  // и вывести данные с применением специального метода ToString()
  // сущностного класса Car.
  using (var context = new AutoLotEntities())
  {
    // foreach (Car c in context.Cars)
    // {
    //   WriteLine(c);
    // }
    // foreach (Car c in context.Cars.SqlQuery("Select CarId,Make,Color,PetName
as CarNickName from Inventory where Make=@p0", "BMW"))
    // {
    //   WriteLine(c);
    // }
    foreach (Car c in context.Cars.Where(c => c.Make == "BMW"))
    {
      WriteLine(c);
    }
}
```

Оператор LINQ транслируется в запрос SQL, который создает объект `DataReader` и затем возвращает коллекцию `Cars`. Сгенерированный запрос выглядит примерно так (на вашей машине он может слегка отличаться):

```
SELECT
    [Extent1].[CarId] AS [CarId],
    [Extent1].[Make] AS [Make],
    [Extent1].[Color] AS [Color],
    [Extent1].[PetName] AS [PetName]
    FROM [dbo].[Inventory] AS [Extent1]
    WHERE N'BMW' = [Extent1].[Make]
```

С учетом того, что вы уже работали со многими выражениями LINQ в главе 13, на текущий момент еще нескольких примеров будет достаточно:

```
private static void FunWithLinqQueries()
{
  using (var context = new AutoLotEntities())
  {
    // Получить проекцию новых данных.
    var colorsMakes = from item in context.Cars select new { item.Color, item.Make };
    foreach (var item in colorsMakes)
    {
      WriteLine(item);
    }

    // Получить только элементы, в которых Color == "Black".
    var blackCars = from item in context.Cars
                    where item.Color == "Black" select item;
```

```
    foreach (var item in blackCars)
    {
      WriteLine(item);
    }
  }
}
```

Наряду с тем, что синтаксис этих запросов довольно прост, не забывайте, что каждый раз, когда к объектному контексту применяется запрос LINQ, происходит обращение к базе данных! Вспомните, что при желании получить независимую копию данных, которая может быть целью новых запросов LINQ, придется использовать немедленное выполнение с помощью расширяющих методов `ToList<T>()`, `ToArray<T>()`, `ToDictionary<K,V>()` и т.д. Ниже показан модифицированный предыдущий метод, который выполняет эквивалент оператора `SELECT *`, кеширует сущности в виде массива и манипулирует данными массива с применением LINQ to Objects:

```
using (var context = new AutoLotEntities())
{
  // Получить все данные из таблицы Inventory.
  // Можно было бы также записать так:
  // var allData = (from item in context.Cars select item).ToArray();
  var allData = context.Cars.ToArray();

  // Получить проекцию новых данных.
  var colorsMakes = from item in allData select new { item.Color, item.Make };
  foreach (var item in colorsMakes)
  {
    WriteLine(item);
  }

  // Получить только элементы, в которых Color == "Black".
  var blackCars = from item in allData where item.Color== "Black" select item;
  foreach (var item in blackCars)
  {
    WriteLine(item);
  }
}
```

Чтобы протестировать это, обновим метод `Main()`:

```
static void Main(string[] args)
{
  WriteLine("***** Fun with ADO.NET EF *****\n");
  // int carId = AddNewRecord();
  // WriteLine(carId);
  // PrintAllInventory();
  FunWithLinqQueries();
  ReadLine();
}
```

Роль навигационных свойств

Как следует из названия, *навигационные свойства* позволяют выражать операции `JOIN` в программной модели Entity Framework (без необходимости в написании сложных операторов SQL). Чтобы учесть такие отношения внешних ключей, каждый класс в модели содержит виртуальные свойства, которые соединяют вместе наши классы. Например, в классе `Inventory` свойство `Orders` определено как `virtual ICollection<Order>`:

```
public virtual ICollection<Order> Orders { get; set; }
```

Тем самым инфраструктура EF информируется о том, что каждая запись базы данных `Inventory` (переименованная в класс `Car` для кода C#) может иметь ноль или множество записей `Order`.

Модель `Order` имеет ноль или одну запись `Inventory` (`Car`), ассоциированную с ней. Модель `Order` перемещается обратно в модель `Inventory` через еще одно виртуальное свойство типа `Inventory`:

```
public virtual Car Car { get; set; }
```

Ленивая, энергичная и явная загрузка

Существуют три способа, которыми EF загружает данные в модель. Ленивая и энергичная загрузка основана на настройках контекста, а явная загрузка управляется разработчиком.

Ленивая загрузка

Модификатор `virtual` позволяет инфраструктуре EF производить *ленивую загрузку* данных. Это означает, что EF загружает для каждого объекта самый минимум и затем извлекает дополнительные детали, когда свойства запрашиваются в коде. Например, при наличии следующего кода инфраструктура EF будет запускать один запрос, чтобы получить все объекты `Car`, и затем для каждого объекта `Car` выполнять другой запрос, чтобы получить все объекты `Order`:

```
using (var context = new AutoLotEntities())
{
  foreach (Car c in context.Cars)
  {
    foreach (Order o in c.Orders)
    {
      WriteLine(o.OrderId);
    }
  }
}
```

Ленивая загрузка предотвращает загрузку в память базы данных целиком (или, по крайней мере, намного большего объема информации, чем планировалось). Так как таблица `Orders` связана с таблицей `Cars`, а таблица `Customers` — с `Orders`, если бы записи загружались энергично, то извлечение всех записей `Cars` приводило бы также к извлечению всех записей `Orders` и `Customers` (исключая клиентов, не разместивших ни одного заказа).

Энергичная загрузка

Иногда *желательно* загружать все связанные записи. Например, если вы совершенно уверены в том, что нуждаетесь во всех записях `Orders` и всех записях `Cars`, то могли бы изменить предыдущий код так, как показано ниже:

```
using (var context = new AutoLotEntities())
{
  foreach (Car c in context.Cars.Include(c=>c.Orders))
  {
    foreach (Order o in c.Orders)
    {
      WriteLine(o.OrderId);
    }
  }
}
```

Тогда исходный запрос извлекал бы все записи `Cars` и все записи `Orders`. Выражение LINQ в вызове `Include()` инструктирует EF о том, что должен быть сформирован один запрос для получения всех записей, примерно так:

```
SELECT
    [Project1].[CarId] AS [CarId],
    [Project1].[Make] AS [Make],
    [Project1].[Color] AS [Color],
    [Project1].[PetName] AS [PetName],
    [Project1].[C1] AS [C1],
    [Project1].[OrderId] AS [OrderId],
    [Project1].[CustId] AS [CustId],
    [Project1].[CarId1] AS [CarId1]
    FROM ( SELECT
        [Extent1].[CarId] AS [CarId],
        [Extent1].[Make] AS [Make],
        [Extent1].[Color] AS [Color],
        [Extent1].[PetName] AS [PetName],
        [Extent2].[OrderId] AS [OrderId],
        [Extent2].[CustId] AS [CustId],
        [Extent2].[CarId] AS [CarId1],
        CASE WHEN ([Extent2].[OrderId] IS NULL) THEN CAST(NULL AS int)
ELSE 1 END AS [C1]
        FROM  [dbo].[Inventory] AS [Extent1]
        LEFT OUTER JOIN [dbo].[Orders] AS [Extent2] ON [Extent1].[CarId]
= [Extent2].[CarId]
    )  AS [Project1]
    ORDER BY [Project1].[CarId] ASC, [Project1].[C1] ASC
```

Точный синтаксис запроса не играет никакой роли; он показан в целях демонстрации того, что все записи `Cars` и `Orders` извлекаются за одно обращение к базе данных.

Явная загрузка

Явная загрузка загружает коллекцию или класс, на который осуществляется ссылка с помощью навигационного свойства. Если ленивая загрузка отключена, то вам придется загружать связанные объекты либо энергичным, либо явным образом. Отключить ленивую загрузку можно установкой свойства `LazyLoadingEnabled` конфигурации объекта `DbContext`:

```
context.Configuration.LazyLoadingEnabled = false;
```

Затем для получения связанного объекта (или объектов) необходимо использовать методы `Collection()` (для коллекций) или `Property()` (для одиночных объектов) контекста и `Load()`.

В следующем коде иллюстрируется применение метода `Collection.Load()`:

```
foreach (Car c in context.Cars)
{
  context.Entry(c).Collection(x => x.Orders).Load();
  foreach (Order o in c.Orders)
  {
    WriteLine(o.OrderId);
  }
}
```

Удаление записи

Один из способов удаления записи из базы данных предусматривает определение местоположения нужного элемента в DbSet<T> и вызов метода Remove() с передачей ему этого экземпляра. Найти необходимую запись можно посредством вызова метода Find() объекта DbSet<T> и передачи ему первичного ключа записи об автомобиле, которую требуется удалить. Добавим в класс следующий метод:

```
private static void RemoveRecord(int carId)
{
  // Найти запись об автомобиле, подлежащую удалению, по первичному ключу.
  using (var context = new AutoLotEntities())
  {
    // Проверить наличие записи.
    Car carToDelete = context.Cars.Find(carId);
    if (carToDelete != null)
    {
      context.Cars.Remove(carToDelete);
      context.SaveChanges();
    }
  }
}
```

Чтобы запустить метод RemoveRecord(), модифицируем метод Main(), как показано ниже (не забудьте убрать комментарий со строки с вызовом AddNewRecord(), если она была закомментирована при проработке предшествующих примеров):

```
static void Main(string[] args)
{
  WriteLine("***** Fun with ADO.NET EF *****\n");
  int carId = AddNewRecord();
  RemoveRecord(carId);
  // WriteLine(carId);
  // PrintAllInventory();
  // FunWithLinqQueries();
  ReadLine();
}
```

На заметку! Вызов метода Find() перед удалением записи требует дополнительного обращения к базе данных. Как вы увидите далее, установка EntityState является намного более эффективным способом удаления записей.

Удаление записи с использованием EntityState

Ранее уже упоминалось о возможности удаления записи с применением EntityState, что не требует обращения к базе данных. Добавим новый метод по имени RemoveRecordUsingEntityState():

```
private static void RemoveRecordUsingEntityState(int carId)
{
  using (var context = new AutoLotEntities())
  {
    Car carToDelete = new Car() { CarId = carId };
    context.Entry(carToDelete).State = EntityState.Deleted;
    try
    {
      context.SaveChanges();
    }
```

```
    catch (DbUpdateConcurrencyException ex)
    {
      WriteLine(ex);
    }
  }
}
```

Здесь создается новый объект `Car`, первичный ключ устанавливается в значение удаляемой записи, `EntityState` устанавливается в `EntityState.Deleted`, после чего вызывается метод `SaveChanges()`. При этом достигается выигрыш в производительности (т.к. не производится дополнительное обращение к базе данных), но утрачивается возможность проверки, существует ли объект в базе данных (если это имеет значение в конкретном сценарии). В случае если запись с указанным значением `CarId` в базе данных отсутствует, инфраструктура EF сгенерирует исключение типа `DbUpdateConcurrencyException`, определенного в пространстве имен `System.Data.Entity.Infrastructure`. Точные ошибочные сущности доступны через свойство `Entries` объекта исключения, поэтому можно легко выяснить необходимые детали.

На заметку! Тот факт, что ничего не было обновлено, не приводит к возникновению исключения `DbUpdateConcurrencyException`. Оно генерируется, когда общее количество объектов, которые *должны* быть обновлены или удалены (на основе состояния сущности), больше общего числа объектов, которые *действительно* были обновлены. Ошибочные объекты доступны посредством свойства `Entries` исключения. Для каждой записи в `Entries` можно просматривать текущие значения свойств, исходные значения свойств и текущие значения свойств в базе данных (выполнив еще один запрос).

Обновление записи

Обновление записи также реализуется просто. Понадобится определить местоположение объекта, подлежащего обновлению, установить новые значения для свойств возвращенной сущности и сохранить изменения, как показано ниже:

```
private static void UpdateRecord(int carId)
{
  // Найти запись об автомобиле, подлежащую обновлению, по первичному ключу.
  using (var context = new AutoLotEntities())
  {
    // Получить запись об автомобиле, обновить ее и сохранить!
    Car carToUpdate = context.Cars.Find(carId);
    if (carToUpdate != null)
    {
      WriteLine(context.Entry(carToUpdate).State);
      carToUpdate.Color = "Blue";
      WriteLine(context.Entry(carToUpdate).State);
      context.SaveChanges();
    }
  }
}
```

Обработка изменений в базе данных

В настоящем разделе мы построили решение EF, основанное на существующей базе данных. Это хорошо подходит в ситуации, когда в организации есть свои администраторы баз данных, и вам предоставляется база данных, над которой у вас отсутствует

контроль. Если база данных со временем изменяется, то все, что вам потребуется предпринять — это запустить мастер заново и повторно создать класс `AutoLotEntities`; классы модели будут перестроены автоматически. Разумеется, скорее всего, придется перепроектировать весь код, в котором используется модель. Но что интересного в том, если бы *абсолютно все* волшебным образом делалось за вас?

В начальном примере был пройден долгий путь к пониманию особенностей работы с Entity Framework.

Исходный код. Проект `AutoLotConsoleApp` доступен в подкаталоге `Chapter_23`.

Сборка AutoLotDAL версии 4

В предыдущем разделе код EF создавался из существующей базы данных. Инфраструктура EF может также самостоятельно создать базу данных на основе классов модели и производного от `DbContext` класса. Вдобавок к созданию начальной базы данных инфраструктура EF делает возможным создание миграций с целью обновления базы данных для соответствия изменениям в модели.

На заметку! Эта версия сборки `AutoLotDAL.dll` будет применяться в оставшихся главах книги.

Первым делом создадим новый проект библиотеки классов по имени `AutoLotDAL`. Удалим стандартный класс, который был создан, и добавим две папки, `EF` и `Models`. Теперь нужно добавить в проект инфраструктуру Entity Framework, используя NuGet. Для этого понадобится щелкнуть правой кнопкой мыши на имени проекта и выбрать в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet), как показано на рис. 23.8. (В предыдущем примере явно добавлять EF не требовалось, потому что об этом позаботился мастер. Добавление пакетов вручную позволяет точно контролировать устанавливаемые версии.)

После открытия окна NuGet Package Manager (Диспетчер пакетов NuGet) выберите пакет `EntityFramework` (рис. 23.9).

После принятия изменений и условий лицензионного соглашения инфраструктура Entity Framework (версии 6.1.3 на момент написания этой главы) установится в проект.

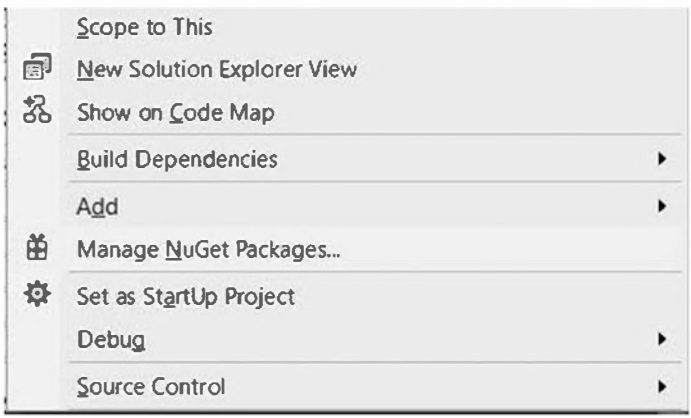


Рис. 23.8. Выбор пункта Manage NuGet Packages

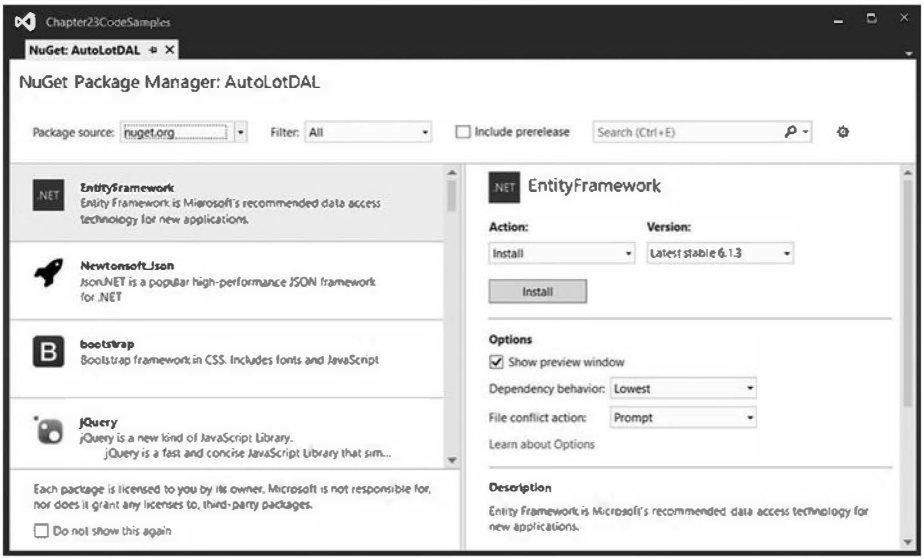


Рис. 23.9. Окно NuGet Package Manager

Аннотации данных Entity Framework

Аннотации данных кратко обсуждались ранее, и вы уже видели применение `Table`, `Column`, `Key` и `StringLength`. Существует множество других аннотаций, которые можно использовать для уточнения модели и добавления проверок достоверности, многие из которых будут применяться в оставшемся материале этой главы и книги.

На заметку! В .NET Framework доступно намного больше аннотаций данных помимо приведенных в табл. 23.5. За дополнительными сведениями обращайтесь к описаниям пространств имен `System.ComponentModel.DataAnnotations` и `System.ComponentModel.DataAnnotations.Schema` в документации .NET Framework 4.6 SDK.

Таблица 23.5. Аннотации данных, поддерживаемые инфраструктурой Entity Framework

Аннотация данных	Описание
`Key`	Определяет первичный ключ для модели. Это не обязательно, если свойству ключа назначено имя `Id` либо комбинация имени класса с `Id`, например, `OrderId`. В случае составного ключа потребуется добавить атрибут `Column` с параметром `Order`, такой как `Column[Order=1]` и `Column[Order=2]`. Поля ключей также неявно `[Required]`
`Required`	Объявляет свойство как не допускающее значения `null`
`ForeignKey`	Объявляет свойство, которое используется в качестве внешнего ключа для навигационного свойства
`StringLength`	Указывает минимальную и максимальную длины для строкового свойства
`ConcurrencyCheck`	Помечает поле для участия в проверке параллелизма, когда сервер выполняет операции обновления, вставки или удаления
`TimeStamp`	Объявляет тип как версию строки или временную метку (в зависимости от поставщика базы данных)

Окончание табл. 23.5

Аннотация данных	Описание
`Table`	Позволяет именовать классы и поля модели иначе, чем они объявлены в базе данных. Атрибут `Table` дает возможность также указывать схему (при условии, что хранилище данных поддерживает схемы)
`Column`	
`DatabaseGenerated`	Указывает, является ли поле сгенерированным базой данных. Допустимые значения: `Computed`, `Identity` и `None`
`NotMapped`	Указывает, что инфраструктура EF должна игнорировать это свойство относительно полей базы данных
`Index`	Указывает, что для столбца должен быть создан индекс. Можно задавать кластеризованный, уникальный, именованный или упорядоченный индекс

На заметку! В дополнение к аннотациям данных инфраструктура EF поддерживает интерфейс Fluent API для определения структуры таблиц и отношений между ними. Хотя ранее был представлен небольшой пример, Fluent API выходит за рамки тематики настоящей главы. Дополнительные сведения по определению таблиц и столбцов с применением Fluent API доступны по адресу `https://msdn.microsoft.com/ru-ru/data/jj591617`. Информация по определению отношений находится по адресу `https://msdn.microsoft.com/ru-ru/data/jj591620`.

Добавление или обновление классов модели

В этом разделе можно либо начать с классов модели, созданных в предыдущем примере, либо начать с нуля и создать новые классы, прорабатывая дальнейшие примеры в главе. Мы начнем с нуля, чтобы продемонстрировать весь процесс от старта до финиша.

Первым делом добавим в проект новую папку по имени `Models` и поместим в нее четыре файла классов — `CreditRisk.cs`, `Customer.cs`, `Inventory.cs` и `Order.cs`.

Создание класса модели `Inventory`

Откроем файл `Inventory.cs`, изменим класс на `public` и `partial`, а затем добавим следующие свойства и операторы `using` для пространств имен `System.ComponentModel.DataAnnotations` и `System.ComponentModel.DataAnnotations.Schema` в начало файла класса:

```
public partial class Inventory
{
  public int CarId { get; set; }
  public string Make { get; set; }
  public string Color { get; set; }
  public string PetName { get; set; }
}
```

Конфигурирование модели с помощью аннотаций данных

Прежде всего, воспользуемся атрибутом `Table` для указания имени таблицы `Inventory`. По умолчанию EF применяет соглашение, в соответствие с которым имена таблиц представляются во множественном числе английского языка, поэтому стандартное имя таблицы выглядит как `Inventories`. Добавим атрибут `Key` к свойству `CarId` и атрибут `StringLength(50)` к каждому строковому свойству. Атрибут `Key` указывает, что поле является первичным ключом для таблицы. Атрибут `StringLength(50)` устанавливает максимальную длину для строкового свойства. Можно также установить мини-

мальную длину, хотя она используется только при проверке достоверности и не влияет на создание поля базы данных, как это делает максимальная длина. Ниже приведен модифицированный код:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
 namespace AutoLotDAL.Models
{
  [Table("Inventory")]
  public partial class Inventory
  {
    [Key]
    public int CarId { get; set; }

    [StringLength(50)]
    public string Make { get; set; }

    [StringLength(50)]
    public string Color { get; set; }

    [StringLength(50)]
    public string PetName { get; set; }
  }
}
```

Добавление навигационного свойства в класс `Inventory`

Как упоминалось в разделе "Роль навигационных свойств", доступ к записям `Orders`, связанных с отдельной записью `Inventory`, осуществляется через `ICollection<Order>`:

```
public virtual ICollection<Order> Orders { get; set; } = new
HashSet<Order>();
```

Вот полный код класса `Inventory`:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace AutoLotDAL.Models
{
  [Table("Inventory")]
  public partial class Inventory
  {
    [Key]
    public int CarId { get; set; }

    [StringLength(50)]
    public string Make { get; set; }

    [StringLength(50)]
    public string Color { get; set; }

    [StringLength(50)]
    public string PetName { get; set; }

    public virtual ICollection<Order> Orders { get; set; } = new HashSet<Order>();
  }
}
```

Добавление класса `InventoryPartial`

Теперь мы собираемся добавить частичный класс для переопределения метода `ToString()` класса `Inventory`. Создадим внутри папки `Models` новую папку под названием `Partials`. Добавим в нее новый файл класса по имени `InventoryPartial.cs`. Откроем файл `InventoryPartial.cs`, переименуем класс в `Inventory` и удостоверимся, что в качестве пространства имен указано `AutoLotDAL.Models` (а не `AutoLotDAL.Models.Partials`, как установлено по умолчанию). Поместим в файл `InventoryPartial.cs` следующий код:

```
public partial class Inventory
{
  public override string ToString()
  {
    // Поскольку столбец PetName может быть пустым,
    // предоставить стандартное имя **No Name**.
    return $"{this.PetName ?? "**No Name**"} is a {this.Color} {this.Make}
with ID {this.CarId}.";
  }
}
```

Далее добавим вычисляемое поле, которое объединяет значения `Make` и `Color` объекта автомобиля. Это поле не будет сохраняться в базе данных, равно как не будет заполняться при создании объекта на основе данных из базы, так что его необходимо снабдить атрибутом `[NotMapped]`:

```
[NotMapped]
public string MakeColor => $"{Make} + ({Color})";
```

Создание класса модели `Customer`

Откроем файл класса `Customer.cs` и добавим в начало операторы `using` для пространств имен `System.ComponentModel.DataAnnotations` и `System.ComponentModel.DataAnnotations.Schema`. Дальнейший процесс совпадает с тем, который выполнялся при создании класса `Inventory`, поэтому здесь будет показан только код. Единственный элемент, требующий внимания — свойство `FullName`, которое вычисляется и, следовательно, декорировано атрибутом `NotMapped`. Вот полный код класса:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace AutoLotDAL.Models
{
  public partial class Customer
  {
    [Key]
    public int CustId { get; set; }

    [StringLength(50)]
    public string FirstName { get; set; }

    [StringLength(50)]
    public string LastName { get; set; }

    [NotMapped]
    public string FullName => FirstName + " " + LastName;

    public virtual ICollection<Order> Orders { get; set; } = new HashSet<Order>();
  }
}
```

Создание класса модели Order

Откроем файл класса Order.cs и добавим в начало операторы using для пространств имен System.ComponentModel.DataAnnotations и System.ComponentModel.DataAnnotations.Schema. Добавим поле первичного ключа OrderId и затем навигационные свойства Customer и Car. В дополнение к навигационным свойствам добавим поля внешних ключей CustId и CarId. Ниже представлен код класса:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace AutoLotDAL.Models
{
  public partial class Order
  {
    public int OrderId { get; set; }
    public int CustId { get; set; }
    public int CarId { get; set; }
    public virtual Customer Customer { get; set; }
    public virtual Inventory Car { get; set; }
  }
}
```

Теперь мы будем применять к классу Order атрибуты аннотаций данных. Первичный ключ, OrderId, также является первичным ключом таблицы (и по этой причине обязательным) и установлен как столбец Identity. К свойству OrderId необходимо добавить три атрибута:

- [Key] — обозначает первичный ключ;
- [Required] — указывает, что поле не допускает значения null;
- [DatabaseGenerated(DatabaseGeneratedOption.Identity)] — означает, что поле является столбцом Identity.

Вспомните из предшествующих глав, что атрибуты можно перечислять по отдельности или представлять в виде списка, разделяя запятыми. Поместим все три атрибута вместе, чтобы свойство OrderId выглядело так:

```
[Key, Required, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public int OrderId { get; set;}
```

Значения, поддерживающие оба навигационных свойства, по умолчанию обязательны, поскольку типы не допускают null. Однако для улучшения читабельности кода мы явно пометим их как обязательные:

```
[Required]
public int CustId { get; set; }

[Required]
public int CarId { get; set; }
```

Наконец, с помощью аннотаций укажем свойства, служащие поддерживающими полями для двух навигационных свойств:

```
[ForeignKey("CustId")]
public virtual Customer Customer { get; set; }

[ForeignKey("CarId")]
public virtual Inventory Car { get; set; }
```

Вот полный код класса:

```
public partial class Order
{
  [Key, Required, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
  public int OrderId { get; set; }

  [Required]
  public int CustId { get; set; }

  [Required]
  public int CarId { get; set; }

  [ForeignKey("CustId")]
  public virtual Customer Customer { get; set; }

  [ForeignKey("CarId")]
  public virtual Inventory Car { get; set; }
}
```

Создание класса CreditRisk

Откроем файл класса `CreditRisk.cs` и добавим в начало операторы `using` для пространств имен `System.ComponentModel.DataAnnotations` и `System.ComponentModel.DataAnnotations.Schema`. Единственное изменение, которое понадобится внести в класс `CreditRisk` — переместить его в новое пространство имен. Ниже показан первоначальный код класса:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace AutoLotDAL.Models
{
  public partial class CreditRisk
  {
    public int CustId { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }
  }
}
```

Добавим атрибут `Key` к свойству `CustId` и атрибут `StringLength` к свойствам `FirstName` и `LastName`. Полный код класса `Order` выглядит следующим образом:

```
namespace AutoLotDAL.Models
{
  public partial class CreditRisk
  {
    [Key]
    public int CustId { get; set; }

    [StringLength(50)]
    public string FirstName { get; set; }

    [StringLength(50)]
    public string LastName { get; set; }

  }
}
```

Добавление класса, производного от DbContext

В нашей мозаике пока еще отсутствует один важный фрагмент: контекстный класс, производный от DbContext. К счастью, добавить его легко. При выделенной папке EF в проекте AutoLotDAL выберем пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент). В левой панели окна выберем узел Data (Данные), в центральной панели выберем элемент ADO.NET Entity Data Model (Модель сущностных данных ADO.NET) и введем AutoLotEntities в поле Name (Имя), как показано на рис. 23.10.

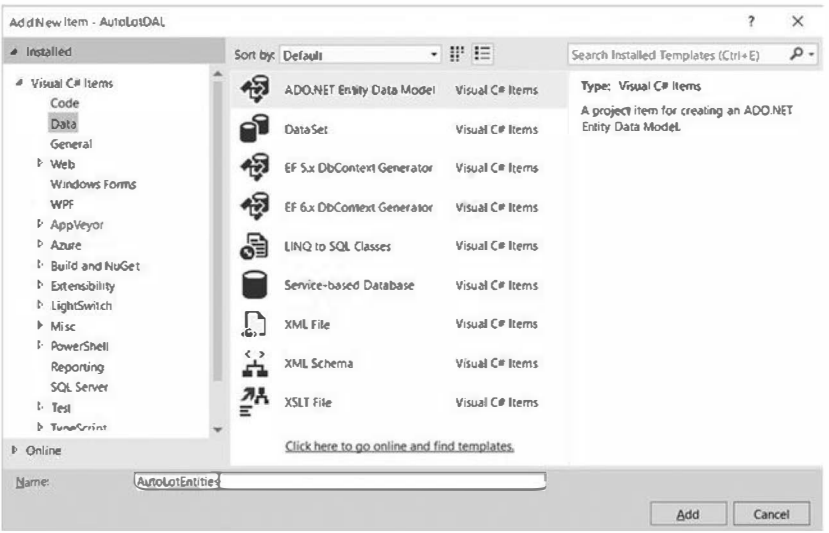


Рис. 23.10. Добавление контекста в проект

В открывшемся мастере создания модели сущностных данных (Entity Data Model Wizard) выберем вариант Empty Code First model (Пустая модель Code First), как показано на рис. 23.11.

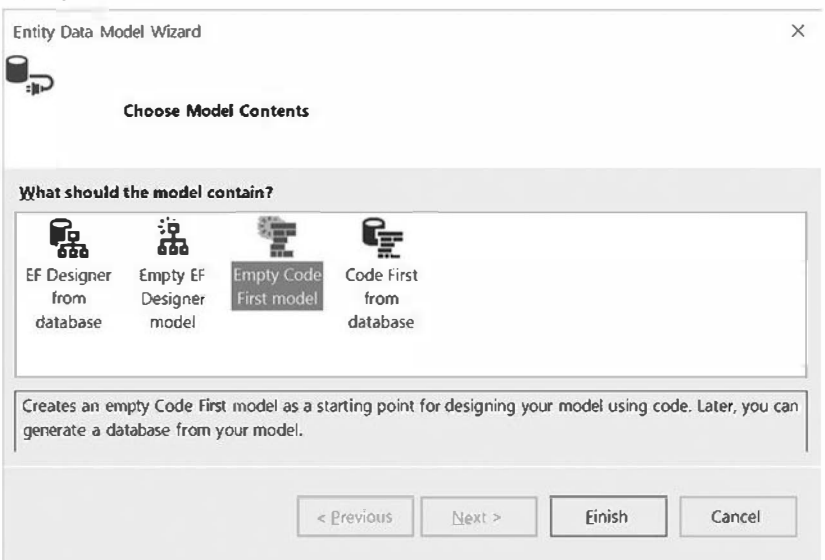


Рис. 23.11. Выбор варианта Empty Code First model

Одно отличие здесь вполне очевидно: мастер вообще не запрашивает строку подключения! Для варианта Empty Code First model предполагается отсутствие базы данных, поэтому мастер строит новую строку подключения, которая исследуется в следующем разделе.

Обновление файла *.config и строки подключения EF

Откроем файл App.config и посмотрим, какие изменения были сделаны EF. Большинство из них должны выглядеть знакомо. Двумя бросающимися в глаза отличиями являются свойства data source и initial catalog в строке подключения:

```
  <connectionStrings>
    <add name="AutoLotEntities" connectionString="data source=(LocalDb)\
MSSQLLocalDB;initial catalog=AutoLotDAL.EF.AutoLotEntities;
integrated security=True;MultipleActiveResultSets=True;
App=EntityFramework" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

LocalDb — это разновидность системы SQL Server Express, ориентированная на разработчиков приложений. В ней используется минимальный набор файлов и не требуется конфигурирование со стороны разработчика. Вместо имени сервера вроде (local)\SQLEXPRESS2014 можно применять (LocalDb) с уникальным именем.

Каталог (имя базы данных) выводится из пространства имен в сочетании с именем, указанным в мастере. В этом случае каталог выглядит как AutoLotDAL.EF.AutoLotEntities.

Позже в главе база данных будет перемещена в систему SQL Server Express (в интересах последующих глав), а пока просто изменим имя каталога на AutoLot и строку подключения на AutoLotConnection:

```
<add name="AutoLotConnection" connectionString="data source=(LocalDb)\
MSSQLLocalDb;initial catalog=AutoLot;integrated security=True;
MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />
```

Обновление контекста

Конструктор нашего производного от DbContext класса передает имя строки подключения конструктору базового класса DbContext. Откроем файл AutoLotEntities.cs и изменим имя строки подключения в конструкторе на AutoLotConnection. Ниже показан модифицированный код:

```
public class AutoLotEntities : DbContext
{
  public AutoLotEntities()
    : base("name=AutoLotConnection")
  {
  }
}
```

Добавим оператор using для модели (AutoLotDAL.Models) и по одному свойству типа DbSet для каждого класса модели. Вот как выглядит код:

```
public virtual DbSet<CreditRisk> CreditRisks { get; set; }
public virtual DbSet<Customer> Customers { get; set; }
public virtual DbSet<Inventory> Inventory { get; set; }
public virtual DbSet<Order> Orders { get; set; }
```

Добавление хранилищ

Распространенным шаблоном проектирования для доступа к данным является Repository (Хранилище). Согласно описанию Мартина Фаулера суть этого шаблона заключается в том, чтобы служить связующим звеном между уровнями предметной области и отображения данных. Несмотря на то что полное объяснение шаблона Repository выходит за рамки этой книги, отметим, что он полезен в устранении дублированного кода.

На заметку! Дополнительные сведения о шаблоне проектирования Repository можно найти на веб-сайте Мартина Фаулера по адресу `www.martinfowler.com/eaaCatalog/repository.html`.

Добавление интерфейса IRepo

Одно из преимуществ EF связано с тем, что все модели и коллекции строго типизированы. Такая характеристика сохраняется и для классов хранилищ. Начнем с добавления в проект `AutoLotDAL` новой папки под названием `Repos`. Затем добавим в папку `Repos` новый интерфейс по имени `IRepo`. Этот интерфейс будет открывать моделям доступ к важнейшим методам CRUD. Будут доступны как синхронные версии методов, так и асинхронные (за дополнительной информацией об асинхронных версиях обращайтесь в главу 19). Ниже приведено полное определение интерфейса `IRepo`:

```
interface IRepo<T>
{
  int Add(T entity);
  Task<int> AddAsync(T entity);
  int AddRange(IList<T> entities);
  Task<int> AddRangeAsync(IList<T> entities);
  int Save(T entity);
  Task<int> SaveAsync(T entity);
  int Delete(int id);
  Task<int> DeleteAsync(int id);
  int Delete(T entity);
  Task<int> DeleteAsync(T entity);
  T GetOne(int? id);
  Task<T> GetOneAsync(int? id);
  List<T> GetAll();
  Task<List<T>> GetAllAsync();
  List<T> ExecuteQuery(string sql);
  Task<List<T>> ExecuteQueryAsync(string sql);
  List<T> ExecuteQuery(string sql,object[] sqlParametersObjects );
  Task<List<T>> ExecuteQueryAsync(string sql, object[] sqlParametersObjects);
}
```

Последние четыре члена позволяют передавать строковый запрос SQL (и имеют перегруженные версии, принимающие параметры для запроса SQL). Выполнение этих методов приведет к загрузке (и отслеживанию) сущностей в объект `DbSet<T>` контекста. Упомянутые методы используются нечасто, поскольку мощные запросы можно строить с помощью языка LINQ, который скрывает детали SQL от разработчика, но они представлены здесь для того, чтобы продемонстрировать, каким образом обращаться к SQL напрямую внутри контекста.

На заметку! О достоинствах и потенциальных проблемах выполнения кода доступа к данным асинхронным образом в высоконагруженных системах (таких как веб-приложение или веб-служба) велось множество дискуссий. В этой книге будут описаны синхронный и асинхронный механизмы в EF, но вы должны самостоятельно тестировать их в своем конкретном приложении.

Добавление класса BaseRepo

Добавим в папку Repos еще один класс по имени BaseRepo. Он будет реализовывать общую функциональность для всех классов хранилищ, которые будут его подклассами. Класс BaseRepo будет обобщенным, так что производные классы хранилищ могут строго типизировать методы. Первым делом добавим защищенное свойство для контекста AutoLotEntities и создадим его экземпляр. Начальное определение класса представлено ниже:

```
using AutoLotDAL.EF;
public abstract class BaseRepo<T> where T:class,new()
{
  public AutoLotEntities Context { get; } = new AutoLotEntities();
}
```

Все действия начинаются со свойства DbSet<T> контекста, поэтому добавим защищенное свойство DbSet<T> по имени Table:

```
using AutoLotDAL.EF;
public abstract class BaseRepo<T>:where T:class,new()
{
  public AutoLotEntities Context { get; } = new AutoLotEntities();
  protected DbSet<T> Table;
}
```

Реализация вспомогательных методов SaveChanges()

Далее добавим два метода, предназначенные для сохранения изменений, один синхронный, а другой асинхронный. Данные методы являются просто оболочками для методов SaveChanges() и SaveChangesAsync() класса DbContext и помещены в базовый класс, чтобы производные классы хранилищ могли разделять эту реализацию. С вызовом упомянутых методов связан значительный объем кода обработки ошибок, и лучше записывать такой код только один раз. Обработчики исключений для метода SaveChanges() класса DbContext представляют собой заглушки. В производственном приложении понадобится соответствующим образом обрабатывать любые исключения.

```
internal int SaveChanges()
{
  try
  {
    return Context.SaveChanges();
  }
  catch (DbUpdateConcurrencyException ex)
  {
    // Генерируется, когда возникла ошибка параллелизма;
    // пока что просто сгенерировать исключение повторно.
    throw;
  }
  catch (DbUpdateException ex)
  {
    // Генерируется, когда обновление базы данных терпит отказ.
    // Проверить внутреннее исключение (исключения), чтобы получить
    // дополнительные сведения и затронутые объекты;
    // пока что просто сгенерировать исключение повторно.
    throw;
  }
```

```
    catch (CommitFailedException ex)
    {
      // Обработать здесь ошибки, связанные с транзакцией;
      // пока что просто сгенерировать исключение повторно.
      throw;
    }
    catch (Exception ex)
    {
      // Были сгенерированы какие-то другие исключения,
      // которые должны быть обработаны.
      throw;
    }
  }
  internal async Task<int> SaveChangesAsync()
  {
    try
    {
      return await Context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException ex)
    {
      // Генерируется, когда возникла ошибка параллелизма;
      // пока что просто сгенерировать исключение повторно.
      throw;
    }
    catch (DbUpdateException ex)
    {
      // Генерируется, когда обновление базы данных терпит отказ.
      // Проверить внутреннее исключение (исключения), чтобы получить
      // дополнительные сведения и затронутые объекты;
      // пока что просто сгенерировать исключение повторно.
      throw;
    }
    catch (CommitFailedException ex)
    {
      // Обработать здесь ошибки, связанные с транзакцией;
      // пока что просто сгенерировать исключение повторно.
      throw;
    }
    catch (Exception ex)
    {
      // Были сгенерированы какие-то другие исключения,
      // которые должны быть обработаны.
      throw;
    }
  }
```

На заметку! Создание нового экземпляра `DbContext` может оказаться затратным процессом с точки зрения производительности. Когда создается новый экземпляр контекстного класса, лежащий в основе `DbContext` взаимодействует с базой данных несколько раз. Объем таких коммуникаций варьируется в зависимости от множества факторов, включая среди прочего сложность модели и количество миграций. Если данный класс применяется в клиентском приложении наподобие WPF или Windows Forms, то в действительности экземпляров этого класса будет создано немного. Если он используется в веб-приложении (вроде ASP.NET Web Forms или ASP.NET MVC), то может быть разумно сделать класс `BaseRepo` одиночкой. Не существует единственного решения для абсолютно всех ситуаций, потому что они отличаются, и требуется настройка под каждое конкретное приложение.

Извлечение записей

Методы `GetOne()/GetOneAsync()` являются оболочками для методов `Find()/FindAsync()` класса `DbSet<T>`. Подобным образом методы `GetAll()/GetAllAsync()` представляют собой оболочки для методов `ToList()/ToListAsync()`.

Ниже показан код:

```
public T GetOne(int? id) => Table.Find(id);
public Task<T> GetOneAsync(int? id) => Table.FindAsync(id);
public List<T> GetAll() => Table.ToList();
public Task<List<T>> GetAllAsync() => Table.ToListAsync();
```

Извлечение записей с помощью SQL

Последние четыре метода интерфейса, которые должны быть реализованы, работают со строковым запросом SQL. Они передают строку и параметры `DbSet<T>`:

```
public List<T> ExecuteQuery(string sql)  => Table.SqlQuery(sql).ToList();
public Task<List<T>> ExecuteQueryAsync(string sql)
  => Table.SqlQuery(sql).ToListAsync();
public List<T> ExecuteQuery(string sql, object[] sqlParametersObjects)
  => Table.SqlQuery(sql, sqlParametersObjects).ToList();
public Task<List<T>> ExecuteQueryAsync(string sql, object[] sqlParametersObjects)
  => Table.SqlQuery(sql).ToListAsync();
```

На заметку! Вы должны соблюдать чрезвычайную осторожность при запуске низкоуровневого запроса SQL в отношении хранилища данных, особенно если запрос принимает пользовательский ввод. Поступая так, вы делаете свое приложение уязвимым к атакам внедрением SQL. В настоящей книге вопросы безопасности не раскрываются, но мы хотим подчеркнуть опасности выполнения низкоуровневых операторов SQL.

Добавление записей

Многие методы могут поддерживаться в классе `BaseRepo` посредством обобщений. Начнем с методов `Add()` и `AddRange()` (вспомните, что в рассматриваемом примере мы реализуем и синхронную, и асинхронную версию; вам может требоваться только какая-то одна из них). Каждый метод `Add()/AddRange()` добавляет объект `T/IList<T>` в `DbSet<T>` (доступный через свойство `Table`). После этого необходимо вызвать метод `SaveChanges()/SaveChangesAsync()`. Код выглядит следующим образом:

```
public int Add(T entity)
{
  Table.Add(entity);
  return SaveChanges();
}
public Task<int> AddAsync(T entity)
{
  Table.Add(entity);
  return SaveChangesAsync();
}
public int AddRange(IList<T> entities)
{
  Table.AddRange(entities);
  return SaveChanges();
}
public Task<int> AddRangeAsync(IList<T> entities)
{
  Table.AddRange(entities);
  return SaveChangesAsync();
}
```

Основная часть класса BaseRepo завершается реализацией интерфейса IDisposable, который помогает обеспечить своевременное освобождение любых ресурсов. Добавим интерфейс IDisposable к классу и затем добавим показанный ниже код (обратите внимание на вызов Context.Dispose() в методе Dispose()):

```
public abstract class BaseRepo: IDisposable
{
  protected AutoLotEntities Context { get; } = new AutoLotEntities();

  // Для краткости код методов SaveChanges() и SaveChangesAsync() не показан.
  bool disposed = false;
  public void Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(this);
  }
  protected virtual void Dispose(bool disposing)
  {
    if (disposed)
      return;
    if (disposing)
    {
      Context.Dispose();
      // Освободить здесь любые управляемые объекты.
      //
    }

    // Освободить здесь любые управляемые объекты.
    //
    disposed = true;
  }
}
```

На заметку! Дополнительные сведения о реализации интерфейса IDisposable можно найти по адресу https://msdn.microsoft.com/en-us/library/system.idisposable(v=vs.110).aspx.

Обновление записей

В методах Save()/SaveAsync() сначала необходимо установить состояние сущности в EntityState.Modified и затем вызвать SaveChanges()/SaveChangesAsync(). Установка состояния гарантирует, что контекст распространит изменения на сервер. Ниже показан код:

```
public int Save(T entity)
{
  Context.Entry(entity).State = EntityState.Modified;
  return SaveChanges();
}

public Task<int> SaveAsync(T entity)
{
  Context.Entry(entity).State = EntityState.Modified;
  return SaveChangesAsync();
}
```

Удаление записей

Код для методов `Delete()/DeleteAsync()` аналогичен. Когда вызывающий код передает объект, обобщенные методы `BaseRepo` устанавливают состояние в `EntityState.Deleted` и затем вызывают `SaveChanges()/SaveChangesAsync()`:

```
public int Delete(T entity)
{
  Context.Entry(entity).State = EntityState.Deleted;
  return SaveChanges();
}
public Task<int> DeleteAsync(T entity)
{
  Context.Entry(entity).State = EntityState.Deleted;
  return SaveChangesAsync();
}
```

Добавление хранилища для таблицы `Inventory`

Добавим в папку `Repos` новый файл класса по имени `InventoryRepo.cs`. Унаследуем его от класса `BaseRepo<Inventory>`, реализуем интерфейс `IRepo<Inventory>` и установим свойство `Table` в `DbSet<Inventory>`. Начальный код должен выглядеть так:

```
public class InventoryRepo : BaseRepo<Inventory>, IRepo<Inventory>
{
  public InventoryRepo()
  {
    Table = Context.Inventory;
  }
}
```

Далее понадобится реализовать все члены интерфейса.

Удаление записей по идентификаторам

Чтобы удалить запись таблицы `Inventory` по первичному ключу, необходимо создать экземпляр класса `Inventory`, присвоить `CarId` значение параметра `id`, установить состояние в `EntityState.Deleted` и вызвать метод `SaveChanges()/SaveChangesAsync()`. Ниже показан код:

```
public int Delete(int id)
{
  Context.Entry(new Inventory() {CarId=id}).State = EntityState.Deleted;
  return SaveChanges();
}

public Task<int> DeleteAsync(int id)
{
  Context.Entry(new Inventory() { CarId = id }).State = EntityState.Deleted;
  return SaveChangesAsync();
}
```

Добавление остальных хранилищ

Классы `CustomerRepo`, `OrderRepo` и `CreditRiskRepo` следуют тому же самому шаблону, что и класс `InventoryRepo`. Скопируем файл `InventoryRepo.cs` в `CreditRiskRepo.cs`, `CustomerRepo.cs` и `OrderRepo.cs`, а затем соответствующим образом обновим методы `Delete()`, обобщенные типы и конструкторы. Далее приведен код упомянутых классов:

```
public class OrderRepo:BaseRepo<Order>,IRepo<Order>
{
  public OrderRepo()
  {
    Table = Context.Orders;
  }
  public int Delete(int id)
  {
    Context.Entry(new Order()
    {
       OrderId = id
    }).State = EntityState.Deleted;
    return SaveChanges();
  }
  public Task<int> DeleteAsync(int id)
  {
    Context.Entry(new Order()
    {
      OrderId = id
    }).State = EntityState.Deleted;
    return SaveChangesAsync();
  }
}
public class CustomerRepo:BaseRepo<Customer>,IRepo<Customer>
{
  public CustomerRepo()
  {
    Table = Context.Customers;
  }
  public int Delete(int id)
  {
    Context.Entry(new Customer()
    {
      CustId = id
    }).State = EntityState.Deleted;
    return SaveChanges();
  }
  public Task<int> DeleteAsync(int id)
  {
    Context.Entry(new Customer()
    {
      CustId = id
    }).State = EntityState.Deleted;
    return SaveChangesAsync();
  }
}
public class CreditRiskRepo:BaseRepo<CreditRisk>,IRepo<CreditRisk>
{
  public CreditRiskRepo()
  {
    Table = Context.CreditRisks;
  }
  public int Delete(int id)
  {
    Context.Entry(new CreditRisk()
    {
      CustId = id
    }).State = EntityState.Deleted;
    return SaveChanges();
  }
```

```
  public Task<int> DeleteAsync(int id)
  {
    Context.Entry(new CreditRisk()
    {
      CustId = id
    }).State = EntityState.Deleted;
    return SaveChangesAsync();
  }
}
```

Инициализация базы данных

Мощным средством инфраструктуры EF является возможность инициализации базы данных начальными данными. Это особенно удобно во время разработки, т.к. позволяет восстанавливать базу данных в известное состояние перед каждым запуском кода. Процесс инициализации предусматривает создание класса, унаследованного от `DropCreateDatabaseIfModelChanges<TContext>` или `DropCreateDatabaseAlways<TContext>`.

Создадим в папке `EF` новый класс по имени `DataInitializer`. Унаследуем его от `DropCreateDatabaseAlways<AutoLotEntities>` и переопределим метод `Seed()`:

```
using System.Collections.Generic;
using System.Data.Entity;
using AutoLotDAL.Models;
public class DataInitializer : DropCreateDatabaseAlways<AutoLotEntities>
{
  protected override void Seed(AutoLotEntities context)
  {
  }
}
```

Класс `DropCreateDatabaseAlways` строго типизирован контекстным классом `AutoLotEntities` и будет удалять, а затем воссоздавать базу данных каждый раз, когда программа запускается. Подобным образом класс `DropCreateDatabaseIfModelChanges <TContext>` будет удалять и воссоздавать базу данных, только когда в модели имеются изменения. Метод `Seed()` получает экземпляр производного контекста, который можно применять для заполнения таблиц. Процесс прост: нужно вызвать метод `Add()` на подходящем объекте `DbSet`, и когда добавление записей завершено, вызвать `SaveChanges()`. Ниже показан пример начального заполнения базы данных теми же самыми записями, которые использовались в предыдущей главе:

```
protected override void Seed(AutoLotEntities context)
{
  var customers = new List<Customer>
  {
    new Customer {FirstName = "Dave", LastName = "Brenner"},
    new Customer {FirstName = "Matt", LastName = "Walton"},
    new Customer {FirstName = "Steve", LastName = "Hagen"},
    new Customer {FirstName = "Pat", LastName = "Walton"},
    new Customer {FirstName = "Bad", LastName = "Customer"},
  };
  customers.ForEach(x => context.Customers.Add(x));
  var cars = new List<Inventory>
  {
    new Inventory {Make = "VW", Color = "Black", PetName = "Zippy"},
    new Inventory {Make = "Ford", Color = "Rust", PetName = "Rusty"},
    new Inventory {Make = "Saab", Color = "Black", PetName = "Mel"},
```

```
    new Inventory {Make = "Yugo", Color = "Yellow", PetName = "Clunker"},
    new Inventory {Make = "BMW", Color = "Black", PetName = "Bimmer"},
    new Inventory {Make = "BMW", Color = "Green", PetName = "Hank"},
    new Inventory {Make = "BMW", Color = "Pink", PetName = "Pinky"},
    new Inventory {Make = "Pinto", Color = "Black", PetName = "Pete"},
    new Inventory {Make = "Yugo", Color = "Brown", PetName = "Brownie"},
  };
  cars.ForEach(x => context.Inventory.Add(x));
  var orders = new List<Order>
  {
    new Order {Car = cars[0], Customer = customers[0]},
    new Order {Car = cars[1], Customer = customers[1]},
    new Order {Car = cars[2], Customer = customers[2]},
    new Order {Car = cars[3], Customer = customers[3]},
  };
  orders.ForEach(x => context.Orders.Add(x));
  context.CreditRisks.Add(
    new CreditRisk
    {
      CustId = customers[4].CustId,
      FirstName = customers[4].FirstName,
      LastName = customers[4].LastName,
  });
  context.SaveChanges();
}
```

Последний шаг заключается в установке инициализатора с помощью такого кода (который будет добавлен в следующем разделе):

```
Database.SetInitializer(new DataInitializer());
```

Тестирование сборки AutoLotDAL

Код для тестирования похож на код, который применялся для предыдущей версии сборки `AutoLotDAL.dll`**, но вместо взаимодействия напрямую с контекстом здесь будут использоваться классы хранилищ. Начнем с добавления к решению нового проекта консольного приложения по имени** `AutoLotTestDrive` **и установки этого проекта в качестве стартового. Посредством NuGet добавим в проект инфраструктуру EF и обновим элемент** `connectionStrings` **в файле** `App.config` **следующим образом:**

```
  <connectionStrings>
    <add name="AutoLotConnection" connectionString="data source=(LocalDb)\
MSSQLLocalDb;initial catalog=AutoLot;integrated security=True;
MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />
  </connectionStrings>
```

Добавим ссылку на проект `AutoLotDAL`**. Откроем файл** `Program.cs` **и поместим в метод** `Main()` **такой код:**

```
static void Main(string[] args)
{
  Database.SetInitializer(new DataInitializer());
  WriteLine("***** Fun with ADO.NET EF Code First *****\n");
  ReadLine();
}
```

Вывод всех записей из Inventory

Для вывода всех записей понадобится вызвать метод GetAll() класса InventoryRepo и выполнить проход по возвращенному списку. Между таким кодом и работой непосредственно с контекстом нет больших отличий, но шаблон Repository предлагает согласованный способ доступа и оперирования с данными для всех классов.

```
private static void PrintAllInventory()
{
  using (var repo = new InventoryRepo())
  {
    foreach (Inventory c in repo.GetAll())
    {
      WriteLine(c);
    }
  }
}
```

Добавление записей в Inventory

Добавление новых записей демонстрирует простоту обращения к EF с применением хранилищ. Разумеется, в производственной системе потребуется организовать обработку ошибок, но добавление записей сводится просто к вызову метода Add() или AddRange() хранилища. Код показан ниже:

```
private static void AddNewRecord(Inventory car)
{
  // Добавить запись в таблицу Inventory базы данных AutoLot.
  using (var repo = new InventoryRepo())
  {
    repo.Add(car);
  }
}
private static void AddNewRecords(IList<Inventory> cars)
{
  // Добавить записи в таблицу Inventory базы данных AutoLot.
  using (var repo = new InventoryRepo())
  {
    repo.AddRange(cars);
  }
}
```

Чтобы протестировать эти методы, добавим в Main() следующий код:

```
static void Main(string[] args)
{
  Database.SetInitializer(new MyDataInitializer());
  WriteLine("***** Fun with ADO.NET EF Code First *****\n");
  var car1 = new Inventory()
    { Make = "Yugo", Color = "Brown", PetName = "Brownie" };
  var car2 = new Inventory()
    { Make = "SmartCar", Color = "Brown", PetName = "Shorty" };
  AddNewRecord(car1);
  AddNewRecord(car2);
  AddNewRecords(new List<Inventory> { car1, car2 });
  PrintAllInventory();
  ReadLine();
}
```

Редактирование записей

Сохранение изменений в записях реализуется просто. Понадобится получить объект `Inventory`, внести некоторые изменения и вызвать метод `Save()` класса `InventoryRepo`. Вот как выглядит код (содержащий дополнительный код для вывода значения свойства `EntityState` объекта):

```
private static void UpdateRecord(int carId)
{
  using (var repo = new InventoryRepo())
  {
    // Получить запись об автомобиле, изменить ее и сохранить!
    var carToUpdate = repo.GetOne(carId);
    if (carToUpdate != null)
    {
      WriteLine("Before change: " + repo.Context.Entry(carToUpdate).State);
      carToUpdate.Color = "Blue";
      WriteLine("After change: " + repo.Context.Entry(carToUpdate).State);
      repo.Save(carToUpdate);
      WriteLine("After save: " + repo.Context.Entry(carToUpdate).State);
    }
  }
}
```

В целях тестирования добавим в метод `Main()` следующий код:

```
static void Main(string[] args)
{
  Database.SetInitializer(new MyDataInitializer());
  WriteLine("***** Fun with ADO.NET EF Code First *****\n");
  var car1 = new Inventory()
    { Make = "Yugo", Color = "Brown", PetName = "Brownie" };
  var car2 = new Inventory()
    { Make = "SmartCar", Color = "Brown", PetName = "Shorty" };
  AddNewRecord(car1);
  AddNewRecord(car2);
  AddNewRecords(new List<Inventory> { car1, car2 });
  UpdateRecord(car1.CarId);
  PrintAllInventory();
  ReadLine();
}
```

Использование навигационных свойств

Добавим метод по имени `ShowAllOrders()`. В этом методе применяется оператор `using` для класса `OrdersRepo`. Для каждой записи, возвращенной методом `GetAll()`, мы выводим значения свойств `itm.Customer.FullName` и `itm.Car.PetName`. Ниже приведен код:

```
private static void ShowAllOrders()
{
  using (var repo = new OrderRepo())
  {
    WriteLine("*********** Pending Orders ***********");
    foreach (var itm in repo.GetAll())
    {
      WriteLine($"->{itm.Customer.FullName} is waiting on {itm.Car.PetName}");
    }
  }
}
```

Теперь добавим вызов метода `ShowAllOrders()` в `Main()`. В результате запуска программы будет получен следующий вывод (он может варьироваться в зависимости от текущих данных в базе):

```
***** Fun with ADO.NET EF Code First *****

********** Pending Orders **********
-> Dave Brenner is waiting on Bimmer
-> Matt Walton  is waiting on Zippy
-> Steve Hagen is waiting on Clunker
-> Pat Walton is waiting on Pinky
```

Если во время выполнения приложения заглянуть в окно Output (Вывод), можно заметить множество индивидуальных обращений к базе данных: одно для получения всех заказов и дополнительные обращения для получения каждого отдельного имени клиента и дружественного имени автомобиля. Как обсуждалось ранее в главе, это связано с ленивой загрузкой. Далее мы будем использовать энергичную загрузку для хранилища `InventoryRepo`.

Нужно также добавить в `Main()` вызов метода `ShowAllOrders()`:

```
ShowAllOrders();
```

Энергичная загрузка

Первым делом добавим оператор `using` для пространства имен `System.Data.Entity`. Затем создадим метод по имени `ShowAllOrdersEagerlyFetched()`. Из-за отсутствия в классе хранилища метода, который бы обеспечивал энергичную загрузку, придется работать напрямую с объектом `AutoLotEntities`. (При необходимости этот метод можно добавить в класс `OrderRepo`.) Ниже показан код метода `ShowAllOrdersEagerlyFetched()`:

```
private static void ShowAllOrdersEagerlyFetched()
{
  using (var context = new AutoLotEntities())
  {
    WriteLine("********** Pending Orders **********");
    var orders = context.Orders
      .Include(x => x.Customer)
      .Include(y => y.Car)
      .ToList();
    foreach (var itm in orders)
    {
      WriteLine($"->{itm.Customer.FullName} is waiting on {itm.Car.PetName}");
    }
  }
}
```

Теперь этот метод можно вызвать в `Main()`:

```
ShowAllOrdersEagerlyFetched();
```

Многотабличные действия и неявные транзакции

Как вы помните, инфраструктура EF автоматически помещает все изменения, распространяемые с помощью вызова `SaveChanges()`, внутрь неявной транзакции. Воспроизвести пример транзакции из главы 22, в котором перемещалась запись из таблицы `Customer` в таблицу `CreditRisk`, чрезвычайно просто. Созданные до сих пор хранилища работают только с одной таблицей за раз, поэтому необходимо взаимодействовать непосредственно с контекстом.

Начнем с создания метода по имени `MakeCustomerARisk()`, который будет переносить запись о клиенте из таблицы `Customers` в таблицу `CreditRisk`. Поскольку метода вроде `Move()` не существует, операцию придется кодировать как состоящую из двух этапов. Добавим запись в `CreditRisk`, после чего удалим ее из `Customers`. Когда сущность создается в одном контексте, она должна быть отсоединена и затем подключена к новому контексту. Это является причиной установки состояния сущности в `EntityState.Detached` в `Main()` и последующего вызова метода `Attach()` на новом контексте.

Ниже показан код:

```
private static CreditRisk MakeCustomerARisk(Customer customer)
{
  using (var context = new AutoLotEntities())
  {
    context.Customers.Attach(customer);
    context.Customers.Remove(customer);
    var creditRisk = new CreditRisk()
    {
      FirstName = customer.FirstName,
      LastName = customer.LastName
    };
    context.CreditRisks.Add(creditRisk);
    try
    {
      context.SaveChanges();
    }
    catch (DbUpdateException ex)
    {
      WriteLine(ex);
    }
    catch (Exception ex)
    {
      WriteLine(ex);
    }
    return creditRisk;
  }
}
```

Чтобы взглянуть на результат, создадим метод по имени `PrintAllCustomersAndCreditRisks()`. Для прохода по записям `Customer` и `CreditRisk` применяются существующие классы хранилищ.

```
private static void PrintAllCustomersAndCreditRisks()
{
  WriteLine("*********** Customers ***********");
  using (var repo = new CustomerRepo())
  {
    foreach (var cust in repo.GetAll())
    {
      WriteLine($"->{cust.FirstName} {cust.LastName} is a Customer.");
    }
  }
  WriteLine("*********** Credit Risks ***********");
  using (var repo = new CreditRiskRepo())
  {
```

```
    foreach (var risk in repo.GetAll())
    {
      WriteLine($"->{risk.FirstName} {risk.LastName} is a Credit Risk!");
    }
  }
}
```

Вызовем этот метод в `Main()`, передав ему новый объект `Customer`:

```
WriteLine("***** Fun with ADO.NET EF Code First *****\n");
PrintAllCustomersAndCreditRisks();
var customerRepo = new CustomerRepo();
var customer = customerRepo.GetOne(4);
customerRepo.Context.Entry(customer).State = EntityState.Detached;
var risk = MakeCustomerARisk(customer);
PrintAllCustomersAndCreditRisks();
```

Если одна из операций (удаление записи из `Customer` или добавление записи в `CreditRisk`) потерпит неудачу, то обе операции будут отменены.

Исходный код. Проект `AutoLotTestDrive` доступен в подкаталоге `Chapter_23`.

Миграции Entity Framework

Эта версия сборки `AutoLotDAL.dll` создается из имеющегося кода, после чего создается база данных. Каждый раз, когда приложение запускается, база данных удаляется и воссоздается заново посредством процесса инициализации. Такой подход хорош при разработке, но после развертывания приложения в производственной среде нельзя удалять базу всякий раз, когда пользователи запускают приложение. Если модель изменится, то необходимо поддерживать базу данных в синхронизированном с ней состоянии. Именно здесь в игру вступают миграции EF. Перед созданием первой миграции мы внесем некоторые изменения, чтобы проиллюстрировать проблему. Откроем файл `Program.cs` и закомментируем следующую строку:

```
Database.SetInitializer(new MyDataInitializer());
```

На заметку! Как обсуждалось ранее, инициализатор удаляет и повторно создает базу данных либо при каждом запуске приложения, либо в случае изменения модели. Если не закомментировать строку с вызовом `SetInitializer()`, то проработать материал следующего раздела не удастся.

Обновление модели

Предположим, что возникла потребность внести несколько изменений в приложение, включая проверку параллелизма. Для этого во все таблицы будет добавлено свойство `Timestamp` (проверка параллелизма посредством EF рассматривается далее в главе). Вспомните из списка аннотаций данных, что для этого используется атрибут `Timestamp`. В SQL Server аннотации `Timestamp` отображаются на тип данных `RowVersion`, который в языке C# представляется с помощью типа `byte[]`. Это единственное изменение, которое будет произведено в классах `Inventory`, `Customer` и `Order`. Мы также добавим свойство `Timestamp` в `CreditRisk`, но планируем в следующем разделе внести в этот класс дополнительные изменения. Далее приведен модифицированный код классов `Inventory`, `Customer` и `Order`.

Класс Inventory

Вот код класса Inventory:

```
[Table("Inventory")]
public partial class Inventory
{
  [Key]
  public int CarId { get; set; }

  [StringLength(50)]
  public string Make { get; set; }

  [StringLength(50)]
  public string Color { get; set; }

  [StringLength(50)]
  public string PetName { get; set; }

  [Timestamp]
  public byte[] Timestamp { get; set; }

  public virtual ICollection<Order> Orders { get; set; } = new
HashSet<Order>();
}
```

Класс Customer

Вот код класса Customer:

```
public partial class Customer
{
  [Key]
  public int CustId { get; set; }

  [StringLength(50)]
  public string FirstName { get; set; }

  [StringLength(50)]
  public string LastName { get; set; }

  [Timestamp]
  public byte[] Timestamp { get; set; }

  [NotMapped]
  public string FullName => FirstName + " " + LastName;

  public virtual ICollection<Order> Orders { get; set; } = new
HashSet<Order>();
}
```

Класс Order

Вот код класса Order:

```
public partial class Order
{
  [Key, Required, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
  public int OrderId { get; set; }

  [Required]
  public int CustId { get; set; }

  [Required]
  public int CarId { get; set; }
```

```
  [Timestamp]
  public byte[] Timestamp { get; set; }
  [ForeignKey("CustId")]
  public virtual Customer Customer { get; set; }
  [ForeignKey("CarId")]
  public virtual Inventory Car { get; set; }
}
```

Класс CreditRisk

В дополнение к свойству `Timestamp` мы собираемся создать уникальный индекс на свойствах `FirstName` и `LastName` с применением аннотаций данных. Поскольку индекс составной, должно быть также указано имя индекса и порядок следования столбцов в индексе. В настоящем примере имя индекса выглядит как `IDX_CreditRisk_Name`, порядок следования столбцов в индексе — `LastName` и затем `FirstName`, к тому же индекс является уникальным. Ниже показан обновленный код:

```
public partial class CreditRisk
{
  [Key]
  public int CustId { get; set; }
  [StringLength(50)]
  [Index("IDX_CreditRisk_Name",IsUnique = true,Order=2)]
  public string FirstName { get; set; }
  [StringLength(50)]
  [Index("IDX_CreditRisk_Name",IsUnique = true,Order = 1)]
  public string LastName { get; set; }
  [Timestamp]
  public byte[] Timestamp { get; set; }
}
```

Тестирование приложения

Закомментируем все строки в методе `Main()` класса `Program` кроме вызова `PrintAllInventory()` (подобно следующему фрагменту кода) и запустим приложение.

```
static void Main(string[] args)
{
  WriteLine("***** Fun with ADO.NET EF Code First *****\n");
  PrintAllInventory();
  ReadLine();
}
```

Сгенерируется исключение `System.InvalidOperationException` с приведенным далее сообщением об ошибке:

```
The model backing the 'AutoLotEntities' context has changed since the
database was created. Consider using Code First Migrations to update the
database (http://go.microsoft.com/fwlink/?LinkId=238269).
```

Поддерживающий модель контекст AutoLotEntities изменился с момента создания базы данных. Обдумайте использование Code First Migrations для обновления базы данных (https://msdn.microsoft.com/ru-ru/data/jj591621).

Понятие миграций EF

Каким образом инфраструктура EF выясняет, что база данных и модель потеряли синхронизацию? Перед первым обращением к базе данных инфраструктура EF ищет таблицу по имени `__MigrationHistory` и сравнивает хеш-значение текущей модели EF с самым последним хеш-значением, хранящимся в этой таблице. Открыв ее в окне Server Explorer, можно увидеть одну запись. Она была автоматически создана EF при создании базы данных. Теперь, когда модель была изменена, необходимо создать новую запись миграции.

На заметку! При создании модели из существующей базы данных таблица `__MigrationHistory` *не* создается (во всяком случае, так было на момент написания книги). Почему это имеет значение? Когда создается экземпляр класса `DbContext` и перед первым обращением к базе данных из специального кода, инфраструктура EF проверяет хронологию миграций. Поскольку таблица `__MigrationHistory` не существует, генерируется последовательность исключений. Как вам хорошо известно, исключения могут быть затратными операциями и, таким образом, повлечь за собой проблему, касающуюся производительности. Даже если вы не планируете использовать миграции, вы должны включить их, как показано в следующем разделе.

Создание миграции базового уровня

Прежде всего, понадобится включить миграции для проекта. Откроем окно Package Manager Console (Консоль диспетчера пакетов), которое является инструментом командной строки для управления пакетами NuGet, выбрав пункт меню View⇨Other Windows⇨Package Manager Console (Вид⇨Другие окна⇨Консоль диспетчера пакетов). Удостоверимся, что в поле Default Project (Стандартный проект) указано имя `AutoLotDAL` и введем команду `enable-migrations` (рис. 23.12).

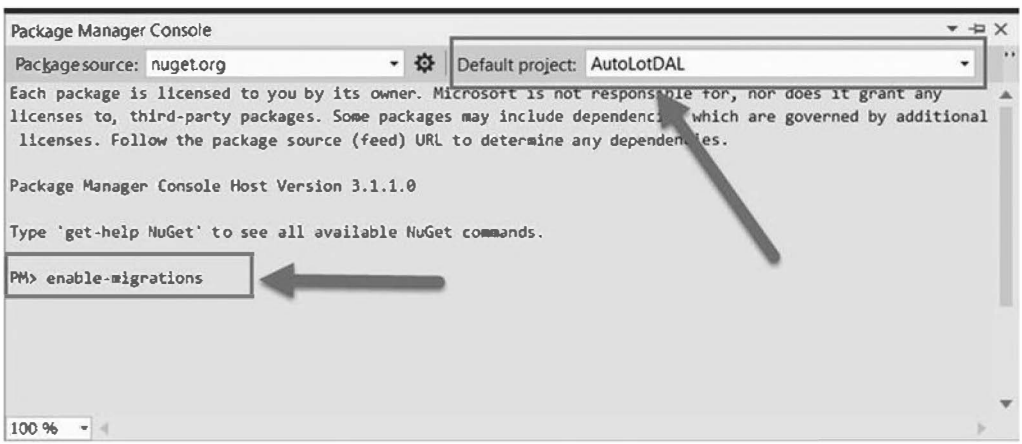


Рис. 23.12. Включение миграций для проекта `AutoLotDAL`

В результате создается папка `Migrations` с двумя файлами классов: `Configuration.cs` и `201510060510505_InitialCreate.cs`. Обратите внимание, что имя второго файла основано на дате и времени центрального процессора, за которым следует имя миграции. Такой формат имени позволяет инфраструктуре EF запускать миграции в корректном хронологическом порядке (если их больше одной). Из-за того, что при включении миграций имя не было указано, используется стандартное имя `InitialCreate`.

Откроем файл класса `InitialCreate.cs`. Этот класс имеет два метода с именами `Up()` и `Down()`. Метод `Up()` предназначен для применения изменений к базе данных, а метод `Down()` — для отката изменений. Инфраструктура EF построила базу данных, основываясь на модели до того, как в предыдущем разделе были внесены изменения, и заполнила таблицу `__MigrationHistory` хеш-значением таблиц модели и их полей. Если просмотреть класс `InitialCreate`, то легко заметить, что поля `Timestamp` и дополнительный индекс в таблице `CreditRisk` отсутствуют. Код этого класса должен выглядеть примерно так:

```
public partial class InitialCreate : DbMigration
{
  public override void Up()
  {
    CreateTable(
      "dbo.CreditRisks",
      c => new
        {
          CustId = c.Int(nullable: false, identity: true),
          FirstName = c.String(maxLength: 50),
          LastName = c.String(maxLength: 50),
        })
        .PrimaryKey(t => t.CustId);

    CreateTable(
     "dbo.Customers",
     c => new
       {
         CustId = c.Int(nullable: false, identity: true),
         FirstName = c.String(maxLength: 50),
         LastName = c.String(maxLength: 50),
       })
       .PrimaryKey(t => t.CustId);

    CreateTable(
      "dbo.Orders",
      c => new
        {
          OrderId = c.Int(nullable: false, identity: true),
          CustId = c.Int(nullable: false),
          CarId = c.Int(nullable: false),
        })
        .PrimaryKey(t => t.OrderId)
        .ForeignKey("dbo.Inventory", t => t.CarId, cascadeDelete: true)
        .ForeignKey("dbo.Customers", t => t.CustId, cascadeDelete: true)
        .Index(t => t.CustId)
        .Index(t => t.CarId);

    CreateTable(
      "dbo.Inventory",
      c => new
        {
          CarId = c.Int(nullable: false, identity: true),
          Make = c.String(maxLength: 50),
          Color = c.String(maxLength: 50),
          PetName = c.String(maxLength: 50),
        })
        .PrimaryKey(t => t.CarId);
  }
```

```
  public override void Down()
  {
    DropForeignKey("dbo.Orders", "CustId", "dbo.Customers");
    DropForeignKey("dbo.Orders", "CarId", "dbo.Inventory");
    DropIndex("dbo.Orders", new[] { "CarId" });
    DropIndex("dbo.Orders", new[] { "CustId" });
    DropTable("dbo.Inventory");
    DropTable("dbo.Orders");
    DropTable("dbo.Customers");
    DropTable("dbo.CreditRisks");
  }
}
```

Класс `Configuration.cs` **также содержит один метод и некоторый код в конструкторе. Код в конструкторе инструктирует EF об отключении автоматических миграций (что будет настройкой, используемой большую часть времени, т.к. нас интересует контроль над тем, как работают миграции) и устанавливает свойство** `ContextKey` **(базового класса) в полностью заданное имя производного от** `DbContext` **класса. Метод** `Seed()` **позволяет добавлять данные в базу; мы будем применять его очень скоро.**

```
internal sealed class Configuration : DbMigrationsConfiguration<AutoLotDA
L.EF.AutoLotEntities>
{
  public Configuration()
  {
    AutomaticMigrationsEnabled = false;
    ContextKey = "AutoLotDAL.EF.AutoLotEntities";
  }
  protected override void Seed(AutoLotDAL.EF.AutoLotEntities context)
  {
  }
}
```

Чтобы создать миграцию, необходимо ввести команду `add-migration TimeStamps` **в окне Package Manager Console. Команда** `add-migration` **заставляет EF взять хеш-значение текущей модели и сравнить его с самым последним хеш-значением в таблице** `__MigrationHistory`**. Второй параметр — это имя миграции, которое может быть любым, но содержательным. В результате выполнения этой команды (рис. 23.13) внутри папки** `Migrations` **создается новый файл, имеющий имя** `<метка-времени>_TimeStamps.cs`.

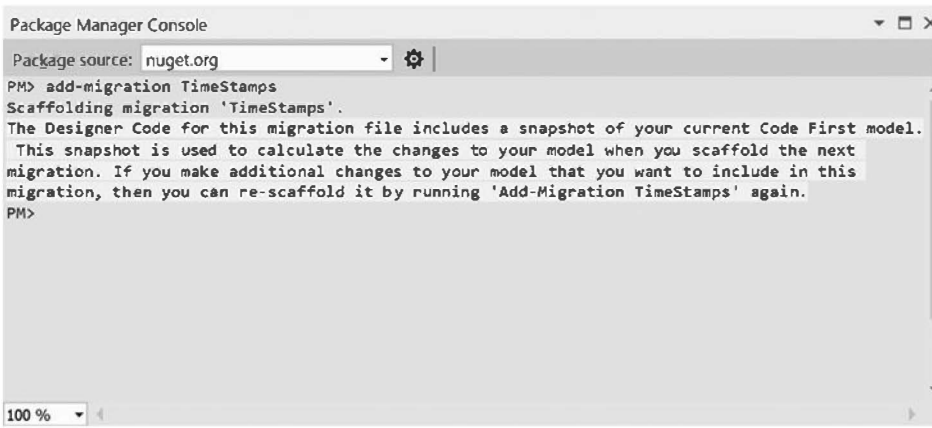


Рис. 23.13. Создание начальной миграции

Откроем новый файл (имеющий имя, скажем, 201510062307304_TimeStamps.cs) и просмотрим его содержимое. Здесь снова есть метод Up(), применяющий изменения, и метод Down(), откатывающий изменения. Ниже показано содержимое файла:

```
public partial class TimeStamps : DbMigration
{
  public override void Up()
  {
    AddColumn("dbo.CreditRisks", "Timestamp",
      c => c.Binary(nullable: false, fixedLength: true, timestamp:
                    true, storeType: "rowversion"));
    AddColumn("dbo.Customers", "Timestamp",
      c => c.Binary(nullable: false, fixedLength: true, timestamp:
                    true, storeType: "rowversion"));
    AddColumn("dbo.Orders", "Timestamp",
      c => c.Binary(nullable: false, fixedLength: true, timestamp:
                    true, storeType: "rowversion"));
    AddColumn("dbo.Inventory", "Timestamp",
      c => c.Binary(nullable: false, fixedLength: true, timestamp:
                    true, storeType: "rowversion"));
    CreateIndex("dbo.CreditRisks", new[] { "LastName", "FirstName" },
                unique: true, name: "IDX_CreditRisk_Name");
  }
  public override void Down()
  {
    DropIndex("dbo.CreditRisks", "IDX_CreditRisk_Name");
    DropColumn("dbo.Inventory", "Timestamp");
    DropColumn("dbo.Orders", "Timestamp");
    DropColumn("dbo.Customers", "Timestamp");
    DropColumn("dbo.CreditRisks", "Timestamp");
  }
}
```

Финальная задача заключается в обновлении базы данных. В окне Package Manager Console понадобится ввести команду update-database, которая по завершении сообщит о применении миграции. Откроем окно Server Explorer и обновим узел Tables (Таблицы). Появится таблица __MigrationHistory. Если выбрать для этой таблицы пункт меню Show Table Data (Показать данные таблицы), то отобразится ее содержимое (рис. 23.14).

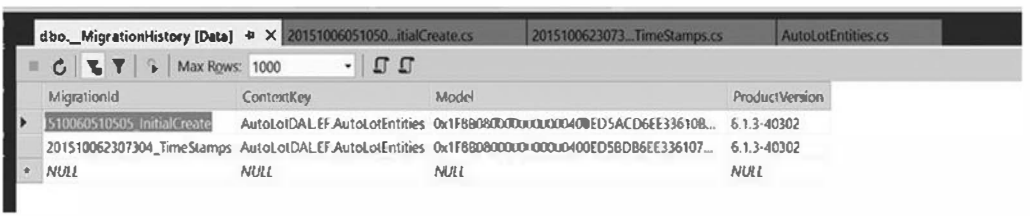


Рис. 23.14. Содержимое таблицы __MigrationHistory

Начальное заполнение базы данных

Метод Seed() в Configure.cs задействует метод AddOrUpdate() класса DbSet. Метод AddOrUpdate() принимает два параметра; первый из них — это лямбда-выражение, которое представляет уникальное идентифицирующее поле (или поля) для обновления, а второй — запись, подлежащая добавлению (или обновлению) внутри базы данных.

Вот основной синтаксис:

```
context.Customers.AddOrUpdate(c => c.CustId,
   new Customer {CustId = 1, FirstName="Foo",LastName="Bar"});
```

В этом примере не производится проверка по первичному ключу, потому что имеется идентичность. Необходимо проверять уникальность комбинации `FirstName` и `LastName`. Для использования составного идентификатора вместо одиночного свойства создается анонимный объект, идентифицирующий поля:

```
context.Customers.AddOrUpdate(c => new {c.FirstName,c.LastName},
   new Customer { FirstName="Foo",LastName="Bar"});
```

Скопируем код из класса `DataInitialize` в метод `Seed()`. Изменим вызовы `Add()` на `AddOrUpdate()`, как показано ниже:

```
protected override void Seed(AutoLotDAL.EF.AutoLotEntities context)
{
  var customers = new List<Customer>
  {
    new Customer {FirstName = "Dave", LastName = "Brenner"},
    new Customer {FirstName = "Matt", LastName = "Walton"},
    new Customer {FirstName = "Steve", LastName = "Hagen"},
    new Customer {FirstName = "Pat", LastName = "Walton"},
    new Customer {FirstName = "Bad", LastName = "Customer"},
  };
  customers.ForEach(x =>
    context.Customers.AddOrUpdate(c => new { c.FirstName,c.LastName},x));
  var cars = new List<Inventory>
  {
    new Inventory {Make = "VW", Color = "Black", PetName = "Zippy"},
    new Inventory {Make = "Ford", Color = "Rust", PetName = "Rusty"},
    new Inventory {Make = "Saab", Color = "Black", PetName = "Mel"},
    new Inventory {Make = "Yugo", Color = "Yellow", PetName = "Clunker"},
    new Inventory {Make = "BMW", Color = "Black", PetName = "Bimmer"},
    new Inventory {Make = "BMW", Color = "Green", PetName = "Hank"},
    new Inventory {Make = "BMW", Color = "Pink", PetName = "Pinky"},
    new Inventory {Make = "Pinto", Color = "Black", PetName = "Pete"},
    new Inventory {Make = "Yugo", Color = "Brown", PetName = "Brownie"},
  };
  cars.ForEach(x =>
    context.Inventory.AddOrUpdate(i => new { i.Make, i.Color, i.PetName }, x));
  var orders = new List<Order>
  {
    new Order {Car = cars[0], Customer = customers[0]},
    new Order {Car = cars[1], Customer = customers[1]},
    new Order {Car = cars[2], Customer = customers[2]},
    new Order {Car = cars[3], Customer = customers[3]},
  };
  orders.ForEach(x =>
     context.Orders.AddOrUpdate(o => new { o.CarId, o.CustId }, x));
  context.CreditRisks.AddOrUpdate(c => new { c.FirstName, c.LastName },
    new CreditRisk
    {
      CustId = customers[4].CustId,
      FirstName = customers[4].FirstName,
      LastName = customers[4].LastName,
    });
}
```

Методы `Seed()` выполняются каждый раз, когда в окне Package Manager Console запускается команда `update-database`. Для начального заполнения базы данных потребуется открыть окно Package Manager Console и ввести команду `update-database`.

Возвращение к тесту транзакций

Теперь, когда таблица `CreditRisk` имеет уникальный индекс, основанный на имени и фамилии клиента, обновим метод `MakeACustomerRisk()`, чтобы добавить новую запись два раза. По причине создания неявной транзакции, когда вызывается метод `SaveChanges()`, запись о заказчике не только не будет добавлена в таблицу `CreditRisk`, но и не будет удалена из таблицы `Customer`. Поместим вызов `SaveChanges()` в блок `try/catch`, перехватывающий исключение `DbUpdateException`. В блоке `catch` выведем информацию об исключении на консоль. После запуска приложения можно видеть, что в действительности изменения в базу данных не были внесены, и в окно консоли выводятся детали исключения. Далее приведен обновленный код:

```
private static CreditRisk MakeCustomerARisk(Customer customer)
{
  using (var context = new AutoLotEntities())
  {
    context.Customers.Attach(customer);
    context.Customers.Remove(customer);
    var creditRisk = new CreditRisk()
    {
      FirstName = customer.FirstName,
      LastName = customer.LastName
    };
    context.CreditRisks.Add(creditRisk);
    var creditRiskDupe = new CreditRisk()
    {
      FirstName = customer.FirstName,
      LastName = customer.LastName
    };
    context.CreditRisks.Add(creditRiskDupe);
    try
    {
      context.SaveChanges();
    }
    catch (DbUpdateException ex)
    {
      WriteLine(ex);
    }
    return creditRisk;
  }
}
```

Параллелизм

Проблемы параллелизма в многопользовательских приложениях являются распространенным явлением. Если в приложении не производится проверка на возникновение проблем параллелизма, то когда два пользователя обновляют одну и ту же запись, в выигрыше оказывается последний из них. Это может полностью подходить для приложения, но если нет, то EF и SQL Server предоставляют удобный механизм для проверки наличия конфликтов параллелизма.

Добавление к классам модели свойств Timestamp в предыдущем разделе привело к изменению того, как EF строит и запускает запросы, которые обновляют или удаляют записи из базы данных. Запрос на удаление теперь ищет не только первичный ключ (CarId), но также и столбец Timestamp. Например, сгенерированный оператор SQL выглядит подобно следующему:

```
Execute NonQuery "DELETE [dbo].[Inventory] WHERE (([CarId] = @0) AND
  ([Timestamp] = @1))"
```

Инфраструктура EF автоматически добавляет столбец Timestamp как часть конструкции WHERE оператора DELETE. Это предотвращает влияние одного пользователя (или процесса) на изменения, производимые другим пользователем (или процессом). Если такой оператор удаления не работает с самой последней версией записи, то ничего обновляться не будет и сгенерируется исключение DbUpdateConcurrencyException.

Корректировка классов хранилищ

Методы Delete(int id) и DeleteAsync(int id) в классах хранилищ теперь при вызове будут отказывать, т.к. им не передается свойство Timestamp. Скорректируем определения методов в интерфейсе IRepo<T> для приема значения Timestamp. Вот обновленный код:

```
int Delete(int id, byte[] timeStamp);
Task<int> DeleteAsync(int id, byte[] timeStamp);
```

Теперь модифицируем все классы хранилищ, чтобы они задействовали новые сигнатуры методов. Ниже показан соответствующий код из класса InventoryRepo (остальные классы хранилищ следуют такому же шаблону; обновленный код можно найти в загружаемом коде примеров):

```
public int Delete(int id, byte[] timeStamp)
{
  Context.Entry(new Inventory()
  {
    CarId=id,
    Timestamp = timeStamp
  }).State = EntityState.Deleted;
  return SaveChanges();
}
public Task<int> DeleteAsync(int id, byte[] timeStamp)
{
  Context.Entry(new Inventory()
  {
    CarId = id,
    Timestamp = timeStamp
  }).State = EntityState.Deleted;
  return SaveChangesAsync();
}
```

Тестирование параллелизма

В следующем далее коде демонстрируется проверка параллелизма для записи Inventory. Код имитирует обновление одной записи двумя разными пользователями. Когда пользователи получают копии записи из базы данных, временные метки в них одинаковы. После того, как один пользователь обновил свою копию записи, временная метка обновляется SQL Server, но у второго пользователя копия записи не обновлялась, поэтому значение временной метки остается прежним. Если второй пользователь попы-

тается сохранить запись, то временные метки не совпадут, вызов SaveChanges() ничего не обновит и сгенерируется исключение. Ниже показан код:

```
private static void UpdateRecordWithConcurrency()
{
  var car = new Inventory()
  { Make = "Yugo", Color = "Brown", PetName = "Brownie" };
  AddNewRecord(car);
  var repo1 = new InventoryRepo();
  var car1 = repo1.GetOne(car.CarId);
  car1.PetName = "Updated";

  var repo2 = new InventoryRepo();
  var car2 = repo2.GetOne(car.CarId);
  car2.Make = "Nissan";

  repo1.Save(car1);
  try
  {
    repo2.Save(car2);
  }
  catch (DbUpdateConcurrencyException ex)
  {
    WriteLine(ex);
  }
  RemoveRecordById(car1.CarId, car1.Timestamp);
}
```

Код всего лишь сообщает о возникновении проблемы. Понадобится решить, какие действия должны предприниматься в случае появления ошибки параллелизма. Они будут зависеть от специфичных бизнес-требований.

Перехват

Последней темой, касающейся EF, является перехват. В рассмотренных ранее примерах вы видели, что "за кулисами" происходит немало "магических" действий для перемещения данных из хранилища в объектную модель и наоборот. Перехват — это выполнение кода на различных фазах такого процесса.

Интерфейс IDbCommandInterceptor

Все начинается с интерфейса IDbCommandInterceptor:

```
public interface IDbCommandInterceptor : IDbInterceptor
{
  void NonQueryExecuted(DbCommand command,
    DbCommandInterceptionContext<int> interceptionContext);
  void NonQueryExecuting(DbCommand command,
    DbCommandInterceptionContext<int> interceptionContext);
  void ReaderExecuted(DbCommand command,
    DbCommandInterceptionContext<DbDataReader> interceptionContext);
  void ReaderExecuting(DbCommand command,
    DbCommandInterceptionContext<DbDataReader> interceptionContext);
  void ScalarExecuted(DbCommand command,
    DbCommandInterceptionContext<object> interceptionContext);
  void ScalarExecuting(DbCommand command,
    DbCommandInterceptionContext<object> interceptionContext);
}
```

Этот интерфейс содержит методы, которые вызываются инфраструктурой EF непосредственно до и после возникновения определенных событий. Например, метод `ReaderExecuting()` вызывается прямо *перед* запуском средства чтения, а метод `ReaderExecuted()` — сразу *после* того, как средство чтения выполнилось. Мы рассмотрим пример вывода на консоль сообщений в каждом из упомянутых методов. В производственной системе логика будет соответствовать существующим требованиям.

Добавление перехвата в AutoLotDAL

Добавим в проект `AutoLotDAL` новую папку по имени `Interception` и поместим в нее новый класс `ConsoleWriterInterceptor`. Сделаем этот класс открытым, добавим оператор `using` для пространства имен `System.Data.Entity.Infrastructure.Interception` и унаследуем его от `IDbCommandInterceptor`. После реализации необходимых членов код должен выглядеть следующим образом:

```
public class ConsoleWriterInterceptor : IDbCommandInterceptor
{
  public void NonQueryExecuting(DbCommand command,
    DbCommandInterceptionContext<int> interceptionContext)
  {
  }

  public void NonQueryExecuted(DbCommand command,
    DbCommandInterceptionContext<int> interceptionContext)
  {
  }

  public void ReaderExecuting(DbCommand command,
    DbCommandInterceptionContext<DbDataReader> interceptionContext)
  {
  }

  public void ReaderExecuted(DbCommand command,
    DbCommandInterceptionContext<DbDataReader> interceptionContext)
  {
  }

  public void ScalarExecuting(DbCommand command,
    DbCommandInterceptionContext<object> interceptionContext)
  {
  }

  public void ScalarExecuted(DbCommand command,
    DbCommandInterceptionContext<object> interceptionContext)
  {
  }
}
```

Для простоты мы собираемся только выводить на консоль информацию о том, является ли вызов асинхронным, и текст команды. Добавим оператор `using static System.Console;` и закрытый метод по имени `WriteInfo()`, который принимает параметры `bool` и `string`. Ниже показан его код:

```
private void WriteInfo(bool isAsync, string commandText)
{
  WriteLine($"IsAsync: {isAsync}, Command Text: {commandText}");
}
```

В каждый метод интерфейса поместим вызов `WriteInfo()` следующего вида:

```
WriteInfo(interceptionContext.IsAsync,command.CommandText);
```

Регистрация перехватчика

Перехватчики могут быть зарегистрированы в коде или в конфигурационном файле приложения. Регистрация в коде изолирует их от изменений конфигурационного файла и по этой причине гарантирует, что они всегда будут зарегистрированы. Если нужна более высокая гибкость, то конфигурационный файл может оказаться лучшим вариантом. В рассматриваемом примере мы зарегистрируем перехватчик в коде.

Откроем файл AutoLotEntities.cs и добавим следующие операторы using:

```
using System.Data.Entity.Infrastructure;
using System.Data.Entity.Infrastructure.Interception;
```

Поместим в конструктор такую строку кода:

```
DbInterception.Add(new ConsoleWriterInterceptor());
```

Выполнив один из тестовых методов, приведенных ранее в главе, можно заметить дополнительный вывод в окне консоли. Хотя данный пример прост, он демонстрирует возможности перехватчика.

На заметку! Класс DbCommandInterceptionContext<T> содержит намного больше членов, чем те, что представлены здесь. За дополнительными сведениями обращайтесь в документацию .NET Framework 4.6 SDK.

Добавление перехватчика DatabaseLogger

Инфраструктура EF теперь поставляется со встроенным регистрирующим перехватчиком, который позволяет выполнять простую регистрацию в журнале. Чтобы добавить такую возможность, откроем файл AutoLotEntities.cs и закомментируем код нашего консольного регистратора. Добавим статический член только для чтения типа DatabaseLogger (из пространства имен System.Data.Entity.Infrastructure.Interception). Его конструктор принимает два параметра; первый из них — имя журнального файла, а второй необязательный параметр указывает, должна ли информация дописываться в журнальный файл (стандартным значением является false). Вызовем в конструкторе метод StartLogging() объекта перехватчика и добавим этот объект в список перехватчиков. Ниже показан модифицированный код:

```
static readonly DatabaseLogger DatabaseLogger =
  new DatabaseLogger("sqllog.txt", true);
public AutoLotEntities() : base("name=AutoLotConnection")
{
  // DbInterception.Add(new ConsoleWriterInterceptor());
  DatabaseLogger.StartLogging();
  DbInterception.Add(DatabaseLogger);
}
```

Последнее изменение касается использования реализации DbContext шаблона IDisposable для прекращения регистрации в журнале и удаления объекта перехватчика:

```
protected override void Dispose(bool disposing)
{
  DbInterception.Remove(DatabaseLogger);
  DatabaseLogger.StopLogging();
  base.Dispose(disposing);
}
```

События ObjectMaterialized и SavingChanges

Класс ObjectContext содержит два события, ObjectMaterialized и SavingChanges. Эти события могут оградить вас от создания перехватчика — разумеется, при условии, что они удовлетворяют вашим нуждам! Событие ObjectMaterialized инициируется, когда объект реконструируется из хранилища данных, а событие SavingChanges происходит, когда данные объекта распространяются в хранилище, сразу после вызова метода SaveChanges() контекста.

Доступ к объектному контексту

Как вы помните, контекст является производным от класса DbContext. К счастью, он также расширяет интерфейс IObjectContextAdapter. Чтобы обратиться к ObjectContext, необходимо привести AutoLotEntities к IObjectContextAdapter. Это делается внутри конструктора:

```
public AutoLotEntities(): base("name=AutoLotConnection")
{
  // Код перехватчика.
  var context = (this as IObjectContextAdapter).ObjectContext;
  context.ObjectMaterialized += OnObjectMaterialized;
  context.SavingChanges += OnSavingChanges;
}
private void OnSavingChanges(object sender, EventArgs eventArgs)
{
}
private void OnObjectMaterialized(object sender,
System.Data.Entity.Core.Objects.ObjectMaterializedEventArgs e)
{
}
```

Событие ObjectMaterialized

Аргументы события ObjectMaterialized предоставляют доступ к реконструируемой сущности. Хотя здесь данное событие не применяется, оно понадобится в главе 30. Пока достаточно знать, что это событие инициируется немедленно после заполнения свойств класса модели инфраструктурой EF и перед тем, как контекст предоставит его вызывающему коду.

Событие SavingChanges

Как уже упоминалось, событие SavingChanges возникает сразу после вызова метода SaveChanges() (на DbContext), но перед обновлением базы данных. За счет обращения к объекту ObjectContext, переданному обработчику события, все сущности в транзакции доступны через свойство ObjectStateEntry класса DbContext. Некоторые основные свойства кратко описаны в табл. 23.6.

Таблица 23.6. Основные свойства класса ObjectStateEntry

Член	Описание
CurrentValues	Текущие значения свойств сущности
OriginalValues	Исходные значения свойств сущности
Entity	Сущность, представленная объектом ObjectStateEntry
State	Текущее состояние сущности (например, Modified, Added, Deleted)

Класс `ObjectStateEntry` также предлагает набор методов, которые можно использовать в отношении сущности. Некоторые из них перечислены в табл. 23.7.

Таблица 23.7. Основные методы класса `ObjectStateEntry`

Метод	Описание
`AcceptChanges`	Принимает текущие значения как исходные
`ApplyCurrentValues`	Устанавливает текущие значения в соответствие со значениями предоставленного объекта
`ApplyOriginalValues`	Устанавливает исходные значения в соответствие со значениями предоставленного объекта
`ChangeState`	Обновляет состояние сущности
`GetModifiedProperties`	Возвращает имена всех измененных свойств
`IsPropertyChanges`	Проверяет специфичное свойство на предмет изменений
`RejectPropertyChanges`	Отклоняет любые изменения, внесенные в свойство

В результате есть возможность писать код, который отклоняет любые изменения цвета автомобиля, если новый цвет является красным:

```
private void OnSavingChanges(object sender, EventArgs eventArgs)
{
  // Параметр sender имеет тип ObjectContext.
  // Можно получать текущие и исходные значения,
  // а также отменять/модифицировать операцию сохранения
  // любым желаемым образом.
  var context = sender as ObjectContext;
  if (context == null) return;
  foreach (ObjectStateEntry item in
    context.ObjectStateManager.GetObjectStateEntries(
      EntityState.Modified | EntityState.Added))
  {
    // Делать здесь что-то важное.
    if ((item.Entity as Inventory)!=null)
    {
      var entity = (Inventory) item.Entity;
      if (entity.Color == "Red")
      {
        item.RejectPropertyChanges(nameof(entity.Color));
      }
    }
  }
}
```

Исходный код. Обновленный проект `AutoLotDAL` доступен в подкаталоге `Chapter_23`.

Развертывание базы данных в системе SQL Server Express

В качестве финального шага мы развернем базу данных в системе SQL Server Express. Задача сводится к изменению строки подключения и выполнению команды `update-database`. Откроем файл `App.config` из проекта `AutoLotDAL` и обновим строку подключения для указания на SQL Server Express. Обратите внимание, что измене-

но также имя стандартного каталога, поскольку база данных AutoLot уже существует. Точная строка будет зависеть о того, каким образом установлена система SQL Server Express, но должна выглядеть похожей на показанную ниже:

```
<connectionStrings>
  <add name="AutoLotConnection"
       connectionString="data source=.\SQLEXPRESS2014;initial
catalog=AutoLot2;integrated security=True;MultipleActiveResultSets=True;
App=EntityFramework"
  providerName="System.Data.SqlClient" />
</connectionStrings>
```

После запуска команды update-database обновления применяются (рис. 23.15).

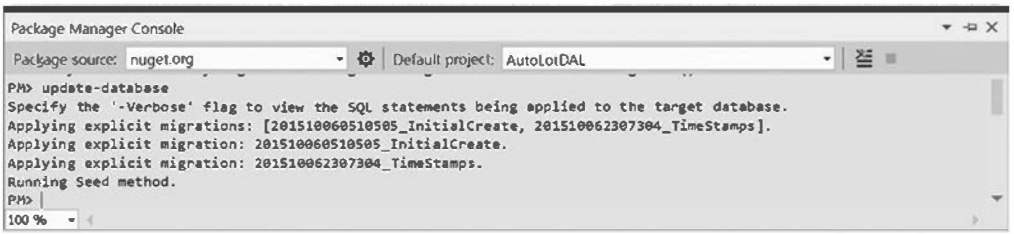


Рис. 23.15. Развертывание базы данных в системе SQL Server Express

На заметку! Если возникает проблема с тем, что миграции не выполняются, то окно Package Manager Console может использовать консольное приложение как целевое. Исправить ситуацию поможет следующая команда:

```
Update-Database -ProjectName AutoLotDAL -StartUpProjectName AutoLotDAL
```

Резюме

В последних трех главах был представлен обзор трех подходов к манипуляции данными с применением ADO.NET, в частности — подключенный и автономный уровни, а также инфраструктура Entity Framework. Каждый подход обладает своими достоинствами и многие разрабатываемые вами приложения, скорее всего, будут задействовать разнообразные аспекты каждого из них. Не забывайте, что здесь был предложен только краткий экскурс в темы, которые можно найти в рамках набора технологий ADO.NET. Для получения исчерпывающих сведений рекомендуем обратиться в документацию .NET Framework 4.6 SDK.

В этой главе формальное исследование программирования для баз данных с использованием ADO.NET завершилось рассмотрением роли инфраструктуры Entity Framerowk. Инфраструктура EF позволяет программировать с помощью концептуальной модели, которая точно отображается на предметную область. Несмотря на то что форму сущностей можно изменять как угодно, исполняющая среда EF гарантирует, что измененные данные модели отображаются на корректные данные физической таблицы.

В ходе изучения вы узнали об аннотациях данных, которые представляют собой один из способов описания отображения между моделью предметной области и моделью базы данных. Вы увидели, каким образом инфраструктура EF обрабатывает транзакции, научились создавать, сохранять и удалять данные, а также ознакомились с состоянием сущностей.

Было показано, как с помощью миграций базы данных синхронизировать изменения, внесенные в модель, с базой данных, проверять наличие ошибок, связанных с параллелизмом, и добавлять регистрацию в журнале и перехват. Наконец, в рамках тестирования сборки AutoLotDAL.dll было предложено множество разнообразных примеров.

ГЛАВА 24

Введение в LINQ to XML

Как разработчик приложений .NET, вы будете встречать данные, основанные на XML, в самых разнообразных местах. Конфигурационные файлы для обычных и веб-приложений хранят информацию в формате XML. Инфраструктура Windows Presentation Foundation использует основанную на XML грамматику (XAML) для представления настольных графических пользовательских интерфейсов. Объекты `DataSet` из ADO.NET могут легко сохранять (или загружать) данные в формате XML. Даже инфраструктура Windows Communication Foundation хранит многочисленные параметры в виде корректно сформатированных строк XML.

Хотя данные XML действительно вездесущи, исторически сложилось так, что программирование с применением XML было утомительным, громоздким и очень сложным, если не владеть довольно большим числом технологий XML (XPath, XQuery, XSLT, DOM, SAX и т.д.). В самом первом выпуске .NET была предоставлена специальная сборка по имени `System.Xml.dll`, предназначенная для программирования с использованием документов XML. В ней находится несколько пространств имен и типов для разнообразных технологий программирования XML, а также ряд API-интерфейсов XML, специфичных для .NET, таких как классы `XmlReader/XmlWriter`.

В наши дни большинство программистов предпочитают взаимодействовать с данными XML с применением API-интерфейса LINQ to XML. Вы увидите в этой главе, что программная модель LINQ to XML позволяет выражать структуру данных XML в коде и предлагает намного более простой способ создания, манипулирования, загрузки и сохранения данных XML. Наряду с тем, что LINQ to XML можно использовать только в качестве упрощенного способа создания документов XML, для быстрого запрашивания информации из них довольно легко задействовать выражения запросов LINQ.

История о двух API-интерфейсах XML

Когда появилась первая версия платформы .NET, программисты имели возможность манипулировать документами XML с применением типов из сборки `System.Xml.dll`. Используя содержащиеся в ней пространства имен и типы, можно было генерировать данные XML в памяти и сохранять их в дисковом хранилище. Вдобавок сборка `System.Xml.dll` предоставляла типы, позволяющие загружать документ XML в память, искать в нем специфические узлы, проверять документ на соответствие заданной схеме и выполнять другие распространенные задачи.

Несмотря на то что эта первоначальная библиотека успешно применялась во многих проектах .NET, работа с ее типами была несколько запутанной (мягко говоря), поскольку программная модель не имела никакого отношения к структуре самого документа XML. Например, пусть необходимо создать документ XML в памяти и сохранить его в файловой системе. В случае использования типов из `System.Xml.dll` можно написать код, подобный показанному ниже (первым делом нужно создать новый проект консоль-

ного приложения по имени LinqToXmlFirstLook и импортировать пространство имен System.Xml):

```
private static void BuildXmlDocWithDOM()
{
  // Создать новый документ XML в памяти.
  XmlDocument doc = new XmlDocument();

  // Заполнить документ корневым элементом по имени <Inventory>.
  XmlElement inventory = doc.CreateElement("Inventory");

  // Создать подэлемент по имени <Car> с атрибутом ID.
  XmlElement car = doc.CreateElement("Car");
  car.SetAttribute("ID", "1000");

  // Построить данные внутри элемента <Car>.
  XmlElement name = doc.CreateElement("PetName");
  name.InnerText = "Jimbo";
  XmlElement color = doc.CreateElement("Color");
  color.InnerText = "Red";
  XmlElement make = doc.CreateElement("Make");
  make.InnerText = "Ford";

  // Добавить к элементу <Car> элементы <PetName>, <Color> и <Make>.
  car.AppendChild(name);
  car.AppendChild(color);
  car.AppendChild(make);

  // Добавить к элементу <Inventory> элемент <Car>.
  inventory.AppendChild(car);

  // Вставить завершенный XML в объект XmlDocument и сохранить в файле.
  doc.AppendChild(inventory);
  doc.Save("Inventory.xml");
}
```

Вызвав этот метод внутри Main(), можно увидеть, что файл Inventory.xml (находящийся в папке bin\Debug) содержит следующие данные:

```
<Inventory>
 <Car ID="1000">
  <PetName>Jimbo</PetName>
  <Color>Red</Color>
  <Make>Ford</Make>
 </Car>
</Inventory>
```

Хотя метод BuildXmlDocWithDOM() работает ожидаемым образом, уместно сделать несколько замечаний. Программная модель System.Xml.dll — это реализация от Microsoft спецификации W3C DOM (Document Object Model — объектная модель документа). Согласно такой модели документ XML строится снизу вверх. Сначала создается документ, затем элементы и, наконец, элементы добавляются в документ. Чтобы выразить это в коде, потребуется написать довольно много вызовов методов из классов XmlDocument и XmlElement (помимо прочих).

В приведенном примере для построения очень простого документа XML понадобилось 16 строк кода (без учета комментариев). Создание более сложного документа с помощью сборки System.Xml.dll требует написания гораздо большего объема кода. Несмотря на то что код определенно можно упростить, выполняя построение узлов посредством циклических и условных конструкций, факт остается фактом — тело кода имеет минимум визуального отражения финального дерева XML.

Интерфейс LINQ to XML как лучшая модель DOM

Альтернативный способ построения, манипулирования и запрашивания документов XML предлагает API-интерфейс LINQ to XML, в котором применяется намного более функциональный подход по сравнению с моделью DOM из пространства имен `System.Xml`. Вместо построения документа XML за счет индивидуального формирования элементов и обновления дерева XML через набор вызовов методов код пишется сверху вниз:

```
private static void BuildXmlDocWithLINQToXml()
{
  // Создать документ XML в более "функциональной" манере.
  XElement doc =
   new XElement("Inventory",
    new XElement("Car", new XAttribute("ID", "1000"),
      new XElement("PetName", "Jimbo"),
      new XElement("Color", "Red"),
      new XElement("Make", "Ford")
    )
  );
  // Сохранить документ в файле.
  doc.Save("InventoryWithLINQ.xml");
}
```

Здесь используется новый набор типов из пространства имен `System.Xml.Linq`, а именно — `XElement` и `XAttribute`. Вызов метода `BuildXmlDocWithLINQToXml()` внутри `Main()` приводит к получению тех же самых данных XML, но на этот раз с гораздо меньшими усилиями. Обратите внимание, что благодаря аккуратным отступам исходный код теперь имеет ту же общую структуру, что и результирующий документ XML. Это очень удобно и само по себе, но вдобавок оцените, насколько компактнее данный код по сравнению с предыдущим примером (сэкономлено около 10 строк).

В коде не применяются выражения запросов LINQ, а просто с помощью типов из пространства имен `System.Xml.Linq` в памяти генерируется документ XML, который затем сохраняется в файле. Фактически API-интерфейс LINQ to XML использовался как лучшая модель DOM. Позже в главе вы увидите, что классы из `System.Xml.Linq` поддерживают LINQ и могут быть целью для той же разновидности запросов LINQ, которая рассматривалась в главе 12.

По мере изучения LINQ to XML, скорее всего, вы начнете отдавать предпочтение этому API-интерфейсу перед первоначальными библиотеками XML платформы .NET. Это не означает, что вы никогда не станете применять пространство имен из библиотеки `System.Xml.dll`, но случаев выбора `System.Xml.dll` для новых проектов будет значительно меньше.

Синтаксис литералов Visual Basic как лучший способ работы с LINQ to XML

Прежде чем приступить к формальным исследованиям LINQ to XML с точки зрения языка C#, имеет смысл кратко упомянуть о том, что язык Visual Basic переносит функциональный подход этого API-интерфейса на следующий уровень. В Visual Basic можно определять *литералы XML*, которые позволяют присваивать объекту `XElement` поток встроенной разметки XML прямо в коде. Предполагая, что есть проект VB, можно создать следующий метод:

```
Public Class XmlLiteralExample
  Public Sub MakeXmlFileUsingLiterals()
    ' Обратите внимание на возможность встраивания данных XML в XElement.
    Dim doc As XElement =
      <Inventory>
        <Car ID="1000">
          <PetName>Jimbo</PetName>
          <Color>Red</Color>
          <Make>Ford</Make>
        </Car>
      </Inventory>
    ' Сохранить в файле.
    doc.Save("InventoryVBStyle.xml")
  End Sub
End Class
```

После того, как компилятор VB обработал литерал XML, он отобразит данные XML на корректную внутреннюю объектную модель LINQ to XML. На самом деле во время работы с LINQ to XML внутри проекта VB синтаксис литералов XML воспринимается IDE-средой просто как сокращенное обозначение связанного кода. На рис. 24.1 обратите внимание, что применение операции точки к закрывающему дескриптору `</Inventory>` приводит к отображению тех же самых членов, как и в случае применения этой операции к строго типизированному `XElement`.

Хотя эта книга посвящена языку программирования C#, некоторые разработчики считают, что поддержка XML в VB является непревзойденной. Даже если вы из тех, кто не может представить себе работу с языком из семейства BASIC, все равно рекомендуется изучить синтаксис литералов VB в документации .NET Framework 4.6 SDK. Все процедуры манипуляций с данными XML могут быть вынесены в отдельную сборку `*.dll`, так что вполне допустимо использовать для них VB!

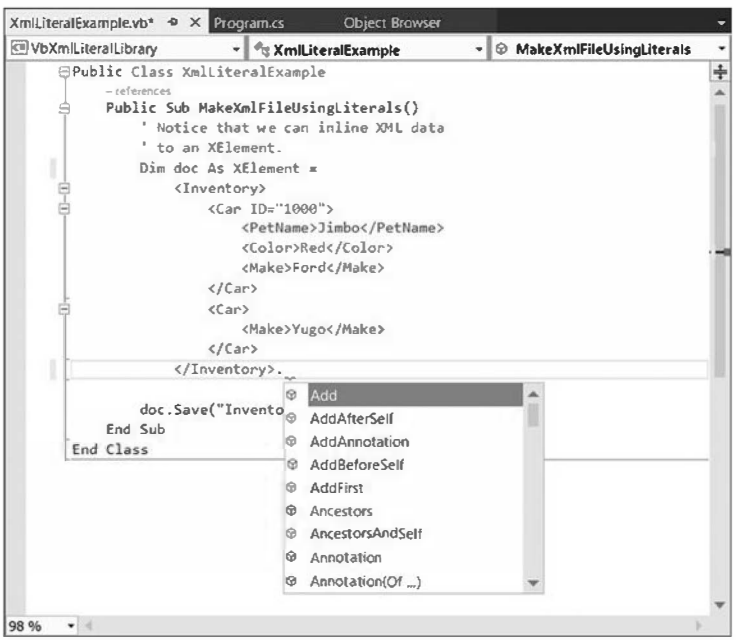


Рис. 24.1. Синтаксис литералов XML в VB представляет собой сокращение для работы с объектной моделью LINQ to XML

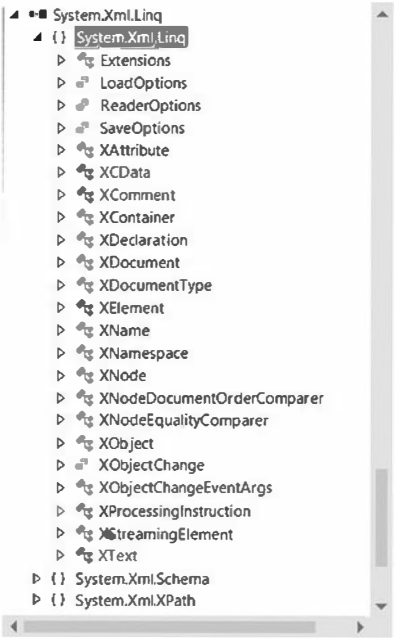


Рис. 24.2. Пространства имен `System.Xml.Linq.dll`

Члены пространства имен `System.Xml.Linq`

Несколько неожиданно, но в основной сборке LINQ to XML (`System.Xml.Linq.dll`) определено относительно небольшое количество типов в трех разных пространствах имен: `System.Xml.Linq`, `System.Xml.Schema` и `System.Xml.XPath` (рис. 24.2).

Главное пространство имен, `System.Xml.Linq`, содержит управляемый набор классов, которые представляют разнообразные аспекты документа XML (элементы и их атрибуты, пространства имен XML, комментарии XML, инструкции обработки и т.д.). В табл. 24.1 кратко описаны избранные основные члены `System.Xml.Linq`.

На рис. 24.3 показана цепочка наследования основных классов.

Таблица 24.1. Избранные члены пространства имен `System.Xml.Linq`

Член	Описание
`XAttribute`	Представляет атрибут XML заданного элемента XML
`XCData`	Представляет раздел `CDATA` в документе XML. Информация в разделе `CDATA` — это данные документа XML, которые должны быть включены, но не отвечают правилам грамматики XML (например, код сценария)
`XComment`	Представляет комментарий XML
`XDeclaration`	Представляет открывающее объявление документа XML
`XDocument`	Представляет полный документ XML
`XElement`	Представляет заданный элемент внутри документа XML, включая корневой
`XName`	Представляет имя элемента или атрибута XML
`XNamespace`	Представляет пространство имен XML
`XNode`	Представляет абстрактную концепцию узла (элемент, комментарий, тип документа, инструкция обработки или текстовый узел) в дереве XML
`XProcessingInstruction`	Представляет инструкцию обработки XML
`XStreamingElement`	Представляет элементы в дереве XML, которые поддерживают отложенный потоковый вывод

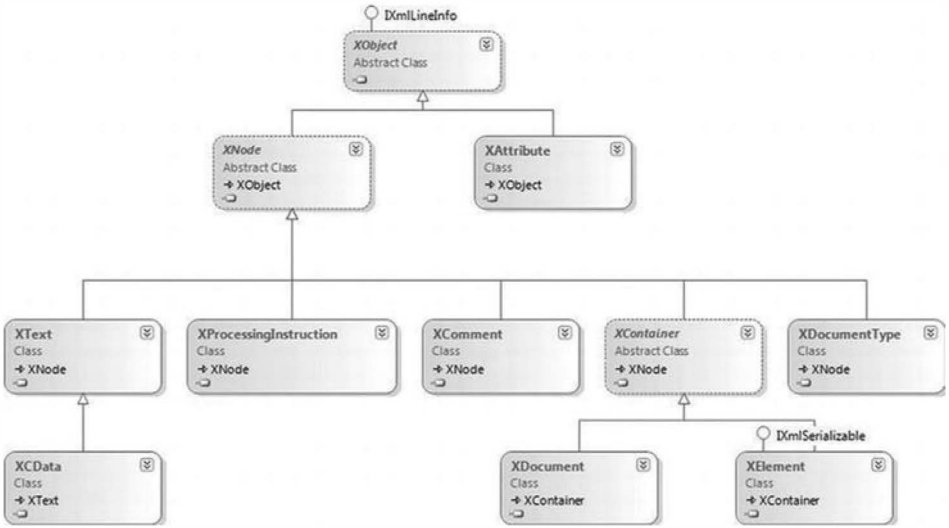


Рис. 24.3. Иерархия основных классов LINQ to XML

Осевые методы LINQ to XML

В дополнение к классам `X*` внутри пространства имен `System.Xml.Linq` определен класс по имени `Extensions`, определяющий набор расширяющих методов, которые обычно расширяют `IEnumerable<T>`, где `T` — потомок `XNode` или `XContainer`. В табл. 24.2 описаны важные расширяющие методы, о которых следует знать (как вы увидите, они удобны при работе с запросами LINQ).

Таблица 24.2. Избранные члены класса LINQ to XML

Член	Описание
`Ancestor<T>()`	Возвращает отфильтрованную коллекцию элементов, которая содержит предков каждого узла в исходной коллекции
`Attributes()`	Возвращает отфильтрованную коллекцию атрибутов каждого элемента в исходной коллекции
`DescendantNodes<T>`	Возвращает коллекцию узлов-потомков каждого документа и элемента в исходной коллекции
`Descendants<T>`	Возвращает отфильтрованную коллекцию элементов, которая содержит элементы-потомки каждого элемента и документа в исходной коллекции
`Elements<T>`	Возвращает коллекцию дочерних элементов каждого элемента и документа в исходной коллекции
`Nodes<T>`	Возвращает коллекцию дочерних узлов каждого документа и элемента в исходной коллекции
`Remove()`	Удаляет каждый атрибут исходной коллекции из родительского элемента
`Remove<T>()`	Удаляет все вхождения заданного узла в исходной коллекции

Как можно понять по именам, эти методы позволяют запрашивать загруженное дерево XML в поисках элементов, атрибутов и их значений. Все вместе такие методы называются *осевыми методами* или просто *осями*. Их можно применять напрямую к частям дерева узлов либо использовать для построения более сложных запросов LINQ.

На заметку! Абстрактный класс `XContainer` поддерживает несколько методов, которые именованы идентично членам класса `Extensions`. Класс `XContainer` является родительским для `XElement` и `XDocument`, поэтому оба класса поддерживают ту же самую общую функциональность.

Примеры применения некоторых осевых методов будут встречаться далее в главе. Пока что вот краткий пример:

```
private static void DeleteNodeFromDoc()
{
  XElement doc =
    new XElement("Inventory",
      new XElement("Car", new XAttribute("ID", "1000"),
        new XElement("PetName", "Jimbo"),
        new XElement("Color", "Red"),
        new XElement("Make", "Ford")
      )
    );
  // Удалить элемент PetName из дерева.
  doc.Descendants("PetName").Remove();
    Console.WriteLine(doc);
}
```

В результате вызова этого метода получается следующее "усеченное" дерево XML:

```
<Inventory>
 <Car ID="1000">
  <Color>Red</Color>
  <Make>Ford</Make>
 </Car>
</Inventory>
```

Странность класса XName (и XNamespace)

Просмотрев сигнатуры осевых методов LINQ to XML (или идентично именованных членов `XContainer`), вы заметите, что они обычно требуют указания того, что выглядит как объект `XName`. Взгляните на сигнатуру метода `Descendants()`, определенного в `XContainer`:

```
public IEnumerable<XElement> Descendants(XName name)
```

Класс `XName` является "странным" в том, что он никогда не будет использоваться в коде напрямую. В действительности, поскольку этот класс не имеет открытого конструктора, объект типа `XName` создавать невозможно:

```
// Ошибка! Объекты XName создавать невозможно!
doc.Descendants(new XName("PetName")).Remove();
```

В формальном определении `XName` вы обнаружите специальную операцию неявного преобразования (специальные операции преобразования были описаны в главе 11), которая отобразит простой тип `System.String` на корректный объект `XName`:

```
// В действительности объект XName создается на заднем плане!
doc.Descendants("PetName").Remove();
```

На заметку! Класс `XNamespace` также поддерживает ту же самую разновидность неявного преобразования строк.

Хорошая новость в том, что при работе с осевыми методами можно применять текстовые значения для представления имен элементов или атрибутов и позволять API-интерфейсу LINQ to XML отображать строковые данные на необходимые типы объектов.

Исходный код. Проект `LinqToXmlFirstLook` доступен в подкаталоге `Chapter_24`.

Работа с `XElement` и `XDocument`

Давайте продолжим исследование LINQ to XML, создав новый проект консольного приложения по имени `ConstructingXmlDocs`. После создания проекта импортируем пространство имен `System.Xml.Linq` в начальный файл кода. Вы уже видели, что класс `XDocument` представляет полный документ XML в программной модели LINQ to XML, т.к. он может использоваться для определения корневого элемента, а также всех содержащихся в нем элементов, инструкций обработки и объявлений XML. Вот еще один пример построения данных XML с применением `XDocument`:

```
static void CreateFullXDocument()
{
  XDocument inventoryDoc =
    new XDocument(
      new XDeclaration("1.0", "utf-8", "yes"),
      new XComment("Current Inventory of cars!"),
      new XProcessingInstruction("xml-stylesheet",
        "href='MyStyles.css' title='Compact' type='text/css'"),
      new XElement("Inventory",
        new XElement("Car", new XAttribute("ID", "1"),
          new XElement("Color", "Green"),
          new XElement("Make", "BMW"),
          new XElement("PetName", "Stan")
        ),
        new XElement("Car", new XAttribute("ID", "2"),
          new XElement("Color", "Pink"),
          new XElement("Make", "Yugo"),
          new XElement("PetName", "Melvin")
        )
      )
    );
  // Сохранить на диске.
  inventoryDoc.Save("SimpleInventory.xml");
}
```

И снова обратите внимание, что конструктор объекта `XDocument` на самом деле представляет собой дерево дополнительных объектов LINQ to XML. Вызываемый здесь конструктор принимает в первом параметре объект `XDeclaration`, за которым следует параметр-массив объектов (вспомните, что параметры-массивы позволяют передавать разделенные запятыми списки аргументов, которые автоматически упаковываются в массив):

```
public XDocument(System.Xml.Linq.XDeclaration declaration,
                 params object[] content)
```

В результате вызова этого метода внутри `Main()` в файл `SimpleInventory.xml` будут записаны приведенные ниже данные:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--Current Inventory of cars!-->
<?xml-stylesheet href='MyStyles.css' title='Compact' type='text/css'?>
<Inventory>
 <Car ID="1">
  <Color>Green</Color>
  <Make>BMW</Make>
  <PetName>Stan</PetName>
 </Car>
 <Car ID="2">
  <Color>Pink</Color>
  <Make>Yugo</Make>
  <PetName>Melvin</PetName>
 </Car>
</Inventory>
```

Как выясняется, стандартное объявление XML для любого `XDocument` предусматривает использование кодировки UTF-8, XML версии 1.0 и автономного документа. Следовательно, код создания объекта `XDeclaration` можно полностью удалить и получить те же самые данные; учитывая, что почти любой документ требует одного и того же объявления, применять `XDeclaration` приходится нечасто.

Если определять инструкции обработки или специальное объявление XML нет необходимости, то можно вообще избежать использования `XDocument` и просто работать с классом `XElement`. Помните, что `XElement` может применяться для представления корневого элемента документа XML и всех подобъектов. Таким образом, вот как можно сгенерировать прокомментированный список складских запасов:

```
static void CreateRootAndChildren()
{
  XElement inventoryDoc =
    new XElement("Inventory",
      new XComment("Current Inventory of cars!"),
        new XElement("Car", new XAttribute("ID", "1"),
          new XElement("Color", "Green"),
          new XElement("Make", "BMW"),
          new XElement("PetName", "Stan")
        ),
        new XElement("Car", new XAttribute("ID", "2"),
          new XElement("Color", "Pink"),
          new XElement("Make", "Yugo"),
          new XElement("PetName", "Melvin")
        )
      )
    );
  // Сохранить на диске.
  inventoryDoc.Save("SimpleInventory.xml");
}
```

Результат будет более или менее идентичным кроме инструкции обработки для гипотетической стилевой таблицы:

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory>
 <!--Current Inventory of cars!-->
 <Car ID="1">
  <Color>Green</Color>
```

```
  <Make>BMW</Make>
  <PetName>Stan</PetName>
 </Car>
 <Car ID="2">
  <Color>Pink</Color>
  <Make>Yugo</Make>
  <PetName>Melvin</PetName>
</Car>
</Inventory>
```

Генерация документов из массивов и контейнеров

До сих пор документы XML строились с использованием жестко закодированных значений для конструктора. Но гораздо чаще требуется генерировать объект `XElement` (или `XDocument`), читая данные из массивов, объектов ADO.NET, файлов данных и т.п. Один из способов отображения данных из памяти на новый объект `XElement` заключается в применении стандартных циклов `for` для перемещения данных в объектную модель LINQ to XML. Хотя это определенно возможно, проще встроить запрос LINQ прямо в код конструирования `XElement`.

Предположим, что имеется анонимный массив анонимных классов (просто чтобы сократить объем кода в примере; подойдет также любой массив, `List<T>` или другой контейнер). Отобразить эти данные на `XElement` можно следующим образом:

```
static void MakeXElementFromArray()
{
  // Создать анонимный массив анонимных типов.
  var people = new[] {
    new { FirstName = "Mandy", Age = 32},
    new { FirstName = "Andrew", Age = 40 },
    new { FirstName = "Dave", Age = 41 },
    new { FirstName = "Sara", Age = 31}
  };
XElement peopleDoc =
  new XElement("People",
    from c in people select new XElement("Person", new XAttribute("Age",
c.Age),
            new XElement("FirstName", c.FirstName))
  );
  Console.WriteLine(peopleDoc);
}
```

Здесь объект `peopleDoc` определяет корневой элемент `<People>` с результатами запроса LINQ. Этот запрос LINQ создает новые объекты `XElement` на основе каждого элемента в массиве `people`. Если встроенный запрос покажется трудным для восприятия, все можно разделить на явные шаги:

```
static void MakeXElementFromArray()
{
  // Создать анонимный массив анонимных типов.
  var people = new[] {
    new { FirstName = "Mandy", Age = 32},
    new { FirstName = "Andrew", Age = 40 },
    new { FirstName = "Dave", Age = 41 },
    new { FirstName = "Sara", Age = 31}
  };
```

```
  var arrayDataAsXElements = from c in people
    select
      new XElement("Person",
        new XAttribute("Age", c.Age),
        new XElement("FirstName", c.FirstName));
  XElement peopleDoc = new XElement("People", arrayDataAsXElements);
  Console.WriteLine(peopleDoc);
}
```

В любом случае вывод будет таким:

```
<People>
 <Person Age="32">
  <FirstName>Mandy</FirstName>
 </Person>
 <Person Age="40">
  <FirstName>Andrew</FirstName>
 </Person>
 <Person Age="41">
  <FirstName>Dave</FirstName>
 </Person>
 <Person Age="31">
  <FirstName>Sara</FirstName>
 </Person>
</People>
```

Загрузка и разбор содержимого XML

Типы `XElement` и `XDocument` поддерживают методы `Load()` и `Parse()`, которые позволяют наполнить объектную модель XML из объектов `string`, содержащих данные XML, либо из внешних файлов XML. Рассмотрим показанный далее метод, иллюстрирующий оба подхода:

```
static void ParseAndLoadExistingXml()
{
  // Построить объект XElement из строки.
  string myElement =
    @"<Car ID ='3'>
       <Color>Yellow</Color>
       <Make>Yugo</Make>
      </Car>";
  XElement newElement = XElement.Parse(myElement);
  Console.WriteLine(newElement);
  Console.WriteLine();
  // Загрузить файл SimpleInventory.xml.
  XDocument myDoc = XDocument.Load("SimpleInventory.xml");
  Console.WriteLine(myDoc);
}
```

Исходный код. Проект `ConstructingXmlDocs` доступен в подкаталоге `Chapter_24`.

Манипулирование документом XML в памяти

Итак, к настоящему моменту вы видели разнообразные способы использования LINQ to XML для создания, сохранения, разбора и загрузки данных XML. Следующий аспект LINQ to XML, который необходимо исследовать — навигация по документу с целью поиска местоположения и изменения специфичных элементов дерева с применением запросов LINQ и осевых методов LINQ to XML.

Для этого мы построим приложение Windows Forms, которое будет отображать данные из документа XML, сохраненного на жестком диске. Графический пользовательский интерфейс позволит пользователю вводить данные для нового узла, который будет добавлен к тому же самому документу XML. Наконец, пользователю будет предоставлено несколько способов для выполнения поиска в документе с помощью набора запросов LINQ.

На заметку! Учитывая, что некоторые запросы LINQ уже строились в главе 12, здесь они повторяться не будут. В разделе "Querying XML Trees" ("Запрашивание деревьев XML") документации .NET Framework 4.6 SDK можно ознакомиться с дополнительными примерами LINQ to XML.

Построение пользовательского интерфейса для приложения LINQ to XML

Создадим проект приложения Windows Forms под названием `LinqToXmlWinApp` и изменим имя первоначального файла `Form1.cs` на `MainForm.cs` (в окне Solution Explorer). Графический пользовательский интерфейс этого приложения довольно прост. В левой части окна находится элемент управления `TextBox` (по имени `txtInventory`), свойство `Multiline` которого установлено в `true`, а свойство `ScrollBars` — в `Both`.

Кроме того, имеется одна группа простых элементов управления `TextBox` (`txtMake`, `txtColor` и `txtPetName`) и элемент `Button` (`btnAddNewItem`), щелчок на котором приводит к добавлению новой записи в документ XML. Наконец, есть еще одна группа элементов управления (`TextBox` по имени `txtMakeToLookUp` и элемент `Button` с именем `btnLookUpColors`), которая позволяет запрашивать из документа XML набор указанных узлов. На рис. 24.4 показана возможная компоновка окна.

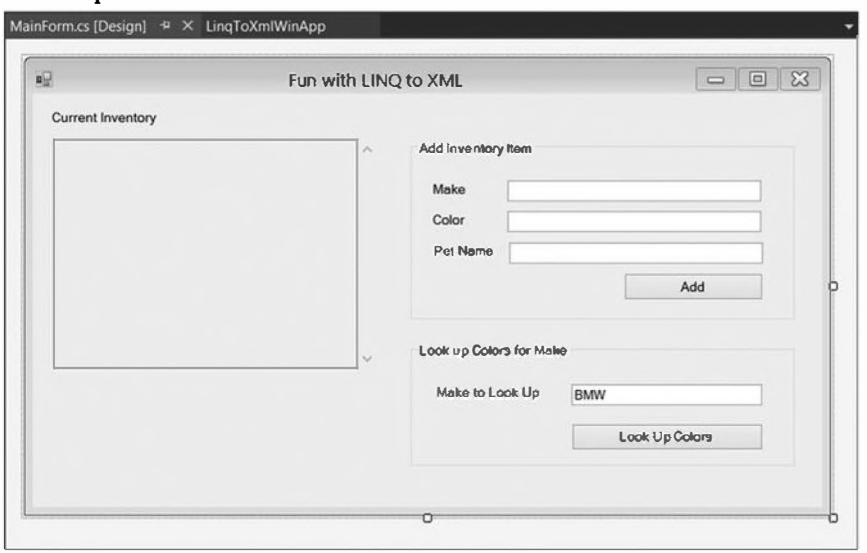


Рис. 24.4. Графический пользовательский интерфейс для приложения LINQ to XML

Потребуется обработать событие `Click` каждой кнопки для генерации методов обработки событий, а также обработать событие `Load` формы. Мы займемся этим чуть позже.

Импортирование файла `Inventory.xml`

В состав загружаемого кода примеров для книги включен файл `Inventory.xml`, в котором имеется набор сущностей внутри корневого элемента `<Inventory>`. Импортируем этот файл в проект, выбрав пункт меню Project⇨Add Existing Item (Проект⇨Добавить существующий элемент). Просмотрев данные, можно заметить, что корневой элемент определяет набор элементов `<Car>`, каждый из которых определен подобно следующему:

```
<Car carID ="0">
 <Make>Ford</Make>
 <Color>Blue</Color>
 <PetName>Chuck</PetName>
</Car>
```

Прежде чем продолжить, этот файл понадобится выбрать в окне Solution Explorer и затем в окне Properties (Свойства) установить свойство Copy to Output Directory (Копировать в выходной каталог) в Copy Always (Копировать всегда). Это обеспечит помещение данных в папку `bin\Debug` при компиляции приложения.

Определение вспомогательного класса LINQ to XML

Чтобы изолировать данные LINQ to XML, добавим в проект новый класс по имени `LinqToXmlObjectModel`. В нем будет определен набор статических методов, инкапсулирующих некоторую логику LINQ to XML. Первым делом определим метод, который возвращает заполненный объект `XDocument` на основе содержимого файла `Inventory.xml` (в новый файл должны быть импортированы пространства имен `System.Xml.Linq` и `System.Windows.Forms`):

```
public static XDocument GetXmlInventory()
{
  try
  {
    XDocument inventoryDoc = XDocument.Load("Inventory.xml");
    return inventoryDoc;
  }
  catch (System.IO.FileNotFoundException ex)
  {
    MessageBox.Show(ex.Message);
    return null;
  }
}
```

Метод `InsertNewElement()`, код которого приведен ниже, получает значения элементов управления `TextBox` из раздела Add Inventory Item (Добавить элемент на склад) и помещает новый узел внутрь элемента `<Inventory>`, используя осевой метод `Descandants()`. После этого документ будет сохранен.

```
public static void InsertNewElement(string make, string color, string
petName)
{
  // Загрузить текущий документ.
  XDocument inventoryDoc = XDocument.Load("Inventory.xml");
```

```
  // Сгенерировать случайное число для идентификатора.
  Random r = new Random();
  // Создать новый объект XElement на основе входных параметров.
  XElement newElement = new XElement("Car", new XAttribute("ID", r.Next(50000)),
    new XElement("Color", color),
    new XElement("Make", make),
    new XElement("PetName", petName));
  // Добавить к объекту XDocument в памяти.
  inventoryDoc.Descendants("Inventory").First().Add(newElement);
  // Сохранить изменения на диске.
  inventoryDoc.Save("Inventory.xml");
}
```

Финальный метод, `LookUpColorsForMake()`, получает данные из последнего элемента управления `TextBox` и с применением запроса LINQ строит строку, которая содержит цвета указанного изготовителя. Взгляните на следующую реализацию:

```
public static void LookUpColorsForMake(string make)
{
  // Загрузить текущий документ.
  XDocument inventoryDoc = XDocument.Load("Inventory.xml");
  // Найти цвета для заданного изготовителя.
  var makeInfo = from car in inventoryDoc.Descendants("Car")
    where (string)car.Element("Make") == make
    select car.Element("Color").Value;
  // Построить строку, представляющую каждый цвет.
  string data = string.Empty;
  foreach (var item in makeInfo.Distinct())
  {
    data += string.Format("- {0}\n", item);
  }
  // Показать цвета.
  MessageBox.Show(data, string.Format("{0} colors:", make));
}
```

Связывание пользовательского интерфейса и вспомогательного класса

К настоящему моменту все, что осталось сделать — это реализовать обработчики событий. Задача сводится к простым вызовам статических вспомогательных методов:

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }
  private void MainForm_Load(object sender, EventArgs e)
  {
    // Отобразить текущий документ XML склада в элементе управления TextBox.
    txtInventory.Text = LinqToXmlObjectModel.GetXmlInventory().ToString();
  }
  private void btnAddNewItem_Click(object sender, EventArgs e)
  {
    // Добавить к документу новый элемент.
    LinqToXmlObjectModel.InsertNewElement(txtMake.Text,
                                          txtColor.Text, txtPetName.Text);
```

```
    // Отобразить текущий документ XML для склада в элементе управления TextBox.
    txtInventory.Text = LinqToXmlObjectModel.GetXmlInventory().ToString();
  }

  private void btnLookUpColors_Click(object sender, EventArgs e)
  {
    LinqToXmlObjectModel.LookUpColorsForMake(txtMakeToLookUp.Text);
  }
}
```

На рис. 24.5 показан результат добавления новой записи об автомобиле и поиска всех записей для BMW.

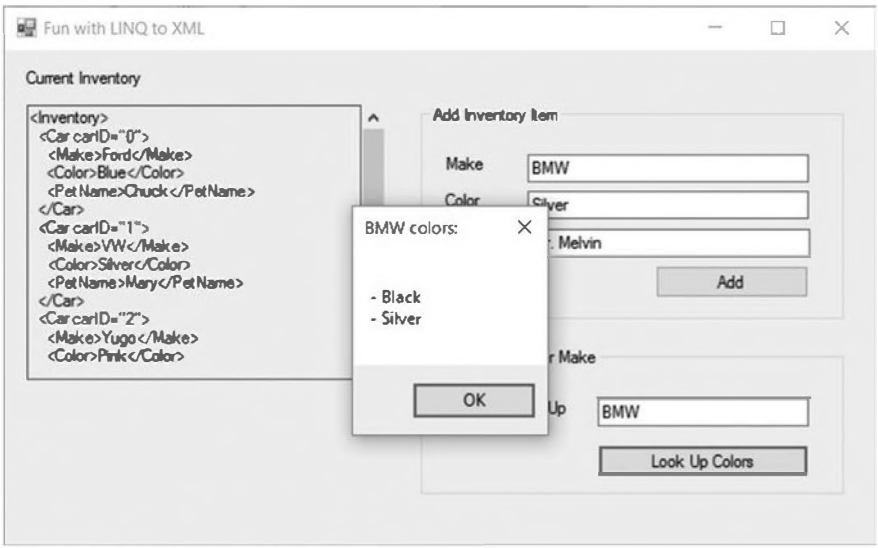


Рис. 24.5. Готовое приложение LINQ to XML

На этом завершается начальное знакомство с LINQ to XML, равно как и исследование LINQ. Впервые вы столкнулись с технологией LINQ в главе 12, где изучали LINQ to Objects. В главе 19 были представлены различные примеры использования PLINQ, а в главе 23 рассматривалось применение LINQ к сущностным объектам ADO.NET. Теперь вы готовы, да и должны двигаться дальше. В Microsoft совершенно ясно дают понять, что по мере расширения платформы .NET технология LINQ продолжит развиваться.

Исходный код. Проект `LinqToXmlWinApp` доступен в подкаталоге `Chapter_24`.

Резюме

В этой главе рассматривалась роль LINQ to XML. Вы увидели, что этот API-интерфейс является альтернативой первоначальной библиотеке для манипуляций данными XML, `System.Xml.dll`, которая поставлялась в составе платформы .NET. С помощью `System.Xml.Linq.dll` появляется возможность генерации новых документов XML с использованием подхода "сверху вниз", при котором структура кода очень напоминает финальные данные XML. В таком свете LINQ to XML — лучшая модель DOM. Вдобавок вы узнали, каким образом строить объекты `XDocument` и `XElement` разнообразными способами (разбор, загрузка из файла, отображение объектов в памяти), а также выполнять навигацию и манипулировать данными с применением запросов LINQ.

ГЛАВА 25

Введение в Windows Communication Foundation

Инфраструктура Windows Communication Foundation (WCF) представляет собой API-интерфейс, спроектированный специально для построения распределенных систем. В отличие от других распределенных API-интерфейсов, которые вы могли использовать в прошлом (например, DCOM, .NET Remoting, веб-службы XML, очереди сообщений), WCF предлагает единую, унифицированную и расширяемую объектную модель для программирования, которую можно применять для взаимодействия с множеством ранее разрозненных распределенных технологий.

Глава начинается с объяснения потребности в инфраструктуре WCF и исследования задач, для решения которых она предназначена, посредством краткого обзора предшествующих API-интерфейсов распределенных вычислений. После рассмотрения служб, предлагаемых WCF, внимание будет переключено на изучение основных сборок, пространств имен и типов, которые представляют эту программную модель. В главе будет построено несколько служб, хостов и клиентов WCF с использованием разнообразных инструментов разработки WCF.

На заметку! В этой главе будет создаваться код, который требует запуска Visual Studio с административными привилегиями (к тому же вы и сами должны иметь административные привилегии). Чтобы запустить Visual Studio с правами администратора, необходимо щелкнуть правой кнопкой мыши на значке Visual Studio и выбрать в контекстном меню пункт Запуск от имени администратора.

Собрание API-интерфейсов распределенных вычислений

Операционная система Windows всегда предоставляла множество API-интерфейсов для построения распределенных систем. Хотя и верно то, что под *распределенной системой* большинство понимает систему, включающую минимум два сетевых компьютера, в широком смысле этот термин может относиться к двум исполняемым сборкам, которые нуждаются в обмене данными, даже если они запущены на одной физической машине. При таком определении выбор распределенного API-интерфейса для решения текущей задачи программирования обычно предусматривает ответ на следующий основополагающий вопрос.

> *Будет ли система использоваться исключительно внутренне или же доступ к функциональности приложения потребуется и внешним пользователям?*

В случае построения распределенной системы для внутреннего применения есть больше возможностей гарантировать, что на каждом подключенном компьютере установлена та же самая операционная система и используется та же самая инфраструктура программирования (например, платформа .NET, COM или Java). Внутренние системы также позволяют задействовать существующую систему безопасности для целей аутентификации, авторизации и т.д. В такой ситуации можно выбрать конкретный распределенный API-интерфейс, который с точки зрения производительности согласуется с имеющейся операционной системой и инфраструктурой программирования.

В противоположность этому при построении системы, которая должна быть доступна внешним потребителям, возникает целый ряд других проблем, подлежащих решению. Во-первых, скорее всего вам *не* удастся диктовать внешним пользователям то, какую они могут применять операционную систему (системы) и инфраструктуру (инфраструктуры) программирования, а также каким образом они должны конфигурировать свои настройки безопасности.

Во-вторых, если вы работаете в крупной компании или в университетской среде, где используются многочисленные операционные системы и технологии программирования, то даже внутреннее приложение неожиданно столкнется с теми же проблемами, что и приложение, ориентированное на внешний мир. В любом из этих случаев вам необходимо ограничиться наиболее гибким распределенным API-интерфейсом, чтобы обеспечить максимальную доступность разрабатываемого приложения.

В зависимости от ответа на приведенный выше ключевой вопрос распределенных вычислений далее потребуется выбрать конкретный API-интерфейс (либо их набор). В последующих разделах представлен краткий обзор ряда основных распределенных API-интерфейсов, которые исторически применялись разработчиками программного обеспечения для Windows. После этого краткого экскурса вы легко сможете оценить полезность инфраструктуры Windows Communication Foundation.

На заметку! Чтобы исключить возможные недоразумения, мы должны обратить внимание на то, что инфраструктура WCF (и охватываемые ею технологии) не имеет ничего общего с построением веб-сайтов, основанных на HTML. Наряду с тем, что веб-приложения могут считаться распределенными, поскольку обычно в обмен вовлечены две машины, инфраструктура WCF ориентирована на установление подключений между машинами с целью совместного использования функциональности удаленных компонентов, а не для отображения HTML-разметки в веб-браузере. Построение веб-сайтов с помощью платформы .NET будет рассматриваться, начиная с главы 32.

Роль DCOM

До выпуска платформы .NET основным API-интерфейсом удаленных вычислений в Microsoft была распределенная модель компонентных объектов (Distributed Component Object Model — DCOM). Посредством DCOM можно было строить распределенные системы с применением объектов COM, системного реестра и определенной доли старания. Одно из преимуществ модели DCOM заключалось в том, что она обеспечивала *прозрачность расположения* компонентов. Говоря просто, это позволяло разрабатывать клиентское программное обеспечение так, что физическое местоположение удаленных объектов не было жестко закодировано в приложении. Независимо от того, располагался удаленный объект на той же самой или на другой сетевой машине, кодовая база могла оставаться нейтральной, потому что действительное местоположение записывалось внешне в системный реестр.

Хотя модель DCOM имела определенный успех, для всех практических целей она была API-интерфейсом, ориентированным на Windows. Сама по себе модель DCOM не предлагала архитектуру для построения комплексных решений, вовлекающих множество операционных систем (скажем, Windows, Unix, Mac), и не способствовала разделению данных между несходными платформами (например, COM, Java или CORBA).

На заметку! Предпринимались попытки переноса DCOM в разнообразные среды Unix/Linux, но результаты не были блестящими и в итоге завели в технологический тупик.

В общем и целом модель DCOM лучше всего подходила для разработки внутренних приложений, т.к. открытие доступа к объектам COM извне компании влекла за собой дополнительные сложности (брандмауэры и т.д.). С выходом платформы .NET модель DCOM быстро стала устаревшей, и если вам не приходится сопровождать унаследованные системы DCOM, то можете считать эту технологию не рекомендуемой.

Роль служб COM+/Enterprise Services

Без посторонней помощи модель DCOM делала немногим более чем определение способа установления коммуникационного канала между двумя частями программного обеспечения, основанного на COM. Чтобы восполнить недостающие компоненты, требующиеся при построении многофункциональных распределенных вычислительных решений, компания Microsoft со временем выпустила сервер транзакций (Microsoft Transaction Server — MTS), который позже был переименован в COM+.

Несмотря на название, COM+ использовали не только программисты COM; эта технология полностью доступна также разработчикам приложений для .NET. Со времени выхода первой версии платформы .NET библиотеки базовых классов предоставляют пространство имен `System.EnterpriseServices`. Оно позволяет программистам .NET строить управляемые библиотеки, которые могут быть установлены в исполняющей среде COM+ для доступа к тому же набору служб, что и традиционный COM-сервер, поддерживающий COM+. В любом случае после установки в исполняющей среде COM+ поддерживающая COM+ библиотека называется *служебным компонентом*.

Технология COM+ предлагает несколько средств, которые служебные компоненты могут задействовать: управление транзакциями, управление временем жизни объектов, службы поддержки пулов, систему безопасности на основе ролей, модель слабо связанных событий и т.д. В свое время это было главным преимуществом, поскольку большинство распределенных систем требовали одинакового набора служб. Вместо того чтобы заставлять разработчиков кодировать их вручную, технология COM+ предложила готовое решение.

Одним из привлекательных аспектов COM+ являлся тот факт, что все эти настройки можно было конфигурировать в декларативной манере с применением административных инструментов. Таким образом, если вы хотели обеспечить мониторинг объекта в транзакционном контексте или отнести его к определенной роли безопасности, то просто должны были отметить подходящие флажки.

Несмотря на то что службы COM+/Enterprise Services по-прежнему используются, данная технология предлагает решение только для Windows, которое лучше всего подходит для разработки внутренних приложений либо в качестве серверной службы, опосредованно управляемой более совершенными пользовательскими интерфейсами (например, это может быть публично доступный веб-сайт, обращающийся к служебным компонентам (объектам COM+) в фоновом режиме).

На заметку! Инфраструктура WCF не предоставляет какого-либо способа построения служебных компонентов. Однако службы WCF имеют возможность взаимодействовать с существующими объектами COM+. Если необходимо строить служебные компоненты на языке C#, то придется напрямую работать с пространством имен `System.EnterpriseServices`. Подробные сведения ищите в документации .NET Framework 4.6 SDK.

Роль MSMQ

API-интерфейс MSMQ (Microsoft Message Queuing — очередь сообщений Microsoft) позволяет разработчикам создавать распределенные системы, которым необходимо обеспечивать надежную доставку сообщений по сети. Как хорошо известно разработчикам, в любой распределенной системе существует риск отказа сетевого сервера, отключения базы данных или потери подключений по необъяснимым причинам. Более того, многие приложения должны быть сконструированы так, чтобы они хранили данные сообщений для доставки в более позднее время (процесс известен как *очередизация данных*).

Изначально в Microsoft представили MSMQ как набор низкоуровневых API-интерфейсов, основанных на C, и объектов COM. После выпуска платформы .NET программисты C# могли применять пространство имен `System.Messaging` для привязки к MSMQ и построения программного обеспечения, которое взаимодействует с периодически подключающимися приложениями в зависимой манере.

К слову, уровень COM+, включающий функциональность MSMQ в исполняющую среду (в упрощенном формате), использует технологию, которая называется *Queued Components* (Очередизованные компоненты), или QC. Такой способ взаимодействия с MSMQ был упакован в пространство имен `System.EnterpriseServices`, упоминаемое в предыдущем разделе.

Независимо от того, какая программная модель применяется для взаимодействия с исполняющей средой MSMQ, в итоге гарантируется, что приложения смогут надежно и своевременно доставлять сообщения. Подобно COM+ технология MSMQ все же является частью архитектуры построения распределенного программного обеспечения для операционной системы Windows.

Роль .NET Remoting

Как было указано ранее, после выхода платформы .NET технология DCOM быстро стала устаревшим распределенным API-интерфейсом. Библиотеки базовых классов, поставляемые с уровнем .NET Remoting, представлены пространством имен `System.Runtime.Remoting`. Этот (теперь устаревший) API-интерфейс позволяет множеству компьютеров распространять объекты при условии, что все эти компьютеры выполняют приложения на платформе .NET.

API-интерфейсы .NET Remoting предложили несколько полезных средств. Наиболее важным было использование конфигурационных файлов на основе XML для декларативного определения внутренних связующих механизмов, которые применяются клиентским и серверным программным обеспечением. За счет использования файлов `*.config` было легко радикально изменять функциональность распределенной системы, просто модифицируя содержимое конфигурационных файлов и перезапуская приложение.

Вдобавок, учитывая, что этот API-интерфейс способны применять только приложения .NET, можно получить выигрыш в производительности, т.к. данные допускается кодировать в компактном двоичном формате, а при определении параметров и возвращаемых значений разрешено использовать общую систему типов (Common Type System — CTS). Хотя .NET Remoting реально применять для построения распределенных систем, охватывающих множество операционных систем (используя платформу Mono,

кратко описанную в главе 1), взаимодействие с другими программными архитектурами (такими как Java) по-прежнему не было возможным.

Роль веб-служб XML

Каждый из предшествующих распределенных API-интерфейсов обеспечивал лишь небольшую (если вообще какую-либо) поддержку доступа внешних вызывающих компонентов к предоставляемой функциональности в *независимой манере*. Когда нужно открыть доступ к службам удаленных объектов *любой* операционной системе и *любой* программной модели, самый простой способ предлагают веб-службы XML.

В отличие от традиционного браузерного веб-приложения веб-служба предоставляет способ открытия доступа к функциональности удаленных компонентов с применением стандартных веб-протоколов. С момента появления начального выпуска .NET программистам была предложена превосходная поддержка построения и потребления веб-служб XML через пространство имен `System.Web.Services`. Во многих случаях построение полнофункциональной веб-службы не сложнее применения атрибута `[WebMethod]` к каждому открытому методу, к которому желательно предоставить доступ. Более того, IDE-среда Visual Studio позволяет подключаться к удаленной веб-службе с помощью максимум пары щелчков на кнопках.

Веб-службы позволяют разработчикам строить сборки .NET, содержащие типы, которые могут быть доступны с использованием простого протокола HTTP. Вдобавок веб-служба кодирует свои данные в виде простой разметки XML. Поскольку веб-службы основаны на открытых отраслевых стандартах (например, HTTP, XML и SOAP), а не на патентованных системах типов и сетевых форматах (как в случае DCOM или .NET Remoting), они допускают высокую степень взаимодействия и возможности обмена данными. Независимая природа веб-служб XML иллюстрируется на рис. 25.1.

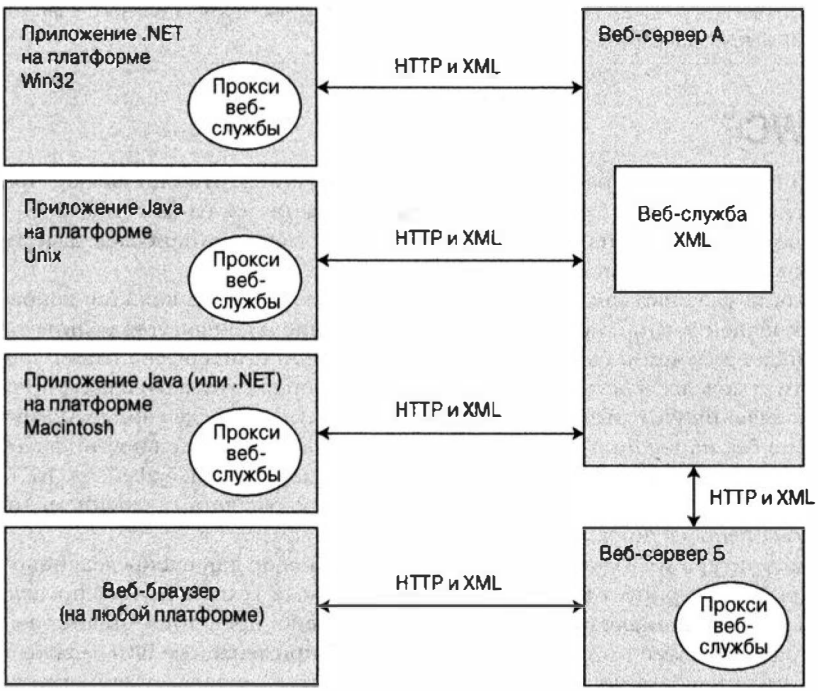


Рис. 25.1. Веб-службы XML обеспечивают высокую степень взаимодействия

Разумеется, безупречных распределенных API-интерфейсов не существует. Один из потенциальных недостатков веб-служб заключается в том, что они могут влечь за собой некоторые проблемы с производительностью (учитывая применение HTTP и представление данных XML). Другой недостаток связан с тем, что веб-службы могут оказаться не идеальным решением для внутренних приложений, где беспрепятственно можно использовать протоколы на основе TCP и двоичное форматирование данных.

Стандарты веб-служб

Еще одна проблема связана с тем, что реализации веб-служб, с которыми мы сталкивались ранее, созданные крупными отраслевыми игроками (скажем, Microsoft, IBM и Sun Microsystems), были не на 100% совместимыми с другими реализациями веб-служб. Вполне очевидно, что это было проблемой, т.к. основной целью веб-служб является достижение высокой степени взаимодействия между платформами и операционными системами!

Чтобы гарантировать возможность взаимодействия веб-служб, группы World Wide Web Consortium (W3C; `www.w3.org`) и Web Services Interoperability Organization (WS-I; `www.ws-i.org`) приступили к разработке спецификаций, определяющих то, как поставщики программного обеспечения (вроде IBM, Microsoft или Sun Microsystems) должны строить библиотеки для веб-служб с поддержкой совместимости.

Все вместе эти спецификации получили общее имя WS-* и раскрыли такие вопросы, как безопасность, вложения, описание веб-служб (с применением языка описания веб-служб (Web Service Description Language — WSDL)), политики, форматы SOAP и массу других важных деталей. Вы увидите, что инфраструктура WCF поддерживает многие спецификации WS-*. Как правило, ваши службы WCF будут делать выбор среди разнообразных спецификаций WS-* на основе используемых *привязок*.

На заметку! В дополнение к распределенным API-интерфейсам, кратко исследованным ранее, разработчики могут также применять различные протоколы межпроцессного взаимодействия, такие как именованные каналы и сокеты.

Роль WCF

Обширный комплект распределенных технологий затрудняет выбор правильного инструмента для работы. Ситуация еще более усложняется из-за того факта, что функциональность некоторых из этих технологий перекрывается (наиболее заметно в областях транзакций и безопасности).

Даже когда разработчик .NET выбрал технологию, которая кажется корректной для решения текущей задачи, построение, сопровождение и конфигурирование такого приложения будет в лучшем случае сложным. Каждый API-интерфейс имеет собственную программную модель, собственный уникальный набор инструментов конфигурирования и т.д. До появления WCF это означало, что подключать распределенные API-интерфейсы было трудно без написания существенного объема кода специальной инфраструктуры. Например, если вы строите систему с использованием API-интерфейсов .NET Remoting и позже решаете, что веб-службы XML будут более подходящим решением, то придется полностью перепроектировать имеющуюся кодовую базу.

Инфраструктура WCF — это инструментальный набор для распределенных вычислений, который интегрирует все эти ранее независимые технологии распределенной обработки в один рационализированный API-интерфейс, представленный главным образом пространством имен `System.ServiceModel`. С применением WCF можно открывать доступ к этим службам для вызывающих компонентов, используя широкое разнообразие приемов. Например, при построении внутреннего приложения, где все подключен-

ные машины основаны на Windows, можно использовать разнообразные протоколы TCP для достижения максимально возможной производительности. Открыть доступ к той же службе можно также с помощью протоколов HTTP и SOAP, чтобы позволить внешним вызывающим компонентам задействовать ее функциональность независимо от языка программирования или операционной системы.

Так как инфраструктура WCF позволяет выбрать корректный протокол для выполнения работы (с применением общей программной модели), вы обнаружите, что подключать лежащий в основе связующий код распределенного приложения становится действительно легко. В большинстве случаев это можно делать без повторной компиляции или развертывания программного обеспечения клиента/службы, потому что утомительные детали часто переносятся в конфигурационные файлы приложения.

Обзор функциональных возможностей WCF

Возможность взаимодействия и интеграция разрозненных API-интерфейсов — это только два важных аспекта WCF. Инфраструктура WCF также предлагает развитую программную архитектуру, которая дополняет поддерживаемые ею технологии удаленной разработки. Ниже приведен список главных средств WCF.

- Поддержка строго типизированных, а также нетипизированных сообщений. Такой подход позволяет приложениям .NET эффективно совместно использовать типы, в то время как программное обеспечение, созданное с применением других платформ (таких как Java), может потреблять потоки слабо типизированного XML.
- Поддержка нескольких привязок (например, низкоуровневый HTTP, TCP, MSMQ, WebSockets, именованные каналы и т.д.) позволяет выбирать наиболее подходящий механизм для транспортировки данных сообщений.
- Поддержка последних спецификаций веб-служб (WS-*).
- Полностью интегрированная модель безопасности, охватывающая как собственные протоколы безопасности Windows/.NET, так и многочисленные нейтральные технологии защиты, построенные на основе стандартов веб-служб.
- Поддержка технологий сеансового управления состоянием, а также поддержка однонаправленных сообщений без состояния.

Как бы впечатляюще не выглядел этот список, он лишь поверхностно касается функциональности, предоставляемой WCF. Технология WCF также предлагает средства трассировки и протоколирования, счетчики производительности, модель публикации и подписки на события, поддержку транзакций и многое другое.

Обзор архитектуры, ориентированной на службы

Еще одно преимущество инфраструктуры WCF связано с тем, что она базируется на принципах проектирования, которые установлены *архитектурой, ориентированной на службы* (service-oriented architecture — SOA). Конечно, SOA является модным словечком в отрасли, и подобно многим модным словечкам SOA может определяться многочисленными путями. Попросту говоря, SOA — это способ проектирования распределенной системы, где несколько автономных *служб* работают совместно, передавая *сообщения* через границы (либо сетевых машин, либо двух процессов на одной машине) с использованием четко определенных *интерфейсов*.

В мире WCF такие четко определенные интерфейсы обычно создаются с применением интерфейсных типов CLR (см. главу 9). Однако в более общем смысле интерфейс службы просто описывает набор членов, которые могут быть вызваны внешними компонентами.

Команда разработчиков WCF следовала четырем принципам проектирования SOA. Несмотря на то что эти принципы соблюдаются автоматически, просто за счет пост-

роения приложения WCF, усвоение представленных четырех важнейших правил проектирования SOA может помочь в лучшем понимании WCF. В последующих разделах приведен краткий обзор данных принципов.

Принцип 1: границы установлены явно

Этот принцип подчеркивает тот факт, что функциональность службы WCF выражается с использованием четко определенных интерфейсов (например, описания каждого члена, его параметров и возвращаемых значений). Единственный способ, которым внешний компонент может взаимодействовать со службой WCF — через интерфейс, при этом оставаясь в полном неведении относительно деталей ее внутренней реализации.

Принцип 2: службы являются автономными

Термин *автономные сущности* относится тому факту, что заданная служба WCF является (насколько возможно) отдельным "островом". Автономная служба должна быть независимой в отношении версии, развертывания и установки. Чтобы содействовать в поддержке этого принципа, можно возвратиться к ключевому аспекту программирования на основе интерфейсов. После того как интерфейс передан в производственную среду, он никогда не должен изменяться (или возникнет риск нарушения работы существующих клиентов). Когда требуется расширить функциональность службы WCF, необходимо просто написать новые интерфейсы, моделирующие желаемую функциональность.

Принцип 3: службы взаимодействуют через контракт, а не реализацию

Третий принцип представляет собой еще один побочный продукт программирования на основе интерфейсов. Детали реализации службы WCF (скажем, на каком языке она написана, как выполняет свою работу и т.п.) не касаются вызывающего внешнего компонента. Клиенты WCF взаимодействуют со службами исключительно через их открытые интерфейсы.

Принцип 4: совместимость служб основана на политике

Поскольку интерфейсы CLR предоставляют строго типизированные контракты всем клиентам WCF (и также могут применяться для генерации связанного документа WSDL на основе выбранной привязки), важно уяснить, что интерфейсы и WSDL сами по себе недостаточно выразительны, чтобы детализировать аспекты того, что способна делать служба. С учетом этого SOA позволяет определять *политики*, которые дополнительно проясняют семантику службы (например, ожидаемые требования безопасности, используемые для взаимодействия со службой). С помощью этих политик можно по существу отделять низкоуровневое синтаксическое описание службы (открытые интерфейсы) от семантических деталей ее работы и способа обращения к ней.

WCF: итоги

Предшествующий краткий исторический экскурс прояснил, почему WCF считается предпочтительным подходом для построения распределенных приложений. Инфраструктура WCF является рекомендуемым API-интерфейсом, когда нужно разрабатывать внутреннее приложение с применением протоколов TCP, перемещать данные между программами на одной машине с использованием именованных каналов или в целом открывать доступ к данным внешнему миру с применением протоколов, основанных на HTTP.

Это вовсе не говорит о невозможности использования в новых разработках исходных пространств имен, связанных с распределенными вычислениями (т.е. `System.Runtime.Remoting`, `System.Messaging`, `System.EnterpriseServices` и `System.Web.Services`).

В некоторых ситуациях (например, когда необходимо строить объекты COM+) применять их придется. В любом случае, если вы использовали эти API-интерфейсы в прошлых проектах, то изучение WCF не представит особого труда. Подобно предшествующим технологиям в WCF интенсивно применяются конфигурационные XML-файлы, атрибуты .NET и утилиты генерации прокси.

Вооружившись всеми этими знаниями, теперь можно сосредоточить внимание на процессе построения приложений WCF. Вы должны понимать, что полное раскрытие инфраструктуры WCF потребовало бы целой книги, т.к. описание каждой поддерживающей службы (скажем, MSMQ, COM+, P2P и именованных каналов) могло бы занять отдельную главу. Здесь вы изучите общий процесс построения программ WCF, использующих протоколы на основе TCP и HTTP (т.е. веб-службу). Это должно подготовить почву для дальнейшего углубления знаний в данной области.

Исследование основных сборок WCF

Как и можно было ожидать, программная архитектура WCF представлена набором сборок .NET, установленных в GAC. В табл. 25.1 описаны основные сборки WCF, которые придется применять почти в любом приложении WCF.

Таблица 25.1. Основные сборки WCF

Сборка	Описание
`System.Runtime.Serialization.dll`	В этой основной сборке определены пространства имен и типы, используемые для сериализации и десериализации объектов в инфраструктуре WCF
`System.ServiceModel.dll`	Эта основная сборка содержит типы, применяемые для построения приложений WCF любого рода

В двух сборках, перечисленных в табл. 25.1, определено много пространств имен и типов. Детальные сведения о них доступны в документации .NET Framework 4.6 SDK, а в табл. 25.2 описаны некоторые важные пространства имен.

Таблица 25.2. Основные пространства имен WCF

Пространство имен	Описание
`System.Runtime.Serialization`	Это пространство имен определяет многочисленные типы, которые используются для управления сериализацией и десериализацией данных в WCF
`System.ServiceModel`	Это первичное пространство имен WCF определяет типы привязки и хостинга, а также базовые типы безопасности и транзакций
`System.ServiceModel.Configuration`	Это пространство имен определяет многочисленные типы, обеспечивающие программный доступ к конфигурационным файлам WCF
`System.ServiceModel.Description`	Это пространство имен определяет типы, которые предоставляют объектную модель для адресов, привязок и контрактов, заданных внутри конфигурационных файлов WCF
`System.ServiceModel.MsmqIntegration`	Это пространство имен определяет типы для интеграции со службой MSMQ
`System.ServiceModel.Security`	Это пространство имен определяет многочисленные типы для управления аспектами уровней безопасности WCF

Шаблоны проектов WCF в Visual Studio

Как будет более подробно объясняться позже в главе, приложение WCF обычно представлено тремя взаимосвязанными сборками; одной из них является библиотека `*.dll`, содержащая типы, с которыми могут взаимодействовать внешние вызывающие компоненты (другими словами, сама служба WCF). Когда вы хотите построить службу WCF, совершенно допустимо выбирать в качестве отправной точки стандартный шаблон проекта Class Library (Библиотека классов), как демонстрировалось в главе 14, и вручную добавлять ссылки на сборки WCF.

В качестве альтернативы создать новую службу WCF можно, выбрав в Visual Studio шаблон проекта WCF Service Library (Библиотека служб WCF), как показано на рис. 25.2. Этот тип проекта автоматически устанавливает ссылки на обязательные сборки WCF, а также генерирует приличный объем начального кода, который довольно часто просто удаляется.

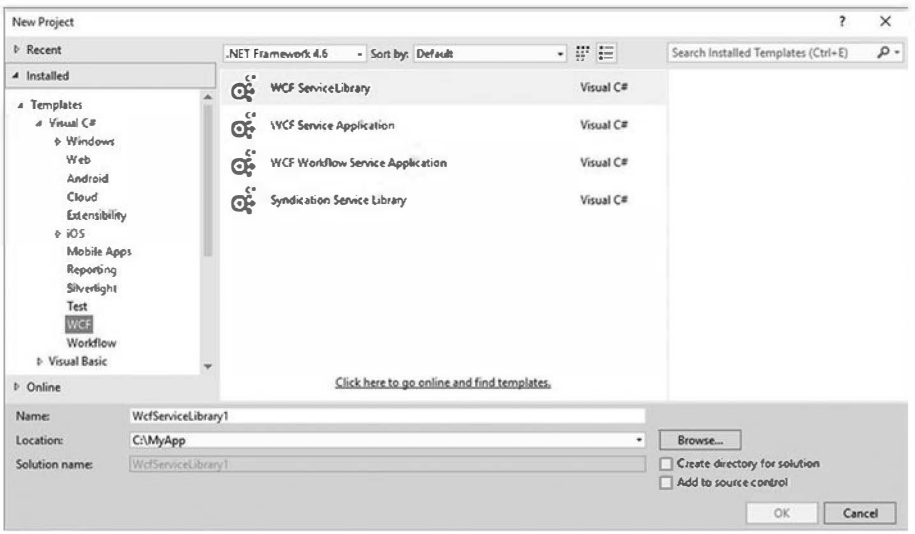


Рис. 25.2. Шаблон проекта WCF Service Library в Visual Studio

Одно из преимуществ выбора шаблона проекта WCF Service Library связано с тем, что он также снабжает вас файлом `App.config`, что поначалу может показаться странным, т.к. строится .NET-сборка `*.dll`, а не `*.exe`. Тем не менее, этот файл полезен тем, что при отладке или запуске проекта IDE-среда Visual Studio автоматически запустит приложение WCF Test Client (Тестовый клиент WCF). Программа `WcfTestClient.exe` ожидает найти настройки в файле `App.config`, поэтому может применяться для тестирования службы. Более подробно эта программа рассматривается далее в главе.

На заметку! Файл `App.config` из проекта WCF Service Library также полезен тем, что демонстрирует начальные настройки, используемые для конфигурирования хост-приложения WCF. В действительности большую часть этого кода можно копировать и вставлять в конфигурационный файл для производственных служб.

В дополнение к базовому шаблону WCF Service Library категория проектов WCF внутри диалогового окна New Project (Новый проект) содержит проект библиотеки WCF, который интегрирует функциональность Windows Workflow Foundation (WF) в службу WCF, а также шаблон для построения библиотеки RSS (см. рис. 25.2).

Шаблон проекта WCF Service для веб-сайта

В Visual Studio доступен еще один шаблон проекта WCF Service (Служба WCF), находящийся в диалоговом окне New Web Site (Новый веб-сайт), которое открывается через пункт меню File⇨New⇨Web Site (Файл⇨Создать⇨Веб-сайт) и показано на рис. 25.3.

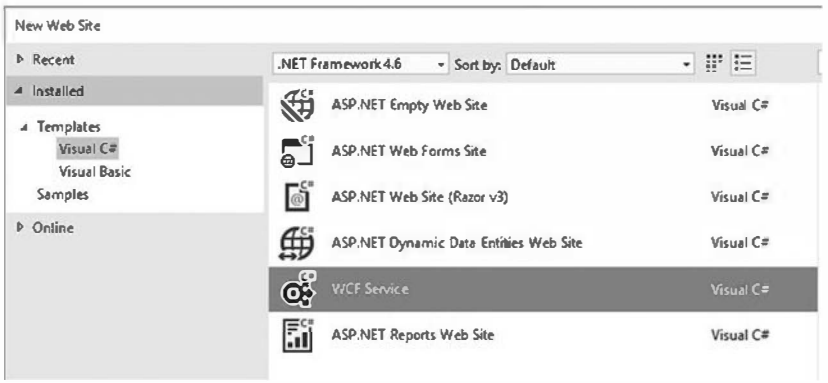


Рис. 25.3. Шаблон проекта WCF Service в Visual Studio

Шаблон проекта WCF Service удобен, когда заранее известно, что служба WCF будет применять протоколы, основанные на HTTP, а не, к примеру, протокол TCP или именованные каналы. Этот шаблон может автоматически создать новый виртуальный каталог Internet Information Services (IIS) для хранения программных файлов WCF, сформировать подходящий файл `Web.config` для открытия доступа к службе по HTTP и сгенерировать необходимый файл `*.svc` (вы узнаете о файлах `*.svc` позже в главе). Таким образом, веб-ориентированный проект WCF Service просто экономит время, поскольку IDE-среда автоматически настраивает всю требуемую инфраструктуру IIS.

По контрасту с этим, если вы строите новую службу WCF, используя шаблон WCF Service Library, то имеете возможность размещения службы разнообразными способами (в том числе в специальном хосте, Windows-службе или вручную созданном виртуальном каталоге IIS). Такой вариант больше подходит, когда для службы WCF необходимо построить специальный хост, который может работать с любым количеством привязок WCF.

Базовая структура приложения WCF

При построении распределенной системы WCF обычно создаются следующие три взаимосвязанных сборки.

- *Сборка службы WCF.* Эта библиотека `*.dll` содержит классы и интерфейсы, представляющие общую функциональность, доступ к которой нужно открыть внешним вызывающим компонентам.
- *Хост службы WCF.* Этот программный модуль является сущностью, которая размещает в себе сборку службы WCF.
- *Клиент WCF.* Это приложение, которое получает доступ к функциональности службы через промежуточный прокси.

Как упоминалось ранее, сборка службы WCF представляет собой библиотеку классов .NET, которая содержит несколько контрактов WCF и их реализации. Ключевое отличие состоит в том, что интерфейсные контракты декорированы разнообразными атрибутами, которые управляют представлением типов данных, взаимодействием исполняющей среды WCF с открытыми типами и т.д.

Вторая сборка, хост службы WCF, может быть буквально любым исполняемым файлом .NET. Далее в этой главе вы увидите, что инфраструктура WCF настроена так, что доступ к службам можно легко открывать из приложения любого типа (например, приложения Windows Forms, Windows-службы, приложения WPF). При построении специального хоста применяется тип `ServiceHost` и возможно связанный с ним файл `*.config`. Файл `*.config` содержит детали, касающиеся механизма серверной стороны, который желательно использовать. Однако если в качестве хоста для службы WCF применяется IIS, то нет необходимости в программном построении специального хоста, т.к. IIS будет использовать "за кулисами" тип `ServiceHost`.

На заметку! Размещение службы WCF также возможно с применением службы активации Windows (Windows Activation Service — WAS); за подробными сведениями обращайтесь в документацию .NET Framework 4.6 SDK.

Финальная сборка представляет клиента, который выполняет обращения к службе WCF. Вполне ожидаемо таким клиентом может быть приложение .NET любого типа. Подобно хосту клиентское приложение обычно использует файл `*.config` клиентской стороны, в котором определен механизм на стороне клиента. Вы должны также знать, что если служба WCF построена с применением привязок, основанных на HTTP, то клиентское приложение может быть реализовано на другой платформе (скажем, Java).

На рис. 25.4 показаны высокоуровневые отношения между этими тремя взаимосвязанными сборками WCF. Для представления требуемого связующего механизма "за кулисами" используются многочисленные низкоуровневые детали (например, фабрики, каналы и слушатели). Такие низкоуровневые детали чаще всего скрыты от глаз; тем не менее, при необходимости они могут быть расширены или настроены. В большинстве случаев хорошо подходит стандартный механизм.

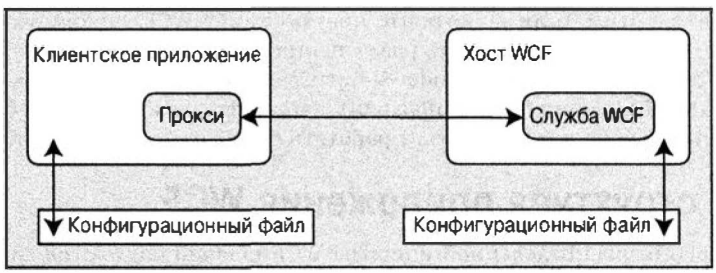


Рис. 25.4. Высокоуровневое представление типичного приложения WCF

Также полезно отметить, что применять файл `*.config` серверной или клиентской стороны формально необязательно. При желании можно жестко закодировать хост (а также клиент), указав необходимый связующий механизм (т.е. конечные точки, привязку и адреса). Очевидная проблема такого подхода заключается в том, что если нужно изменить детали связующего механизма, то придется вносить изменения в код, перекомпилировать и заново развертывать множество сборок. Использование файла `*.config` делает кодовую базу намного более гибкой, поскольку изменение связующего механизма сводится к обновлению содержимого конфигурационного файла и перезапуску приложения. С другой стороны, программная конфигурация обеспечивает приложению более высокую динамическую гибкость — например, связующий механизм можно конфигурировать по-разному в зависимости от определенных условий.

Адрес, привязка и контракт в WCF

Хосты и клиенты взаимодействуют друг с другом путем согласования *адресов*, *привязок* и *контрактов*, которые являются основными строительными блоками приложения WCF и обозначаются посредством аббревиатуры ABC (address, binding, contract — адрес, привязка, контракт).

- *Адрес.* Описывает местоположение службы. В коде он представляется с помощью типа `System.Uri`, однако обычно значение адреса хранится в файлах `*.config`.
- *Привязка.* Инфраструктура WCF поставляется со многими разными привязками, которые указывают сетевые протоколы, механизмы кодирования и транспортный уровень.
- *Контракт.* Предоставляет описание каждого метода в службе WCF, к которому открыт доступ.

Вы должны понимать, что аббревиатура ABC вовсе не подразумевает, что разработчик обязан определить сначала адрес, за ним привязку и в конце контракт. Во многих случаях разработчик WCF начинает с определения контракта для службы, после чего устанавливает адрес и привязки (допустим любой порядок при условии, что учтены все аспекты). Прежде чем перейти к построению первого приложения WCF, давайте более детально рассмотрим ABC.

Понятие контрактов WCF

При построении службы WCF понятие *контракта* считается ключевым. Несмотря на то что это не является обязательным, подавляющее большинство приложений WCF начинается с определения набора интерфейсных типов .NET, применяемых для представления набора членов, которые будет поддерживать заданная служба WCF. В частности, интерфейсы, представляющие контракт WCF, называются *контрактами служб*. Классы (или структуры), которые реализуют их, носят название *типов служб*.

Контракты служб WCF декорируются разнообразными атрибутами, наиболее распространенные из которых определены в пространстве имен `System.ServiceModel`. Когда члены контракта службы (методы в интерфейсе) содержат только простые типы данных (такие как числовые, булевские и строковые), завершенную службу WCF можно построить с использованием только атрибутов `[ServiceContract]` и `[OperationContract]`.

Тем не менее, если члены открывают доступ к специальным типам, то вероятно будут применяться различные типы из пространства имен `System.Runtime.Serialization` (рис. 25.5), находящегося в сборке `System.Runtime.Serialization.dll`. Здесь вы обнаружите дополнительные атрибуты (например, `[DataMember]` и `[DataContract]`), которые предназначены для тонкой настройки процесса определения того, как составные типы будут сериализоваться и десериализоваться из XML при передаче в и из операций службы.

Строго говоря, вы не обязаны использовать интерфейсы CLR для определения контракта WCF. Многие из этих атрибутов могут применяться к открытым членам открытого класса (или структуры). Однако, учитывая многочисленные преимущества программирования на основе интерфейсов (например, полиморфизм и элегантная поддержка множества версий), использование интерфейсов CLR для описания контракта WCF имеет смысл рассматривать как рекомендуемый прием.

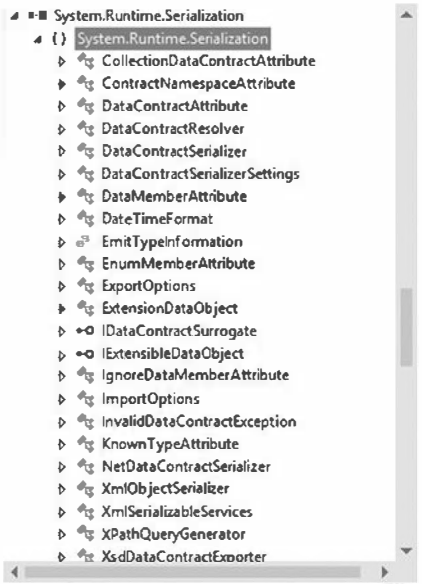


Рис. 25.5. В пространстве имен `System.Runtime.Serialization` определены атрибуты, используемые при построении контрактов данных WCF

Понятие привязок WCF

После определения и реализации контракта (или набора контрактов) в библиотеке службы следующий логический шаг предусматривает построение агента хостинга для самой службы WCF. Как упоминалось ранее, на выбор доступно множество возможных хостов, и все они должны указывать *привязки*, применяемые удаленными клиентами для получения доступа к функциональности типа службы.

Инфраструктура WCF поставляется со многими вариантами привязок, каждая из которых ориентирована на определенные нужды. Если ни одна из готовых привязок не удовлетворяет существующим потребностям, то можно создать собственную привязку, расширив тип `CustomBinding` (в главе это делаться не будет). Привязка WCF может описывать следующие характеристики:

- транспортный уровень, используемый для перемещения данных (HTTP, MSMQ, именованные каналы, REST, WebSockets и TCP);
- каналы, применяемые транспортом (однонаправленные, запрос-ответ и дуплексные);
- механизм кодирования, используемый для работы с самими данными (например, XML и двоичный);
- любые поддерживаемые протоколы веб-служб (если разрешены привязкой), такие как WS-Security, WS-Transactions, WS-Reliability и т.д.

Давайте рассмотрим возможные варианты.

Привязки на основе HTTP

Классы `BasicHttpBinding`, `WSHttpBinding`, `WSDualHttpBinding` и `WSFederationHttpBinding` предназначены для открытия доступа к контрактным типам через протоколы HTTP/SOAP. Если для разрабатываемой службы требуется более широкий доступ (например, из множества операционных систем и множества программных архитектур), то на эти привязки следует обратить внимание, т.к. все они кодируют данные на основе представления XML и применяют в сети протокол HTTP.

В табл. 25.3 показано, как можно представлять привязки WCF в коде (с использованием классов из пространства имен `System.ServiceModel`) или в виде атрибутов XML, определенных внутри файлов `*.config`.

Как и можно было догадаться, класс `BasicHttpBinding` реализует простейший из всех протоколов, ориентированных на веб-службы. В частности, эта привязка обеспечивает соответствие службы WCF спецификации под названием WS-I Basic Profile 1.1 (определенной WS-I). Главная причина использования такой привязки состоит в поддержке обратной совместимости с приложениями, ранее построенными для взаимодействия с веб-службами ASP.NET (которые были частью библиотек .NET, начиная с версии 1.0).

Протокол `WSHttpBinding` не только включает поддержку подмножества спецификации WS-* (транзакции, безопасность и надежные сеансы), но также предоставляет возможность обработки двоичного кодирования данных с применением механизма оптимизации передачи сообщений (Message Transmission Optimization Mechanism — MTOM).

Таблица 25.3. Привязки WCF на основе HTTP

Класс привязки	Элемент привязки	Описание
`BasicHttpBinding`	`<basicHttpBinding>`	Применяется для построения службы WCF, совместимой с профилем WS-Basic Profile (WS-I Basic Profile 1.1). Эта привязка использует HTTP в качестве транспорта и Text/XML в качестве стандартного метода кодирования сообщений
`WSHttpBinding`	`<wsHttpBinding>`	Подобен классу `BasicHttpBinding`, но предоставляет больше функциональных возможностей веб-службы. Эта привязка добавляет поддержку транзакций, надежной доставки сообщений и спецификации WS-Addressing
`WSDualHttpBinding`	`<wsDualHttpBinding>`	Подобен классу `WSHttpBinding`, но предназначен для применения с дуплексными контрактами (например, когда служба и клиент могут отправлять сообщения туда и обратно). Эта привязка поддерживает только безопасность SOAP и требует надежного обмена сообщениями
`WSFederationHttpBinding`	`<wsFederationHttpBinding>`	Безопасная привязка с возможностью взаимодействия, которая поддерживает протокол WS-Federation, позволяя объединенным в федерацию организациям эффективным образом выполнять аутентификацию и авторизацию пользователей

Основное преимущество привязки `WSDualHttpBinding` в том, что она добавляет возможность *дуплексного обмена сообщениями* между отправителем и получателем, что представляет собой причудливый способ указания, что они участвуют в двустороннем взаимодействии. При выборе `WSDualHttpBinding` можно задействовать модель публикации/подписки на события WCF.

И, наконец, `WSFederationHttpBinding` — это протокол на основе веб-служб, об использовании которого стоит подумать, когда крайне важна безопасность в рамках группы организаций. Эта привязка поддерживает спецификации WS-Trust, WS-Security и WS-SecureConversation, которые представлены API-интерфейсами CardSpace в WCF.

Привязки на основе TCP

При построении распределенного приложения, которое функционирует на машинах с установленными библиотеками .NET 4.6 (другими словами, все машины работают под управлением Windows), можно получить выигрыш в производительности за счет обхода привязок веб-служб и применения привязки TCP, обеспечивающей кодирование данных в компактном двоичном формате вместо XML. Когда используются привязки, описанные в табл. 25.4, клиент и хост должны быть приложениями .NET.

Таблица 25.4. Привязки WCF на основе TCP

Класс привязки	Элемент привязки	Описание
NetNamedPipeBinding	<netNamedPipeBinding>	Безопасная, надежная, оптимизированная привязка для коммуникаций между приложениями .NET на одной машине
NetPeerTcpBinding	<netPeerTcpBinding>	Безопасная привязка для сетевых приложений P2P
NetTcpBinding	<netTcpBinding>	Безопасная и оптимизированная привязка, подходящая для межмашинных коммуникаций между приложениями .NET

Для перемещения двоичных данных между клиентом и службой WCF класс NetTcpBinding применяет протокол TCP. Как упоминалось ранее, это дает в результате более высокую производительность по сравнению с протоколами веб-служб, но ограничивается внутренними решениями Windows. Положительной стороной является поддержка классом NetTcpBinding транзакций, надежных сеансов и безопасных коммуникаций.

Подобно NetTcpBinding класс NetNamedPipeBinding поддерживает транзакции, надежные сеансы и безопасные коммуникации; тем не менее, он не обладает возможностью делать межмашинные вызовы. Если вы ищете самый быстрый способ передачи данных между приложениями WCF на одной машине (например, для реализации взаимодействия между приложениями в домене), то привязка NetNamedPipeBinding не будет иметь себе равных. Более подробные сведения о классе NetPeerTcpBinding можно найти в разделе документации .NET Framework 4.6 SDK, посвященном сетям P2P.

Привязки на основе MSMQ

И, наконец, если цель заключается в интеграции с сервером MSMQ, то непосредственный интерес представляют привязки NetMsmqBinding и MsmqIntegrationBinding. Детали использования привязок MSMQ в этой главе не рассматриваются, но в табл. 25.5 описано основное назначение каждой из них.

Таблица 25.5. Привязки WCF на основе MSMQ

Класс привязки	Элемент привязки	Описание
MsmqIntegrationBinding	<msmqIntegrationBinding>	Эту привязку можно применять для того, чтобы позволить приложениям WCF отправлять и получать сообщения от существующих приложений MSMQ, которые используют COM, собственный C++ или типы, определенные в пространстве имен System.Messaging
NetMsmqBinding	<netMsmqBinding>	Эту привязку на основе очередей можно применять для межмашинных коммуникаций между приложениями .NET. В рамках привязок, основанных на MSMQ, такой подход является предпочтительным

Понятие адресов WCF

После установления контрактов и привязок остается указать *адрес* для службы WCF. Это важно, поскольку удаленные вызывающие компоненты не смогут взаимодействовать с удаленными типами, если не удастся найти их местоположение. Подобно большинству аспектов WCF адрес может быть жестко закодирован в сборке (с использованием типа `System.Uri`) или вынесен в файл `*.config`.

В любом случае точный формат адреса WCF будет отличаться в зависимости от выбранной привязки (на основе HTTP, именованные каналы, на основе TCP или на основе MSMQ). На самом высоком уровне адреса WCF могут задавать перечисленные ниже единицы информации.

- `Scheme`. Транспортный протокол (например, HTTP).
- `MachineName`. Полностью заданное доменное имя машины.
- `Port`. Во многих ситуациях это необязательный параметр; скажем, привязка HTTP по умолчанию работает с портом 80.
- `Path`. Путь к службе WCF.

Указанная информация может быть представлена с помощью следующего обобщенного шаблона (значение `Port` необязательно, т.к. в некоторых привязках порт не применяется):

```
Scheme://<MachineName>[:Port]/Path
```

Когда используется привязка на основе HTTP (`basicHttpBinding`, `wsHttpBinding`, `wsDualHttpBinding` или `wsFederationHttpBinding`), адрес разбивается примерно так (вспомните, что если номер порта не указан, то протоколы на основе HTTP по умолчанию выбирают порт 80):

```
http://localhost:8080/MyWCFService
```

Если применяется привязка, основанная на TCP (вроде `NetTcpBinding` или `NetPeerTcpBinding`), то URI принимает следующий формат:

```
net.tcp://localhost:8080/MyWCFService
```

Привязки на основе MSMQ (`NetMsmqBinding` и `MsmqIntegrationBinding`) уникальны в своем формате URI, потому что MSMQ может использовать открытые или закрытые очереди (доступные только на локальной машине), а номера портов не имеют смысла в URI, связанных с MSMQ. Взгляните на приведенный ниже URI, который описывает закрытую очередь по имени `MyPrivateQ`:

```
net.msmq://localhost/private$/MyPrivateQ
```

И последнее, но не менее важное замечание: формат адреса, применяемый привязкой для именованных каналов (`NetNamedPipeBinding`), выглядит так, как показано ниже (вспомните, что именованные каналы делают возможными межпроцессные коммуникации приложений на одной физической машине):

```
net.pipe://localhost/MyWCFService
```

Хотя одиночная служба WCF может открывать доступ только к одному адресу (основанному на единственной привязке), есть возможность сконфигурировать коллекцию уникальных адресов (с разными привязками). Это делается в файле `*.config` за счет определения множества элементов `<endpoint>`. Для одной и той же службы можно указывать любое количество ABC. Такой подход полезен, когда необходимо позволить вызывающим компонентам выбирать протокол, который они желают использовать для взаимодействия со службой.

Построение службы WCF

Теперь, когда вы получили представление о строительных блоках приложения WCF, наступило время создать первый пример приложения, чтобы посмотреть, как ABC учитываются в коде и конфигурации. В первом примере шаблоны проектов WCF из Visual Studio не применяются, так что можно будет сконцентрироваться на специфических шагах по созданию службы WCF.

Начнем с создания нового проекта библиотеки классов C# по имени `MagicEightBallServiceLib`. Переименуем начальный файл `Class1.cs` в `MagicEightBallService.cs` и добавим ссылку на сборку `System.ServiceModel.dll`. Добавим в начальный файл кода оператор `using` для пространства имен `System.ServiceModel`. К этому моменту файл C# должен выглядеть следующим образом (обратите внимание, что в настоящее время мы имеем открытый класс):

```
// Основное пространство имен WCF.
using System.ServiceModel;

namespace MagicEightBallServiceLib
{
  public class MagicEightBallService
  {
  }
}
```

В классе реализован единственный контракт службы WCF, представленный строго типизированным интерфейсом CLR по имени `IEightBall`. Как вам наверняка известно, магический шар Magic 8-Ball — это игрушка, позволяющая получить ответ на задаваемый вопрос из набора фиксированных ответов. В интерфейсе будет определен единственный метод, который позволит клиенту задать вопрос магическому шару, чтобы получить случайный ответ.

Интерфейсы служб WCF оснащены атрибутом `[ServiceContract]`, в то время как каждый член интерфейса декорирован атрибутом `[OperationContract]` (вскоре вы получите больше сведений об этих двух атрибутах). Вот определение интерфейса `IEightBall`:

```
[ServiceContract]
public interface IEightBall
{
  // Задайте вопрос, получите ответ!
  [OperationContract]
  string ObtainAnswerToQuestion(string userQuestion);
}
```

На заметку! Допускается определять интерфейс контракта службы, который содержит методы, не оснащенные атрибутом `[OperationContract]`; однако к таким членам не будет открываться доступ через исполняющую среду WCF.

Как известно из главы 8, интерфейс — довольно бесполезная вещь, пока он не реализован классом или структурой, которая наполняет его функциональностью. Подобно реальному магическому шару реализация типа службы (`MagicEightBallService`) будет возвращать случайно выбранный подготовленный ответ из массива строк. Стандартный конструктор будет отображать информационное сообщение, которое (в конечном итоге) выводится в окне консоли хоста (в целях диагностики):

```
public class MagicEightBallService : IEightBall
{
  // Просто для отображения на хосте.
  public MagicEightBallService()
  {
    Console.WriteLine("The 8-Ball awaits your question...");
  }
  public string ObtainAnswerToQuestion(string userQuestion)
  {
    string[] answers = { "Future Uncertain", "Yes", "No",
      "Hazy", "Ask again later", "Definitely" };
    // Возвратить случайный ответ.
    Random r = new Random();
    return answers[r.Next(answers.Length)];
  }
}
```

На этом библиотека службы WCF завершена. Тем не менее, перед конструированием хоста для этой службы необходимо ознакомимся с дополнительными деталями, касающимися атрибутов `[ServiceContract]` и `[OperationContract]`.

Атрибут [ServiceContract]

Чтобы интерфейс CLR принимал участие в службах, предоставляемых WCF, он должен быть декорирован атрибутом `[ServiceContract]`. Подобно многим другим атрибутам .NET тип `ServiceContractAttribute` поддерживает много свойств, с помощью которых производится дальнейшее уточнение его целевого назначения. Два свойства, `Name` и `NameSpace`, могут быть установлены для управления именем типа службы и названием пространства имен XML, где тип службы определен. Если используется привязка HTTP, то эти значения применяются для определения элементов `<portType>` связанного документа WSDL.

Здесь мы не заботимся о присваивании значения свойству `Name`, т.к. стандартное имя типа службы основано прямо на имени класса C#. Однако стандартным названием для лежащего в основе пространства имен XML будет просто `http://tempuri.org` (оно должно быть изменено для всех создаваемых служб WCF).

При построении службы WCF, которая будет отправлять и получать данные специальных типов (чего мы в настоящий момент не делаем), важно установить осмысленное значение для лежащего в основе пространства имен XML, поскольку это обеспечивает уникальность специальных типов. Как вам может быть известно из опыта построения веб-служб XML, пространства имен XML предоставляют способ помещения специальных типов в уникальный контейнер, гарантируя отсутствие конфликтов с типами из другой организации.

По этой причине определение интерфейса можно обновить, указав в качестве пространства имен URI места происхождения службы, что очень похоже на процесс определения пространства имен XML в проекте веб-службы .NET:

```
[ServiceContract(Namespace = "http://MyCompany.com")]
public interface IEightBall
{
  ...
}
```

Помимо `Namespace` и `Name` атрибут `[ServiceContract]` может быть сконфигурирован посредством дополнительных свойств, которые кратко описаны в табл. 25.6. Имейте в виду, что в зависимости от выбранной привязки некоторые из перечисленных настроек будут игнорироваться.

Таблица 25.6. Разнообразные именованные свойства атрибута [ServiceContract]

Свойство	Описание
CallbackContract	Устанавливает, требует ли этот контракт службы функциональность обратного вызова для двустороннего обмена сообщениями (например, дуплексные привязки)
ConfigurationName	Определяет местоположение элемента службы в конфигурационном файле приложения. По умолчанию представляет собой имя класса, реализующего службу
ProtectionLevel	Позволяет указать степень, до которой привязка контракта требует шифрования, цифровых подписей или того и другого для конечных точек, открываемых контрактом
SessionMode	Устанавливает, разрешены ли сеансы, не разрешены или являются обязательными для этого контракта службы

Атрибут [OperationContract]

Методы, которые планируется использовать внутри инфраструктуры WCF, должны быть декорированы атрибутом [OperationContract], который также может быть сконфигурирован с помощью различных именованных свойств. Свойства, описанные в табл. 25.7, можно применять для объявления о том, что заданный метод предназначен для однонаправленной работы, поддерживает асинхронные вызовы, требует шифрования данных сообщений и т.д. (опять-таки, в зависимости от выбранной привязки многие из этих значений могут быть проигнорированы).

Таблица 25.7. Различные именованные свойства атрибута [OperationContract]

Свойство	Описание
AsyncPattern	Указывает, реализована ли операция асинхронно с использованием пары методов Begin/End службы. Это позволяет службе передавать обработку другому потоку серверной стороны, но не имеет ничего общего с асинхронным вызовом метода клиентом!
IsInitiating	Указывает, может ли эта операция быть начальной операцией в сеансе
IsOneWay	Указывает, состоит ли операция только из одного входного сообщения (и не имеет ассоциированного вывода)
IsTerminating	Указывает, должна ли исполняющая среда WCF пытаться завершить текущий сеанс после выполнения операции

В этом начальном примере дополнительное конфигурирование метода ObtainAnswerToQuestion() не требуется, т.е. атрибут [OperationContract] можно оставить в его текущем виде.

Типы служб как контракты операций

И, наконец, вспомните, что при построении типов служб WCF использовать интерфейсы не обязательно. На самом деле атрибуты [ServiceContract] и [OperationContract] можно применять прямо к самому типу службы:

```
// Только в целях иллюстрации;
// в текущем примере не используется.
[ServiceContract(Namespace = "http://MyCompany.com")]
```

```
public class ServiceTypeAsContract
{
  [OperationContract]
  void SomeMethod() { }

  [OperationContract]
  void AnotherMethod() { }
}
```

Вы можете принять такой подход; тем не менее, явное определение интерфейсного типа для представления контракта службы обеспечивает массу преимуществ. Самый очевидный выигрыш заключается в том, что отдельно взятый интерфейс можно применять к нескольким типам служб (написанных с использованием разных языков и архитектур) и достичь высокой степени полиморфизма. Еще одно преимущество связано с тем, что интерфейс контракта службы может выступать в качестве основы для новых контрактов (с применением наследования интерфейсов) без необходимости заботиться о реализации.

Итак, разработка первой библиотеки службы WCF завершена. Скомпилируем проект, чтобы удостовериться в отсутствии опечаток.

Исходный код. Проект `MagicEightBallServiceLib` доступен в подкаталоге `MagicEightBallServiceHTTP` внутри подкаталога `Chapter_25`.

Хостинг службы WCF

Теперь все готово для определения хоста. Хотя служба производственного уровня должна размещаться в Windows-службе или в виртуальном каталоге IIS, наш первый хост будет просто консольным приложением по имени `MagicEightBallServiceHost`.

После создания нового проекта консольного приложения добавим ссылку на сборки `System.ServiceModel.dll` и `MagicEightBallServiceLib.dll`, после чего обновим начальный файл кода, импортировав пространства имен `System.ServiceModel` и `MagicEightBallServiceLib`:

```
using System;
...
using System.ServiceModel;
using MagicEightBallServiceLib;

namespace MagicEightBallServiceHost
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Console Based WCF Host *****");
      Console.ReadLine();
    }
  }
}
```

Первый шаг, который должен быть предпринят при построении хоста для служебного типа службы WCF, касается решения относительно того, определяется ли необходимая логика хостинга полностью в коде или же несколько низкоуровневых деталей будут перемещены в конфигурационный файл приложения. Как упоминалось ранее, преимущество файлов `*.config` заключается в том, что хост может изменять связующий механизм

без перекомпиляции и повторного развертывания исполняемого файла. Однако всегда помните о том, что это совершенно не обязательно, поскольку логику хостинга можно жестко закодировать с использованием типов из сборки `System.ServiceModel.dll`.

В этом консольном хосте будет применяться конфигурационный файл приложения, поэтому добавим новый такой файл (если его еще нет в проекте) в текущий проект посредством выбора пункта меню Project⇨Add New Item (Проект⇨Добавить новый элемент) и указания элемента Application Configuration File (Конфигурационный файл приложения).

Установка ABC внутри файла `App.config`

При построении хоста для типа службы WCF необходимо следовать предсказуемому набору шагов, часть из которых полагается на конфигурацию, а часть — на код.

- Определить конечную точку для службы WCF в конфигурационном файле хоста.
- Использовать в коде тип `ServiceHost` для открытия доступа к типам служб, видимых из этой конечной точки.
- Обеспечить, чтобы хост остался функционирующим для обслуживания входящих клиентских запросов. Очевидно, что данный шаг не обязателен, если для хостинга применяется Windows-служба или IIS.

В мире WCF термин *конечная точка* представляет адрес, привязку и контракт, которые вместе объединены в аккуратный пакет. В XML конечная точка выражается с использованием элемента `<endpoint>` и его атрибутов `address`, `binding` и `contract`. Модифицируем файл `*.config`, указав в нем единственную конечную точку (достижимую через порт 8080), доступ к которой открывает этот хост:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService">
        <endpoint address = "http://localhost:8080/MagicEightBallService"
                  binding = "basicHttpBinding"
                  contract = "MagicEightBallServiceLib.IEightBall"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Обратите внимание, что элемент `<system.serviceModel>` является корнем всех настроек WCF хоста. Каждая служба, открываемая хостом, представлена элементом `<service>`, который находится внутри базового элемента `<services>`. Здесь в единственном элементе `<service>` для указания дружественного имени типа службы применяется (необязательный) атрибут `name`.

С помощью вложенного элемента `<endpoint>` устанавливается адрес, модель привязки (`basicHttpBinding` в этом примере) и полностью заданное имя интерфейсного типа, определяющего контракт службы WCF (`IEightBall`). Поскольку используется привязка на основе HTTP, применяется схема `http://` с указанием произвольного номера порта.

Кодирование с использованием типа `ServiceHost`

Благодаря текущему конфигурационному файлу действительная программная логика, требуемая для завершения хоста, будет простой. Когда исполняемая программа

запускается, создается экземпляр типа ServiceHost, которому сообщается служба WCF, отвечающая за хостинг. Во время выполнения этот объект автоматически читает данные из элемента <system.serviceModel> в файле *.config хоста для определения корректного адреса, привязки и контракта. Затем объект создает необходимый связующий механизм:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Console Based WCF Host *****");
  using (ServiceHost serviceHost = new ServiceHost(typeof(MagicEightBallService)))
  {
    // Открыть хост и начать прослушивание входящих сообщений.
    serviceHost.Open();

    // Оставить службу функционирующей до тех пор,
    // пока не будет нажата клавиша <Enter>.
    Console.WriteLine("The service is ready.");
    Console.WriteLine("Press the Enter key to terminate service.");
    Console.ReadLine();
  }
}
```

После запуска приложения обнаружится, что хост расположился в памяти и готов к приему входящих запросов от удаленных клиентов.

На заметку! Вспомните, что для выполнения многих типов проектов WCF среда Visual Studio должна быть запущена с административными привилегиями!

Указание базовых адресов

В настоящий момент объект ServiceHost создается с применением конструктора, который требует только указания информации о типе службы. Тем не менее, в качестве аргумента конструктору можно также передавать массив элементов типа System.Uri, чтобы представить коллекцию адресов, по которым доступна данная служба. В текущее время адрес находится с использованием файла *.config. Однако предположим, что область using модифицирована следующим образом:

```
using (ServiceHost serviceHost = new
  ServiceHost(typeof(MagicEightBallService),
  new Uri[]{new Uri("http://localhost:8080/MagicEightBallService")}))
{
  ...
}
```

Теперь конечную точку можно определить так:

```
<endpoint address = ""
          binding = "basicHttpBinding"
          contract = "MagicEightBallServiceLib.IEightBall"/>
```

Разумеется, слишком большой объем жесткого кодирования внутри кодовой базы хоста снижает гибкость, поэтому в текущем примере предполагается, что хост службы создается просто путем предоставления информации о типе, как делалось раньше:

```
using (ServiceHost serviceHost = new ServiceHost(typeof(MagicEightBallService)))
{
  ...
}
```

Слегка обескураживающий аспект написания файлов *.config для хостов связан с тем, что в зависимости от объема жесткого кодирования существует несколько способов создания дескрипторов XML (как было в случае с необязательным массивом Uri). Показанная ниже модификация демонстрирует еще один способ написания файла *.config:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService">
        <!-- Адрес получен из <baseAddresses> -->
        <endpoint address = ""
                  binding = "basicHttpBinding"
                  contract = "MagicEightBallServiceLib.IEightBall"/>
        <!-- Перечислить все базовые адреса в выделенном разделе -->
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

В этом случае атрибут address элемента <endpoint> по-прежнему пуст; невзирая на то, что при создании ServiceHost массив Uri в коде не указывается, приложение функционирует, как и ранее, т.к. нужное значение извлекается из элемента baseAddresses. Преимущество хранения базового адреса в разделе <baseAddresses> элемента <host> связано с тем, что другим частям файла *.config также необходимо знать адрес конечной точки службы. Таким образом, вместо копирования и вставки значений адресов внутрь файла *.config можно изолировать единственное значение, как было показано в предшествующем фрагменте.

На заметку! В примере, приведенном позже в главе, будет представлен графический инструмент конфигурирования, который позволяет создавать конфигурационные файлы менее утомительным способом.

В любом случае перед построением клиентского приложения, предназначенного для взаимодействия со службой, придется проделать чуть больше работы. В частности, необходимо более глубоко изучить роль класса ServiceHost и элемента <service.serviceModel>, а также роль служб обмена метаданными (metadata exchange — MEX).

Подробный анализ типа ServiceHost

Класс ServiceHost применяется для конфигурирования и открытия доступа к службе WCF из размещающей исполняемой сборки. Тем не менее, имейте в виду, что вы будете использовать этот тип напрямую только при построении специальной сборки *.exe для размещения ваших служб. Если для открытия доступа к службе применяется IIS, то объект ServiceHost создается автоматически.

Как уже было показано, данный тип требует полного описания службы, которое получается динамически через конфигурационные настройки файла *.config хоста. Хотя это происходит автоматически при создании объекта, состояние объекта

ServiceHost можно сконфигурировать вручную с помощью его членов. Помимо методов Open() и Close() (которые взаимодействуют со службой в синхронной манере) класс ServiceHost имеет и другие члены, перечисленные в табл. 25.8.

Таблица 25.8. Избранные члены класса ServiceHost

Член	Описание
Authorization	Это свойство получает уровень авторизации для размещаемой службы
AddDefaultEndpoints()	Этот метод используется для программного конфигурирования хоста службы WCF, чтобы он воспринимал любое количество готовых конечных точек, предоставленных инфраструктурой
AddServiceEndpoint()	Этот метод позволяет программно регистрировать конечную точку для хоста
BaseAddresses	Это свойство получает список зарегистрированных базовых адресов для текущей службы
BeginOpen() BeginClose()	Эти методы позволяют асинхронно открывать и закрывать объект ServiceHost с применением стандартного асинхронного синтаксиса делегатов .NET
CloseTimeout	Это свойство позволяет устанавливать и получать время, отведенное службе на закрытие
Credentials	Это свойство получает удостоверения безопасности, используемые текущей службой
EndOpen() EndClose()	Эти методы являются асинхронными аналогами методов BeginOpen() и BeginClose()
OpenTimeout	Это свойство позволяет устанавливать и получать время, отведенное службе на запуск
State	Это свойство получает значение, которое указывает текущее состояние коммуникационного объекта, представленное перечислением CommunicationState (например, Opened, Closed, Created)

Чтобы продемонстрировать некоторые дополнительные аспекты ServiceHost в действии, модифицируем класс Program, добавив новый статический метод, который выводит на консоль ABC каждой конечной точки, применяемой хостом:

```
static void DisplayHostInfo(ServiceHost host)
{
  Console.WriteLine();
  Console.WriteLine("***** Host Info *****");
  foreach (System.ServiceModel.Description.ServiceEndpoint se
    in host.Description.Endpoints)
  {
    Console.WriteLine("Address: {0}", se.Address);        // Адрес
    Console.WriteLine("Binding: {0}", se.Binding.Name);   // Привязка
    Console.WriteLine("Contract: {0}", se.Contract.Name); // Контракт
    Console.WriteLine();
  }
  Console.WriteLine("**********************");
}
```

Предположим, что новый метод вызывается внутри Main() после открытия хоста:

```
using (ServiceHost serviceHost = new ServiceHost(typeof(MagicEightBallService)))
{
  // Открыть хост и начать прослушивание входящих сообщений.
  serviceHost.Open();
  DisplayHostInfo(serviceHost);
  ...
}
```

В результате выводится следующая статистика:

```
***** Console Based WCF Host *****
***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
**********************
The service is ready.
Press the Enter key to terminate service.
```

На заметку! При запуске хоста (или клиента) в этой главе удостоверьтесь в том, что действительно "запускаете" программу из Visual Studio (<Ctrl+F5>), а не отлаживаете (<F5>) ее, чтобы обеспечить выполнение процессов хоста и клиента независимым образом.

Подробный анализ элемента <system.serviceModel>

Подобно любому элементу XML внутри <system.serviceModel> допускается определять набор подэлементов, каждый из которых может быть снабжен разнообразными атрибутами. Несмотря на то что за подробными сведениями о наборе возможных атрибутов следует обращаться в документацию .NET Framework 4.6 SDK, ниже приведен каркас со списком некоторых (но не всех) полезных подэлементов:

```
<system.serviceModel>
  <behaviors>
  </behaviors>
  <client>
  </client>
  <commonBehaviors>
  </commonBehaviors>
  <diagnostics>
  </diagnostics>
  <comContracts>
  </comContracts>
  <services>
  </services>
  <bindings>
  </bindings>
</system.serviceModel>
```

Далее в главе вы увидите более экзотические конфигурационные файлы; избранные подэлементы кратко описаны в табл. 25.9.

Включение обмена метаданными

Вспомните, что клиентские приложения WCF взаимодействуют со службой WCF через промежуточный тип прокси. Наряду с тем, что код прокси можно было бы писать целиком вручную, это будет утомительным и подверженным ошибкам процессом.

Таблица 25.9. Избранные подэлементы <service.serviceModel>

Подэлемент	Описание
behaviors	Инфраструктура WCF поддерживает различные линии поведения конечных точек и служб. По существу *линия поведения* позволяет дополнительно уточнять функциональность хоста, службы или клиента
bindings	Этот элемент позволяет тонко настраивать каждую привязку WCF (например, basicHttpBinding и netMsmqBinding), а также указывать любые специальные привязки, используемые хостом
client	Этот элемент содержит список конечных точек, которые клиент применяет для подключения к службе. Очевидно, что он не особенно полезен в файле *.config хоста
comContracts	Этот элемент определяет контракты COM, обеспечивающие возможность взаимодействия WCF и COM
commonBehaviors	Этот элемент может устанавливаться только внутри файла machine.config. Он используется для определения всех линий поведения, применяемых каждой службой WCF на заданной машине
diagnostics	Этот элемент содержит настройки для средств диагностики WCF. Пользователь может включать/отключать трассировку, счетчики производительности и поставщика WMI, а также добавлять специальные фильтры сообщений
services	Этот элемент содержит коллекцию служб WCF, к которым хост открывает доступ

В идеале должен использоваться какой-то инструмент для генерации необходимого рутинного кода (включая файл *.config клиентской стороны). К счастью, в .NET Framework 4.6 SDK доступен инструмент командной строки (svcutil.exe), предназначенный именно для этой цели. Вдобавок Visual Studio предлагает похожую функциональность, доступную в результате выбора пункта меню Project⇨Add Service Reference (Проект⇨Добавить ссылку на службу).

Однако для того, чтобы упомянутые инструменты генерировали нужный код прокси и файл *.config, они должны быть в состоянии выяснять формат интерфейсов службы WCF и любых определенных контрактов данных (т.е. имена методов и типы параметров).

Обмен метаданными (metadata exchange — MEX) — это *линия поведения службы* WCF, которая может применяться для тонкой настройки способа обработки службы исполняющей средой WCF. Выражаясь просто, каждый элемент <behavior> позволяет определять набор действий, на которые заданная служба может подписываться. Инфраструктура WCF предоставляет многочисленные линии поведения в готовом виде, к тому же можно строить собственные линии поведения.

Линия поведения MEX (которая по умолчанию отключена) будет перехватывать любые запросы метаданных, отправленные посредством HTTP-запроса GET. Чтобы позволить инструменту svcutil.exe или Visual Studio автоматизировать создание требуемого прокси клиентской стороны и файла *.config, понадобится включить MEX.

Включение MEX предусматривает корректировку файла *.config хоста с помощью подходящих настроек (или написание соответствующего кода C#). Во-первых, необходимо добавить новый элемент <endpoint> конкретно для MEX. Во-вторых, нужно определить линию поведения WCF для разрешения доступа HTTP-запросам GET. В-третьих, эту линию поведения потребуется ассоциировать по имени со службой, используя атрибут behaviorConfiguration в открывающем элементе <service>.

Наконец, в-четвертых, понадобится добавить элемент <host> для определения базового класса службы (MEX будет искать здесь местоположения описываемых типов).

На заметку! Финальный шаг можно опустить, если для представления базового адреса конструктору класса SeviceHost передается в качестве параметра объект System.Uri.

Взгляните на следующий измененный файл *.config хоста, который создает специальный элемент <behavior> (по имени EightBallServiceMEXBehavior), ассоциированный со службой через атрибут behaviorConfiguration внутри определения <service>:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService"
               behaviorConfiguration="EightBallServiceMEXBehavior">
        <endpoint address = ""
                  binding = "basicHttpBinding"
                  contract = "MagicEightBallServiceLib.IEightBall"/>
        <!-- Включить конечную точку MEX -->
        <endpoint address = "mex"
                  binding = "mexHttpBinding"
                  contract = "IMetadataExchange" />
<!-- Это необходимо добавить, чтобы MEX был известен адрес нашей службы -->
        <host>
          <baseAddresses>
           <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
    <!-- Определение линии поведения для MEX -->
    <behaviors>
      <serviceBehaviors>
        <behavior name = "EightBallServiceMEXBehavior" >
          <serviceMetadata httpGetEnabled = "true" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Теперь приложение хоста службы можно перезапустить и просмотреть описание метаданных в веб-браузере. Для этого при функционирующем хосте необходимо ввести в строке адреса такой URL:

```
http://localhost:8080/MagicEightBallService
```

На домашней странице службы WCF (рис. 25.6) предоставляются базовые сведения о том, как программно взаимодействовать с этой службой. Щелчок на гиперссылке в верхней части страницы позволяет просмотреть контракт WSDL. Вспомните, что язык описания веб-служб (Web Service Description Language — WSDL) — это грамматика, описывающая структуру веб-служб в заданной конечной точке.

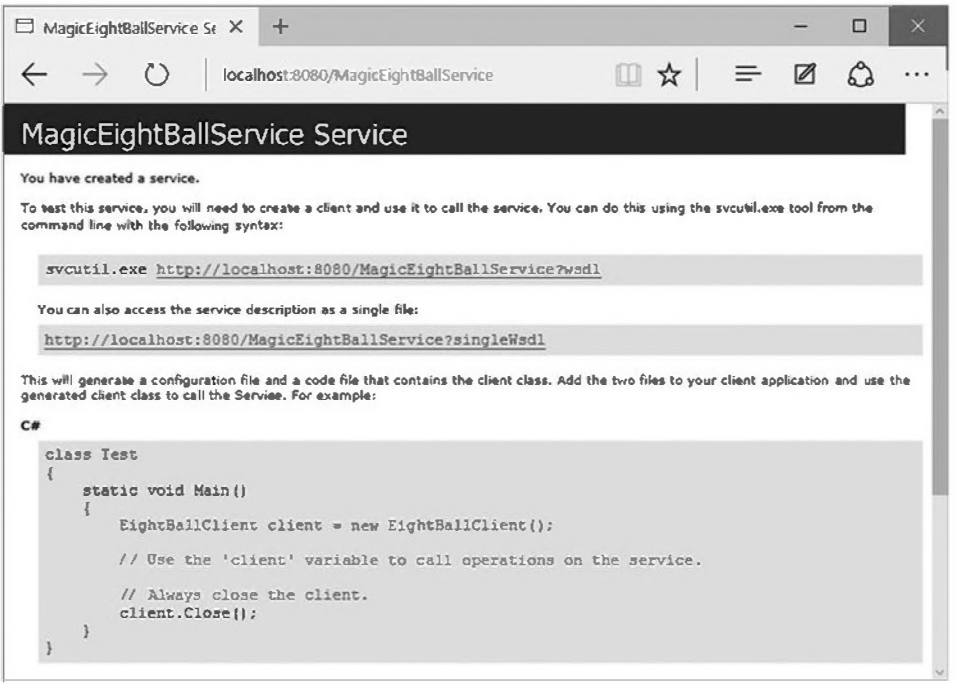


Рис. 25.6. Просмотр метаданных с применением MEX

Хост теперь открывает доступ к двум разным конечным точкам (одна для службы и одна для MEX), поэтому консольный вывод хоста будет выглядеть следующим образом:

```
***** Console Based WCF Host *****
***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
Address: http://localhost:8080/MagicEightBallService/mex
Binding: MetadataExchangeHttpBinding
Contract: IMetadataExchange
**********************
The service is ready.
```

Исходный код. Проект `MagicEightBallServiceHost` доступен в подкаталоге `MagicEightBallServiceHTTP` внутри подкаталога `Chapter_25`.

Построение клиентского приложения WCF

Теперь, имея готовый хост, последняя задача заключается в построении фрагмента программного обеспечения для взаимодействия с этим типом службы WCF. Хотя можно избрать длинный путь и построить всю необходимую инфраструктуру вручную (осуществимая, но трудоемкая задача), в .NET Framework 4.6 SDK предлагается несколько подходов для быстрой генерации прокси клиентской стороны. Начнем с создания нового проекта консольного приложения по имени `MagicEightBallServiceClient`.

Генерация кода прокси с использованием `svcutil.exe`

Первый способ построения прокси клиентской стороны предусматривает применение инструмента командной строки `svcutil.exe`. С его помощью можно генерировать новый файл на языке C#, представляющий сам код прокси, а также конфигурационный файл клиентской стороны. Для этого в первом параметре указывается конечная точка службы. Параметр `/out:` используется для определения имени файла `*.cs`, содержащего код прокси, а параметр `/config:` позволяет указать имя генерируемого файла `*.config` клиентской стороны.

Предполагая, что служба в текущий момент запущена, следующий набор параметров, переданный `svcutil.exe`, приведет к генерации двух новых файлов в рабочем каталоге (вся команда с параметрами должна вводиться в одной строке):

```
svcutil http://localhost:8080/MagicEightBallService
      /out:myProxy.cs /config:app.config
```

Если открыть файл `myProxy.cs`, то в нем можно обнаружить представление клиентской стороны интерфейса `IEightBall`, а также новый класс по имени `EightBallClient`, который и является классом прокси. Этот класс будет производным от обобщенного класса `System.ServiceModel.ClientBase<T>`, где `T` — зарегистрированный интерфейс службы.

В дополнение к нескольким специальным конструкторам каждый метод класса прокси (который основан на исходных методах интерфейса) будет реализован для применения унаследованного свойства `Channels` с целью вызова корректного метода службы. Вот частичный код типа прокси:

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",
                                                "4.0.0.0")]
public partial class EightBallClient :
  System.ServiceModel.ClientBase<IEightBall>, IEightBall
{
  ...
  public string ObtainAnswerToQuestion(string userQuestion)
  {
    return base.Channel.ObtainAnswerToQuestion(userQuestion);
  }
}
```

При создании экземпляра типа прокси в клиентском приложении базовый класс установит подключение к конечной точке с использованием настроек, указанных в конфигурационном файле приложения клиентской стороны. Во многом похоже на конфигурационный файл серверной стороны сгенерированный файл `App.config` клиентской стороны содержит элемент `<endpoint>` и детали о привязке `basicHttpBinding`, которая применяется для взаимодействия со службой.

Вы также найдете следующий элемент `<client>`, который устанавливает ABC с точки зрения клиента:

```
<client>
  <endpoint
   address = "http://localhost:8080/MagicEightBallService"
   binding = "basicHttpBinding"
   bindingConfiguration = "BasicHttpBinding_IEightBall"
   contract = "IEightBall" name = "BasicHttpBinding_IEightBall" />
</client>
```

В данный момент можно было бы включить эти два файла в проект клиента (вместе со ссылкой на сборку `System.ServiceModel.dll`) и затем использовать тип прокси для коммуникаций с удаленной службой WCF. Тем не менее, мы примем другой подход и посмотрим, каким образом Visual Studio может помочь в дальнейшей автоматизации создания файлов прокси клиентской стороны.

Генерация кода прокси в Visual Studio

Подобно любому хорошему инструменту командной строки в утилите `svcutil.exe` предусмотрено огромное количество параметров, которые можно применять для управления генерацией прокси. Если расширенные параметры не нужны, то те же два файла можно сгенерировать в IDE-среде Visual Studio. Для этого нужно создать новый проект консольного приложения и выбрать пункт Add Service Reference (Добавить ссылку на службу) в меню Project (Проект).

После выбора этого пункта меню будет предложено ввести URI службы. Щелчок на кнопке Go (Перейти) позволяет просмотреть описание службы (рис. 25.7).

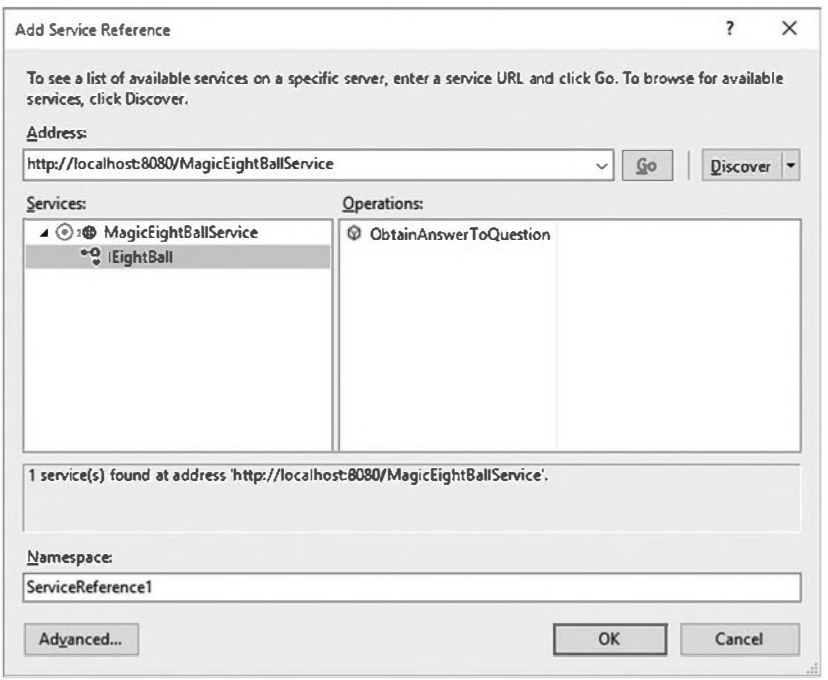


Рис. 25.7. Генерация файлов прокси в Visual Studio

Помимо создания и вставки файлов прокси в текущий проект будут автоматически добавлены ссылки на сборки WCF. В соответствии с соглашением об именовании класс прокси определен внутри пространства имен `ServiceReference1`, которое вложено в пространство имен клиента (во избежание возможных конфликтов имен). Ниже приведен полный код клиента:

```
// Местоположение прокси.
using MagicEightBallServiceClient.ServiceReference1;

namespace MagicEightBallServiceClient
{
```

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Ask the Magic 8 Ball *****\n");
    using (EightBallClient ball = new EightBallClient())
    {
      Console.Write("Your question: ");
      string question = Console.ReadLine();
      string answer =
        ball.ObtainAnswerToQuestion(question);
      Console.WriteLine("8-Ball says: {0}", answer);
    }
    Console.ReadLine();
  }
}
```

Предполагая, что хост WCF запущен, можно выполнить программу клиента. Вот возможный вывод:

```
***** Ask the Magic 8 Ball *****

Your question: Will I ever finish Fallout 4?
8-Ball says: No

Press any key to continue . . .
```

Исходный код. Проект `MagicEightBallServiceClient` доступен в подкаталоге `MagicEightBallServiceHTTP` внутри подкаталога `Chapter_25`.

Конфигурирование привязки на основе TCP

К настоящему моменту приложения хоста и клиента сконфигурированы для использования простейшей из привязок, основанных на HTTP — `basicHttpBinding`. Вспомните, что преимуществом выноса настроек в конфигурационные файлы является возможность изменения внутреннего связующего механизма в декларативной манере и открытия доступа к множеству привязок для одной и той же службы.

В целях иллюстрации проведем небольшой эксперимент. Создадим новую папку на диске C: (или там, где был сохранен код) по имени `EightBallTCP` и внутри нее две папки `Host` и `Client`.

Затем в проводнике Windows перейдем в папку `\bin\Debug` проекта хоста (рассмотренного ранее в главе) и скопируем файлы `MagicEightBallServiceHost.exe`, `MagicEightBallServiceHost.exe.config` и `MagicEightBallServiceLib.dll` в папку `C:\EightBallTCP\Host`. Откроем файл `*.config` в простом текстовом редакторе и модифицируем его содержимое следующим образом:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService">
        <endpoint address = ""
                  binding = "netTcpBinding"
                  contract = "MagicEightBallServiceLib.IEightBall"/>
        <host>
          <baseAddresses>
        <add baseAddress = "net.tcp://localhost:8090/MagicEightBallService"/>
```

```
        </baseAddresses>
      </host>
    </service>
  </services>
 </system.serviceModel>
</configuration>
```

По существу из файла `*.config` удалены все настройки MEX (т.к. уже создан прокси) и установлено применение типа привязки `NetTcpBinding` через уникальный порт. Запустим приложение, дважды щелкнув на файле `*.exe`. Если все было сделано правильно, то должен появиться показанный ниже вывод:

```
***** Console Based WCF Host *****

***** Host Info *****
Address: net.tcp://localhost:8090/MagicEightBallService
Binding: NetTcpBinding
Contract: IEightBall

**********************
The service is ready.
Press the Enter key to terminate service.
```

Для завершения теста скопируем файлы `MagicEightBallServiceClient.exe` и `MagicEightBallServiceClient.exe.config` из папки `\bin\Debug` клиентского приложения (обсуждалось ранее в главе) в папку `C:\EightBallTCP\Client`. Модифицируем конфигурационный файл следующим образом:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <client>
      <endpoint address = "net.tcp://localhost:8090/MagicEightBallService"
                binding = "netTcpBinding"
                contract = "ServiceReference1.IEightBall"
                name = "netTcpBinding_IEightBall" />
    </client>
  </system.serviceModel>
</configuration>
```

Данный конфигурационный файл клиентской стороны значительно проще файла, создаваемого генератором прокси Visual Studio. Обратите внимание, что существующий элемент `<bindings>` удален. Первоначально файл `*.config` содержал элемент `<bindings>` с подэлементом `<basicHttpBinding>`, который определял множество деталей, касающихся настроек привязки клиента (например, таймауты).

На самом деле для рассматриваемого примера такие детали никогда не понадобятся, поскольку мы автоматически получаем стандартные значения лежащего в основе объекта `BasicHttpBinding`. Конечно, при необходимости можно было бы обновить существующий элемент `<bindings>`, определив детали подэлемента `<netTcpBinding>`, но это совершенно не обязательно, если устраивают стандартные значения объекта `NetTcpBinding`.

Теперь должно быть все готово к запуску клиентского приложения. При условии, что хост все еще функционирует в фоновом режиме, появится возможность перемещения данных между сборками по протоколу TCP.

Исходный код. Конфигурационные файлы проекта `MagicEightBallTCP` доступны в подкаталоге `Chapter_25`.

Упрощение конфигурационных настроек

При проработке первого примера в настоящей главе вы могли заметить, что логика конфигурации хостинга довольно многословна. Например, файл `*.config` хоста (для исходной базовой привязки HTTP) должен определять один элемент `<endpoint>` для службы, еще один элемент `<endpoint>` для MEX, элемент `<baseAddresses>` (формально необязательный) для сокращения избыточных URI и затем раздел `<behaviors>` для указания характеристик обмена метаданными во время выполнения.

По правде говоря, изучение правил написания файлов `*.config` при построении служб WCF может оказаться трудной задачей. Чтобы еще более усложнить положение дел, порядочное число служб WCF склонно требовать те же самые базовые настройки в конфигурационном файле хоста. Например, если создается новая служба WCF и новый хост, и нужно открыть доступ к этой службе, используя элемент `<basicHttpBinding>` с поддержкой MEX, то необходимое содержимое файла `*.config` будет выглядеть практически идентичным созданному ранее.

К счастью, начиная с выпуска .NET 4.0, инфраструктура Windows Communication Foundation включает ряд упрощений, в числе которых стандартные настройки (и другие сокращения), намного облегчающие процесс построения конфигурации хоста.

Использование стандартных конечных точек

До появления поддержки стандартных конечных точек, когда вызывался метод `Open()` на объекте `ServiceHost`, а в конфигурационном файле не было определено ни одного элемента `<endpoint>`, исполняющая среда генерировала исключение. Аналогичный результат получался при вызове метода `AddServiceEndpoint()` в коде для указания конечной точки. Однако, начиная с версии .NET 4.5, каждая служба WCF автоматически получает *стандартные конечные точки*, которые фиксируют общепринятые детали конфигурации для каждого поддерживаемого протокола.

После открытия файла `machine.config` для .NET 4.5 в нем обнаружится новый элемент по имени `<protocolMapping>`. Этот элемент документирует привязки WCF, которые будут применяться по умолчанию, если ни одной привязки не указано:

```
<system.serviceModel>
  ...
  <protocolMapping>
    <add scheme = "http" binding="basicHttpBinding"/>
    <add scheme = "net.tcp" binding="netTcpBinding"/>
    <add scheme = "net.pipe" binding="netNamedPipeBinding"/>
    <add scheme = "net.msmq" binding="netMsmqBinding"/>
  </protocolMapping>
  ...
</system.serviceModel>
```

Для использования таких стандартных привязок потребуется только указать базовые адреса в конфигурационном файле хоста. Чтобы увидеть это в действии, откроем основанный на HTTP проект `MagicEightBallServiceHost` в Visual Studio. Модифицируем файл `*.config` хоста, полностью удалив элемент `<endpoint>` для службы WCF и все данные, связанные с MEX. Теперь конфигурационный файл должен выглядеть примерно так:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
```

```
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Поскольку в <baseAddress> указан допустимый базовый адрес HTTP, хост автоматически будет применять привязку basicHttpBinding. Запустив хост снова, можно увидеть тот же самый вывод данных ABC:

```
***** Console Based WCF Host *****
***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
**********************
The service is ready.
Press the Enter key to terminate service.
```

Здесь пока еще не включены службы MEX, но это вскоре будет сделано с использованием другого упрощения, которое называется *конфигурациями стандартного поведения*. Тем не менее, давайте сначала посмотрим, как открывать доступ к одиночной службе WCF с множеством привязок.

Открытие доступа к одиночной службе WCF, использующей множество привязок

Со времен своего первого выпуска инфраструктура WCF позволяет одному хосту открывать доступ к службе WCF с несколькими конечными точками. Например, открыть доступ к службе MagicEightBallService, применяющей привязки HTTP, TCP и именованных каналов, можно за счет добавления новых конечных точек в конфигурационный файл. После перезапуска хоста весь необходимый связующий механизм создается автоматически.

Это является огромным преимуществом по многим причинам. До появления WCF открывать доступ к одной службе с множеством привязок было трудно, т.к. каждый тип привязки (например, HTTP и TCP) имел собственную модель программирования.

Однако возможность разрешения вызывающему коду выбирать наиболее подходящую привязку чрезвычайно удобна. Внутренние вызывающие компоненты могут отдавать предпочтение привязкам TCP, внешние клиенты (находящиеся за брандмауэром компании) — использовать для доступа HTTP, в то время как клиенты на той же самой машине — применять именованный канал.

Чтобы сделать это в версиях, предшествующих .NET 4.5, внутри конфигурационного файла хоста требовалось вручную определять множество элементов <endpoint>. Также для каждого протокола необходимо было определять множество элементов <baseAddress>. Тем не менее, теперь можно просто подготовить следующий конфигурационный файл:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
```

```
        <host>
          <baseAddresses>
           <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
           <add baseAddress =
                "net.tcp://localhost:8099/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Скомпилировав проект (для обновления развернутого файла `*.config`) и перезапустив хост, можно будет увидеть следующие данные конечной точки:

```
***** Console Based WCF Host *****
***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
Address: net.tcp://localhost:8099/MagicEightBallService
Binding: NetTcpBinding
Contract: IEightBall
**********************
The service is ready.
Press the Enter key to terminate service.
```

Теперь, когда служба WCF достижима из двух конечных точек, возникает вопрос: каким образом клиент может производить выбор между ними? При генерации прокси клиентской стороны инструмент, запускаемый выбором пункта меню Add Service Reference, назначит каждой доступной конечной точке строковое имя в файле `*.config` клиентской стороны. В коде можно передавать корректное строковое имя конструктору прокси и иметь уверенность в том, что будет выбрана правильная привязка. Однако прежде чем делать это, потребуется переустановить MEX для модифицированного конфигурационного файла хоста и научиться настраивать параметры стандартной привязки.

Изменение параметров привязки WCF

В случае указания ABC службы в коде C# (что будет осуществляться далее в главе) способ изменения стандартных параметров привязки WCF вполне очевиден: нужно просто модифицировать значения свойств объекта. Например, если необходимо использовать привязку `BasicHttpBinding`, но изменить параметры таймаута, то можно написать такой код:

```
void ConfigureBindingInCode()
{
  BasicHttpBinding binding = new BasicHttpBinding();
  binding.OpenTimeout = TimeSpan.FromSeconds(30);
  ...
}
```

Параметры привязки всегда допускается конфигурировать в декларативной манере. Например, версия платформы .NET 3.5 позволяет строить конфигурационный файл хоста, в котором изменяется свойство `OpenTimeout` класса `BasicHttpBinding`, как показано ниже:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name = "myCustomHttpBinding"
                 openTimeout = "00:00:30" />
      </basicHttpBinding>
    </bindings>
    <services>
      <service name = "WcfMathService.MyCalc">
        <endpoint address = "http://localhost:8080/MyCalc"
                  binding = "basicHttpBinding"
                  bindingConfiguration = "myCustomHttpBinding"
                  contract = "WcfMathService.IBasicMath" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Здесь мы имеем конфигурационный файл для службы по имени `WcfMathService.MyCalc`, которая поддерживает единственный интерфейс `IBasicMath`. Обратите внимание, что раздел `<bindings>` позволяет определить именованный элемент `<binding>`, который изменяет параметры для заданной привязки. Внутри элемента `<endpoint>` службы можно подключать специфические параметры с применением атрибута `bindingConfiguration`.

Такой вид конфигурации хостинга по-прежнему работает ожидаемым образом; тем не менее, если задействуется стандартная конечная точка, то подключить `<binding>` к `<endpoint>` не удастся! К счастью, параметрами стандартной конечной точки можно управлять, просто опуская атрибут `name` элемента `<binding>`. Например, в следующем фрагменте кода изменяются некоторые свойства стандартных объектов `BasicHttpBinding` и `NetTcpBinding`, используемые на заднем плане:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress =
                 "http://localhost:8080/MagicEightBallService"/>
            <add baseAddress =
                 "net.tcp://localhost:8099/MagicEightBallService"/>
            </baseAddresses>
        </host>
      </service>
    </services>
    <bindings>
      <basicHttpBinding>
        <binding openTimeout = "00:00:30" />
      </basicHttpBinding>
      <netTcpBinding>
        <binding closeTimeout = "00:00:15" />
      </netTcpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

Использование конфигурации стандартного поведения MEX

Инструмент генерации прокси должен сначала обнаружить композицию службы во время выполнения. В WCF такое обнаружение во время выполнения разрешается путем включения MEX. В большинстве конфигурационных файлов хоста MEX понадобится включать (по крайней мере, на протяжении разработки); к счастью, способ конфигурирования MEX изменяется редко, поэтому в .NET 4.5 и последующих версиях предлагается несколько удобных сокращений.

Наиболее полезным сокращением является готовая поддержка MEX. Вам не придется добавлять конечную точку MEX, определять именованную линию поведения служб MEX и затем подключать именованную привязку к службе (как это делалось в HTTP-версии `MagicEightBallServiceHost`); взамен теперь можно просто добавить следующий код:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
           <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
           <add baseAddress =
                "net.tcp://localhost:8099/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>

    <bindings>
      <basicHttpBinding>
        <binding openTimeout = "00:00:30" />
      </basicHttpBinding>
      <netTcpBinding>
        <binding closeTimeout = "00:00:15" />
      </netTcpBinding>
    </bindings>

    <behaviors>
      <serviceBehaviors>
        <behavior>
          <!-- Для получения стандартного MEX
               не именуйте элемент <serviceMetadata> -->
          <serviceMetadata httpGetEnabled = "true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Трюк заключается в том, что элемент `<serviceMetadata>` больше не имеет атрибута `name` (кроме того, элемент `<service>` больше не нуждается в атрибуте `behaviorConfiguration`). После указанных корректировок во время выполнения становится доступной поддержка MEX. Чтобы проверить это, можно запустить хост (после компиляции с целью обновления конфигурационного файла) и ввести в браузере такой URL:

```
http://localhost:8080/MagicEightBallService
```

Затем можно щелкнуть на ссылке `wsdl` в верхней части веб-страницы и просмотреть описание WSDL службы (см. рис. 25.6). Обратите внимание, что в выводе внутри

окна консоли хоста отсутствуют данные о конечной точке MEX, т.к. конечная точка для `IMetadataExchange` в конфигурационном файле явно не определялась. Однако службы MEX включены, и можно приступать к построению клиентских прокси.

Обновление клиентского прокси и выбор привязки

Предполагая, что обновленный хост скомпилирован и функционирует в фоновом режиме, теперь необходимо открыть клиентское приложение и обновить текущую ссылку на службу. Начнем с открытия папки Service References (Ссылки на службы) в окне Solution Explorer. Далее щелкнем правой кнопкой мыши на элементе `ServiceReference` и выберем в контекстном меню пункт Update Service Reference (Обновить ссылку на службу), как показано на рис. 25.8.

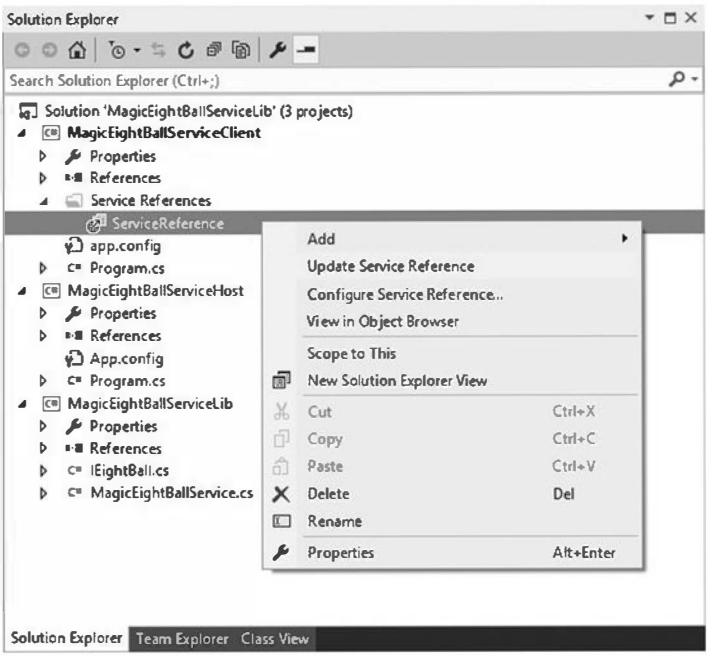


Рис. 25.8. Обновление прокси и файла `*.config` клиентской стороны

После этого в файле `*.config` клиентской стороны появятся на выбор две привязки: одна для HTTP, а другая для TCP. Каждой привязке назначено подходящее имя. Ниже приведен частичный листинг обновленного конфигурационного файла:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name = "BasicHttpBinding_IEightBall" ... />
      </basicHttpBinding>
      <netTcpBinding>
        <binding name = "NetTcpBinding_IEightBall" ... />
      </netTcpBinding>
    </bindings>
    ...
  </system.serviceModel>
</configuration>
```

Клиент может применять эти имена при создании прокси-объекта для выбора желаемой привязки. Таким образом, если клиент предпочитает использовать TCP, то код C# клиентской стороны можно модифицировать следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Ask the Magic 8 Ball *****\n");
  using (EightBallClient ball = new EightBallClient("NetTcpBinding_IEightBall"))
  {
    ...
  }
  Console.ReadLine();
}
```

Если же клиент вместо этого будет применять привязку HTTP, то можно написать так:

```
using (EightBallClient ball = new EightBallClient("BasicHttpBinding_IEightBall"))
{
  ...
}
```

На этом текущий пример, в котором демонстрировалось несколько полезных сокращений, завершен. Такие средства упрощают написание конфигурационных файлов хостов. Далее будет показано, как использовать шаблон проекта WCF Service Library (Библиотека служб WCF).

Исходный код. Проект `MagicEightBallServiceHTTPDefaultBindings` доступен в подкаталоге `Chapter_25`.

Использование шаблона проекта WCF Service Library

Прежде чем строить более причудливую службу WCF, которая взаимодействует с созданной в главе 21 базой данных `AutoLot`, в следующем примере будут проиллюстрированы важные темы, в том числе преимущества шаблона проекта WCF Service Library, приложение WCF Test Client, редактор конфигурации WCF, размещение служб WCF внутри Windows-службы и асинхронные клиентские вызовы. Чтобы сосредоточить все внимание на новых концепциях, эта служба WCF будет намеренно сохранена простой.

Построение простой математической службы

Для начала создадим новый проект WCF Service Library по имени `MathServiceLibrary`, выбрав корректный вариант в узле WCF внутри диалогового окна New Project (см. рис. 25.2). Затем изменим имя начального файла `IService1.cs` на `IBasicMath.cs`. После этого *удалим* весь код из пространства имен `MathServiceLibrary` и заменим его следующим кодом:

```
[ServiceContract(Namespace="http://MyCompany.com")]
public interface IBasicMath
{
  [OperationContract]
  int Add(int x, int y);
}
```

Далее переименуем файл Service1.cs в MathService.cs, удалим весь код из пространства имен MathServiceLibrary и реализуем контракт службы, как показано ниже:

```
public class MathService : IBasicMath
{
  public int Add(int x, int y)
  {
    // Для эмуляции длительного запроса.
    System.Threading.Thread.Sleep(5000);
    return x + y;
  }
}
```

Обратите внимание, что этот файл *.config уже включает поддержку MEX; по умолчанию конечная точка службы применяет протокол basicHttpBinding.

Тестирование службы WCF с помощью WcfTestClient.exe

Одно из преимуществ использования проекта WCF Service Library связано с тем, что при отладке или запуске библиотеки он будет читать настройки из файла *.config и применять их для загрузки приложения WCF Test Client (WcfTestClient.exe). Упомянутое приложение с графическим пользовательским интерфейсом позволяет протестировать каждый член интерфейса службы по мере ее построения; это значит, что вручную строить хост/клиент просто для целей тестирования, как это делалось ранее, не придется.

На рис. 25.9 показана тестовая среда для службы MathService. Обратите внимание, что двойной щелчок на методе интерфейса позволяет указать входные параметры и вызвать метод.

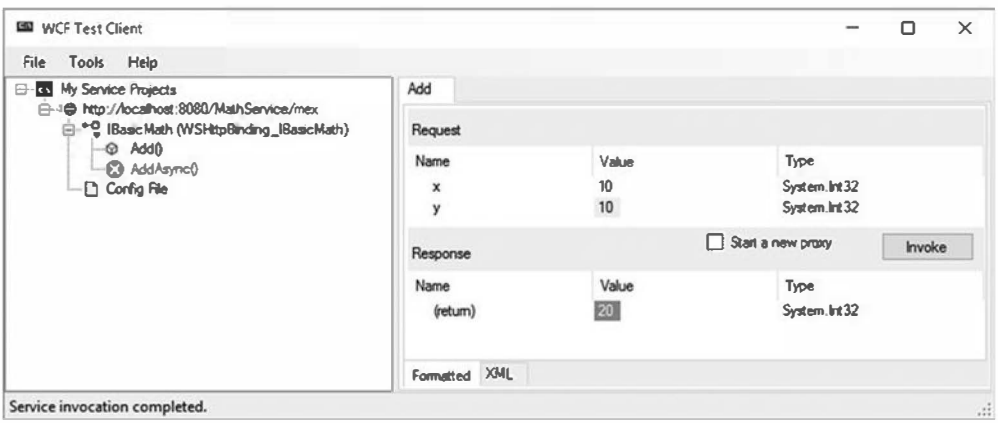


Рис. 25.9. Тестирование службы WCF с использованием WcfTestClient.exe

Эта утилита работает сразу же после создания проекта WCF Service Library; тем не менее, имейте в виду, что данный инструмент можно применять для тестирования любой службы WCF, запустив его в командной строке и указав конечную точку MEX. Например, если запустить приложение MagicEightBallServiceHost.exe, то в окне командной строки разработчика можно ввести следующую команду:

```
wcftestclient http://localhost:8080/MagicEightBallService
```

Затем в похожей манере можно вызвать метод ObtainAnswerToQuestion().

Изменение конфигурационных файлов с использованием SvcConfigEditor.exe

Еще одно преимущество применения проекта WCF Service Library заключается в том, что щелчком правой кнопкой мыши на файле App.config в окне Solution Explorer можно активизировать графический редактор конфигурирования службы (Service Configuration Editor), SvcConfigEditor.exe (рис. 25.10). Тот же самый прием может использоваться в клиентском приложении, которое ссылается на службу WCF.

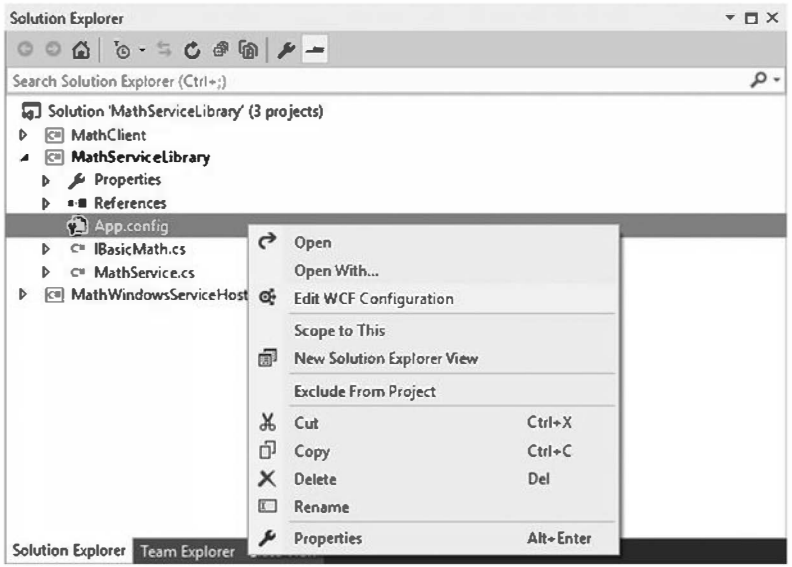


Рис. 25.10. Запуск графического редактора файлов *.config

После запуска этого инструмента можно изменять данные в формате XML с применением дружественного пользовательского интерфейса. Использование инструментов вроде этого для сопровождения файлов *.config дает много преимуществ. Первое (и самое главное): появляется гарантия того, что сгенерированная разметка соответствует ожидаемому формату и не содержит опечаток. Второе: это замечательный способ *увидеть* допустимые значения, которые могут быть присвоены каждому атрибуту. И третье: больше не понадобится вручную вводить данные XML.

На рис. 25.11 показан внешний вид окна редактора Service Configuration Editor. По правде говоря, описание всех интересных средств SvcConfigEditor.exe заняло бы целую главу. Найдите время для исследования этого инструмента; помните о возможности доступа к детальной справочной системе по нажатию <F1>.

На заметку! Утилита SvcConfigEditor.exe позволяет редактировать (или создавать) конфигурационные файлы, даже если не был выбран начальный проект WCF Service Library. Необходимо просто запустить этот инструмент в окне командной строки разработчика и с помощью пункта меню File⇨Open (Файл⇨Открыть) загрузить существующий файл *.config для редактирования.

Конфигурировать службу MathService больше не понадобится, поэтому можно перейти к задаче построения специального хоста.

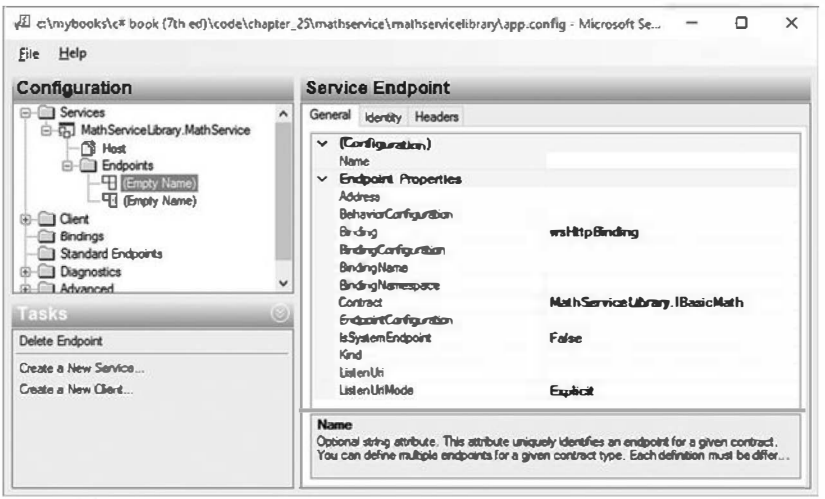


Рис. 25.11. Работа с редактором Service Configuration Editor

Хостинг службы WCF внутри Windows-службы

Хостинг службы WCF в консольном приложении (или внутри настольного приложения с графическим пользовательским интерфейсом) — не идеальный вариант для сервера производственного уровня, поскольку хост всегда должен оставаться функционирующим в фоновом режиме и готовым к обслуживанию клиентов. Даже если свернуть приложение-хост в панели задач Windows, то все равно очень легко случайно прекратить его работу и тем самым разорвать подключения клиентских приложений.

На заметку! Хотя и верно то, что настольное приложение Windows *не обязано* отображать главное окно, типичная программа `*.exe` требует взаимодействия с пользователем для ее загрузки и запуска. Однако службу Windows (описанную далее) можно сконфигурировать на запуск даже при отсутствии пользователей, вошедших в систему рабочей станции.

Если вы строите внутреннее приложение WCF, то еще одной альтернативой для хостинга библиотеки службы WCF будет ее размещение внутри Windows-службы. Преимущество такого решения связано с тем, что Windows-служба может быть сконфигурирована для автоматического запуска при загрузке системы на целевой машине. Другое преимущество заключается в том, что Windows-служба выполняется невидимо в фоновом режиме (в отличие от консольного приложения) и не требует участия пользователя (к тому же на машине хостинга не требуется наличие установленного сервера IIS).

Давайте посмотрим, как строить такой хост. Начнем с создания нового проекта Windows Service (Служба Windows) по имени `MathWindowsServiceHost` (рис. 25.12). В окне Solution Explorer переименуем начальный файл `Service1.cs` на `MathService.cs`.

Указание ABC в коде

Теперь предположим, что установлены ссылки на сборки `MathServiceLibrary.dll` и `System.ServiceModel.dll`. Все, что осталось сделать — применить тип `ServiceHost` внутри методов `OnStart()` и `OnStop()` типа Windows-службы. Откроем файл кода для класса хоста службы (щелкнув правой кнопкой мыши на поверхности визуального конструктора и выбрав в контекстном меню пункт View Code (Просмотреть код)) и добавим следующий код:

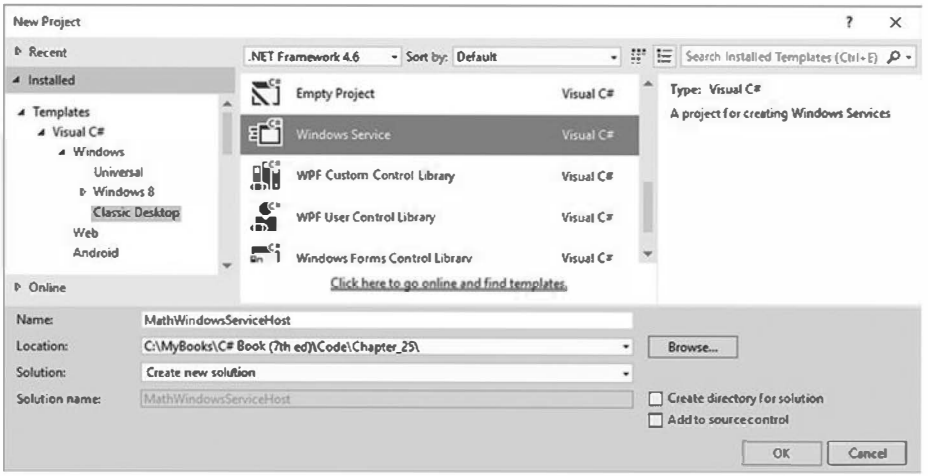


Рис. 25.12. Создание Windows-службы для хостинга службы WCF

```
// Должны быть импортированы следующие пространства имен:
using MathServiceLibrary;
using System.ServiceModel;

namespace MathWindowsServiceHost
{
  public partial class MathWinService: ServiceBase
  {
    // Переменная-член типа ServiceHost.
    private ServiceHost myHost;

    public MathWinService()
    {
      InitializeComponent();
    }
    protected override void OnStart(string[] args)
    {
      // Проверить для подстраховки.
      if (myHost != null)
      {
        myHost.Close();
        myHost = null;
      }
      // Создать хост.
      myHost = new ServiceHost(typeof(MathService));
      // Указать ABC в коде.
      Uri address = new Uri("http://localhost:8080/MathServiceLibrary");
      WSHttpBinding binding = new WSHttpBinding();
      Type contract = typeof(IBasicMath);
      // Добавить эту конечную точку.
      myHost.AddServiceEndpoint(contract, binding, address);
      // Открыть хост.
      myHost.Open();
    }
    protected override void OnStop()
    {
```

```
      // Остановить хост.
      if(myHost != null)
        myHost.Close();
    }
  }
}
```

Хотя при построении хоста для службы WCF на основе Windows-службы ничто не препятствует использованию конфигурационного файла, здесь (для разнообразия) вместо файла `*.config` конечная точка устанавливается программно с применением классов `Uri`, `WSHttpBinding` и `Type`. После создания всех аспектов ABC хост программно информируется о них с помощью метода `AddServiceEndpoint()`.

Если исполняющей среде нужно указать о том, что необходим доступ ко всем привязкам стандартных конечных точек, которые хранятся в конфигурационном файле `machine.config` платформы .NET 4.6, то программную логику можно упростить, указывая базовые адреса при вызове конструктора `ServiceHost`. В таком случае не придется задавать ABC в коде вручную или вызывать метод `AddServiceEndPoint()`; взамен необходимо просто вызвать `AddDefaultEndPoints()`. Взгляните на следующее изменение кода:

```
protected override void OnStart(string[] args)
{
  if (myHost != null)
  {
    myHost.Close();
  }
  // Создать хост и указать URL для привязки HTTP.
  myHost = new ServiceHost(typeof(MathService),
           new Uri("http://localhost:8080/MathServiceLibrary"));
  // Выбрать стандартные конечные точки.
  myHost.AddDefaultEndpoints();
  // Открыть хост.
  myHost.Open();
}
```

Включение MEX

Несмотря на то что включить MEX можно программно, мы сделаем это в конфигурационном файле. Модифицируем файл `App.config` в проекте Windows-службы, поместив в него следующие стандартные настройки MEX:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MathServiceLibrary.MathService">
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior>
        <serviceMetadata httpGetEnabled = "true"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
  </system.serviceModel>
</configuration>
```

Создание программы установки для Windows-службы

Чтобы зарегистрировать Windows-службу в операционной системе, к проекту понадобится добавить программу установки, которая будет содержать необходимый код для регистрации службы. Для этого щелкнем правой кнопкой мыши на поверхности визуального конструктора Windows-службы и выберем в контекстном меню пункт Add Installer (Добавить программу установки), как показано на рис. 25.13.

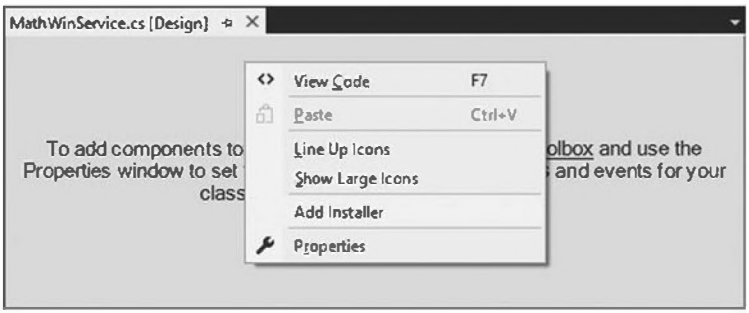


Рис. 25.13. Добавление программы установки для Windows-службы

В результате на поверхность визуального конструктора добавятся два новых компонента. Первый компонент (именуемый по умолчанию `serviceProcessInstaller1`) представляет элемент, который может установить новую Windows-службу на целевой машине. Выберем этот элемент в визуальном конструкторе и в окне Properties (Свойства) установим свойство `Account` в `LocalSystem` (рис. 25.14).

Второй компонент (под названием `serviceInstaller`) представляет тип, который будет устанавливать конкретную Windows-службу. В окне Properties изменим значение свойства `ServiceName` на `MathService`, установим свойство `StartType` в `Automatic` и укажем дружественное описание зарегистрированной Windows-службы в свойстве `Description` (рис. 25.15).

Теперь приложение можно компилировать.

Установка Windows-службы

Служба Windows может быть установлена на машине-хосте с помощью традиционной программы установки (такой как установщик `*.msi`) либо инструмента командной строки `installutil.exe`.

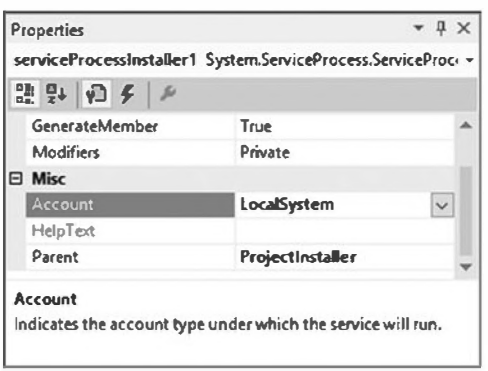


Рис. 25.14. Служба Windows должна запускаться от имени учетной записи локальной системы

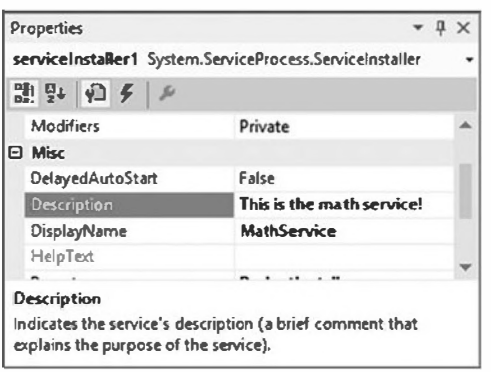


Рис. 25.15. Конфигурирование деталей, связанных с программой установки

На заметку! Для установки Windows-службы с использованием `installutil.exe` окно командной строки разработчика должно быть запущено с административными привилегиями. Чтобы сделать это, нужно щелкнуть правой кнопкой мыши на значке Developer Command Prompt (Командная строка разработчика) и выбрать в контекстном меню пункт Запуск от имени администратора.

В окне командной строки перейдем в папку `\bin\Debug` проекта `MathWindows ServiceHost` и введем следующую команду (требуются административные привилегии):

```
installutil MathWindowsServiceHost.exe
```

Предполагая, что установка прошла успешно, можно щелкнуть на значке Services (Службы) в папке Administrative Tools (Администрирование) панели управления Windows. В списке служб, упорядоченном по алфавиту, должно присутствовать дружественное имя службы `MathService` (рис. 25.16). Если служба еще не функционирует, то ее необходимо запустить посредством ссылки Start (Запустить).

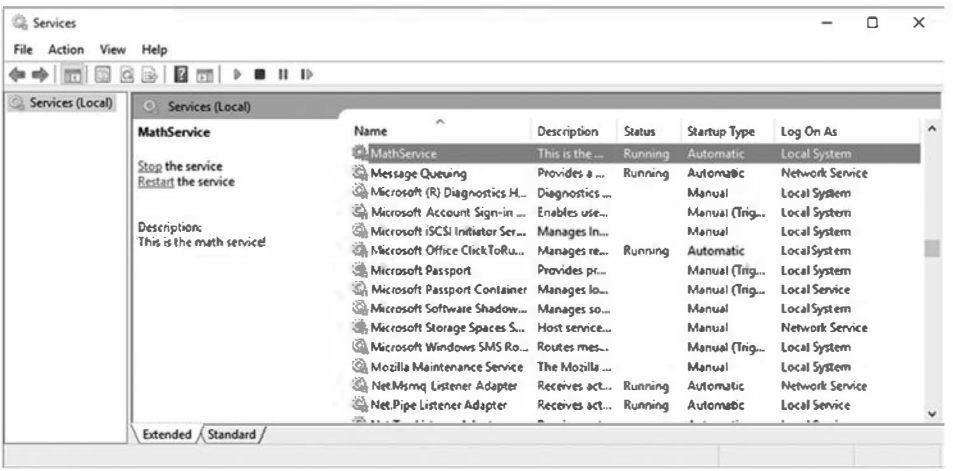


Рис. 25.16. Служба Windows, в которой размещается наша служба WCF

Теперь, когда служба установлена и работает, остается построить клиентское приложение для ее потребления.

Исходный код. Проект `MathWindowsServiceHost` доступен в подкаталоге `Chapter_25`.

Асинхронный вызов службы из клиента

Создадим новый проект консольного приложения по имени `MathClient` и установим ссылку на выполняющуюся службу WCF (в настоящий момент размещенную в Windows-службе, функционирующей в фоновом режиме), выбрав пункт меню Project⇨Add Service Reference (Проект⇨Добавить ссылку на службу) в Visual Studio (в поле Address (Адрес) понадобится ввести URL, который должен выглядеть как `http://localhost:8080/MathServiceLibrary`). Пока не следует щелкать на кнопке OK. В нижнем левом углу диалогового окна Add Service Reference (Добавление ссылки на службу) находится кнопка Advanced (Дополнительно), как показано на рис. 25.17. Щелкнем на кнопке Advanced, чтобы просмотреть дополнительные настройки конфигурации прокси в диалоговом окне Service Reference Settings (Настройки ссылки на службу), представленном на рис. 25.18.

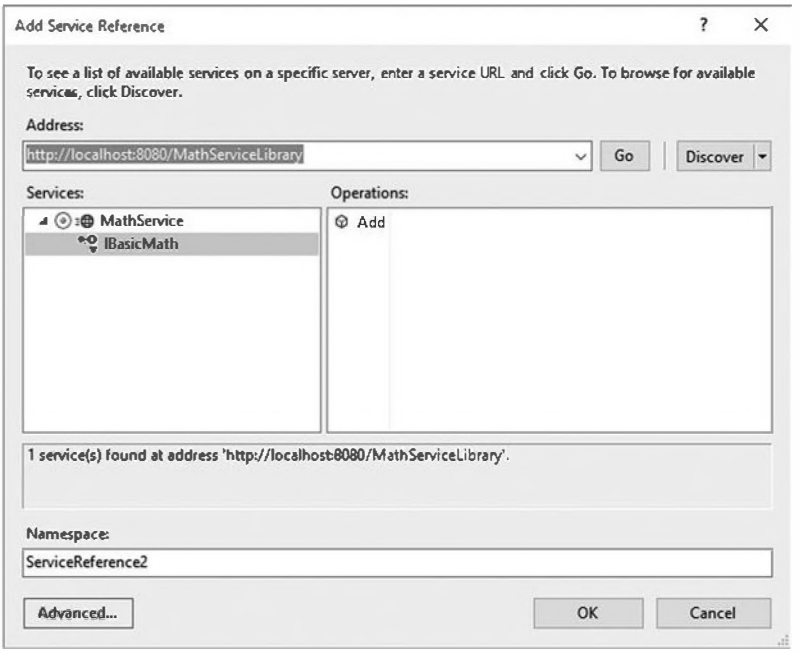


Рис. 25.17. Добавление ссылки на службу `MathService` и подготовка к конфигурированию расширенных настроек

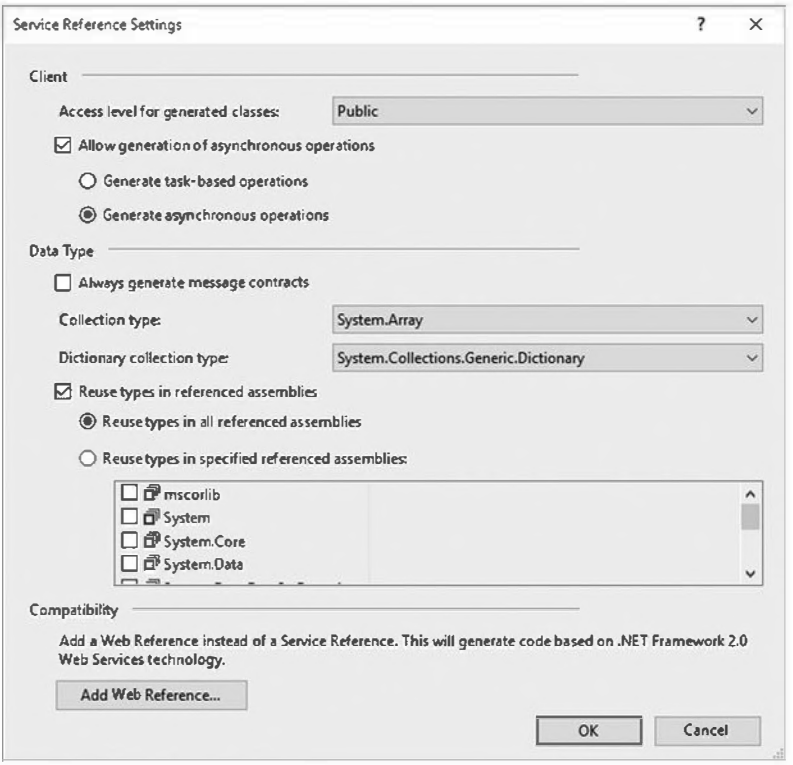


Рис. 25.18. Дополнительные настройки конфигурации прокси клиентской стороны

В этом диалоговом окне можно сгенерировать код, который позволит вызывать удаленные методы в асинхронной манере, если выбран переключатель Generate asynchronous operations (Генерировать асинхронные операции). Выберем упомянутый переключатель.

В данный момент код прокси содержит дополнительные методы, которые позволяют обращаться к каждому члену контракта службы с применением ожидаемого шаблона асинхронных вызовов Begin/End, описанного в главе 19. Ниже приведена простая реализация, в которой вместо строго типизированного делегата AsyncCallback используется лямбда-выражение:

```
using System;
using MathClient.ServiceReference1;
...
namespace MathClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** The Async Math Client *****\n");
      using (BasicMathClient proxy = new BasicMathClient())
      {
        proxy.Open();
        //Суммировать числа в асинхронной манере с применением лямбда-выражения.
        IAsyncResult result = proxy.BeginAdd(2, 3,
          ar =>
          {
            Console.WriteLine("2 + 3 = {0}", proxy.EndAdd(ar));
          },
          null);
        while (!result.IsCompleted)
        {
          Thread.Sleep(200);
          Console.WriteLine("Client working...");
        }
      }
      Console.ReadLine();
    }
  }
}
```

Исходный код. Проект MathClient доступен в подкаталоге Chapter_25.

Проектирование контрактов данных WCF

В последнем примере главы демонстрируется конструирование *контрактов данных* WCF. В ранее созданных службах WCF были определены очень простые методы, оперирующие примитивными типами данных CLR. Когда используется любой тип привязки HTTP (скажем, basicHttpBinding и wsHttpBinding), входные и выходные простые типы данных автоматически форматируются в виде элементов XML. Кстати говоря, если применяется привязка на основе TCP (такая как netTcpBinding), то параметры и возвращаемые значения простых типов данных передаются в компактном двоичном формате.

На заметку! Исполняющая среда WCF также будет автоматически кодировать любой тип, помеченный атрибутом `[Serializable]`; тем не менее, это не является предпочтительным способом определения контрактов WCF и предназначено только для обратной совместимости.

Однако при определении контрактов служб, которые используют специальные классы в качестве параметров или возвращаемых значений, эти данные рекомендуется моделировать с применением контрактов данных WCF. Выражаясь просто, контракт данных представляет собой тип, декорированный атрибутом `[DataContract]`. Подобным же образом каждое поле, которое планируется использовать как часть предполагаемого контракта, должно быть помечено атрибутом `[DataMember]`.

На заметку! В ранних версиях платформы .NET применение атрибутов `[DataContract]` и `[DataMember]` было обязательным для обеспечения корректного представления специальных типов данных. Это требование в Microsoft было ослаблено; формально вы не обязаны использовать указанные атрибуты в специальных типах данных; тем не менее, в .NET такой прием считается рекомендуемым подходом.

Использование веб-ориентированного шаблона проекта WCF Service

Следующая служба WCF позволит внешним вызывающим компонентам взаимодействовать с базой данных `AutoLot`, созданной в главе 21. Более того, эта финальная служба WCF будет создана с применением веб-ориентированного шаблона проекта WCF Service и размещена в IIS.

Для начала запустим Visual Studio (с правами администратора) и выберем пункт меню File⇨New⇨Web Site (Файл⇨Создать⇨Веб-сайт). Укажем тип проекта WCF Service (Служба WCF) и удостоверимся в том, что в раскрывающемся списке Web Location (Веб-местоположение) выбран вариант HTTP (это приведет к установке службы в IIS). Откроем доступ к службе из такого URI:

```
http://localhost/AutoLotWCFService
```

На рис. 25.19 показаны настройки для проекта.

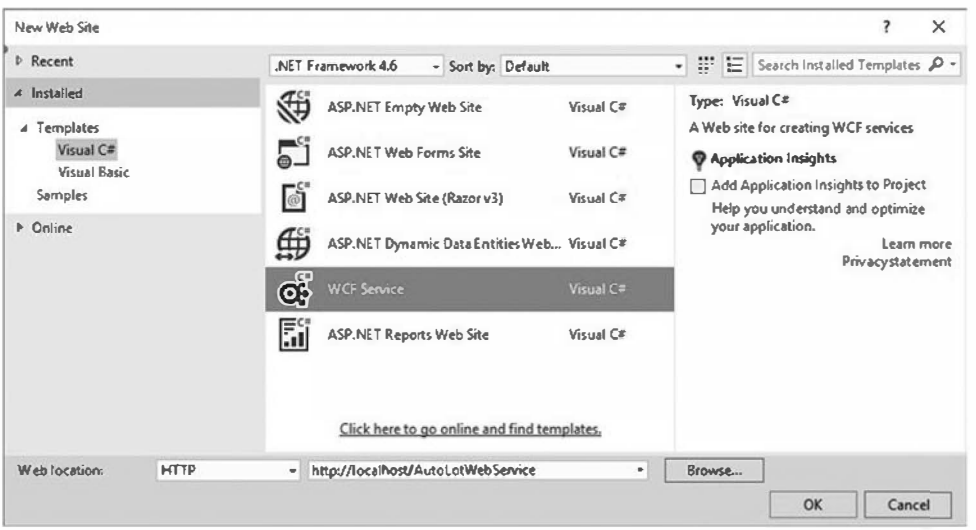


Рис. 25.19. Создание веб-ориентированной службы WCF

После этого установим ссылку на сборку AutoLotDAL.dll, созданную в главе 21 (используя пункт меню Website⇨Add Reference (Веб-сайт⇨Добавить ссылку)). Будет предоставлен начальный код (в папке Add_Code), который необходимо удалить. Первым делом переименуем исходный файл IService.cs в IAutoLotService.cs и внутри переименованного файла определим контракт службы, как показано ниже:

```
[ServiceContract]
public interface IAutoLotService
{
  [OperationContract]
  void InsertCar(int id, string make, string color, string petname);

  [OperationContract]
  void InsertCar(InventoryRecord car);

  [OperationContract]
  InventoryRecord[] GetInventory();
}
```

В этом интерфейсе определены три метода, один из которых возвращает массив объектов типа InventoryRecord (пока еще не созданного). Вы можете вспомнить, что метод GetInventory() класса InventoryDAL просто возвращает объект DataTable, из-за чего возникает вопрос: почему бы методу GetInventory() создаваемой службы не делать то же самое?

Хотя метод службы WCF мог бы возвращать тип DataTable, инфраструктура WCF обязана следовать принципам SOA, в том числе программированию на основе контрактов, а не реализаций.

Таким образом, вместо возвращения внешнему вызывающему компоненту специфичного для .NET типа DataTable мы будем возвращать специальный контракт данных (InventoryRecord), корректно выраженный внутри документа WSDL в независимой манере.

Также обратите внимание, что в показанном ранее интерфейсе определен перегруженный метод по имени InsertCar(). Его первая версия принимает четыре входных параметра, а вторая — один параметр типа InventoryRecord. Контракт данных InventoryRecord может быть определен следующим образом:

```
[DataContract]
public class InventoryRecord
{
  [DataMember]
  public int ID;

  [DataMember]
  public string Make;

  [DataMember]
  public string Color;

  [DataMember]
  public string PetName;
}
```

Если реализовать интерфейс IAutoLotService в таком виде, а затем построить хост и попытаться вызвать эти методы на стороне клиента, то возникнет исключение времени выполнения. Причина в том, что одно из требований описания WSDL предусматривает назначение *уникального имени* каждому методу, доступ к которому открыт из заданной конечной точки. Следовательно, в то время как перегрузка методов замечательно работает в контексте языка C#, текущие спецификации веб-служб не разрешают существование двух методов с одним и тем же именем InsertCar().

К счастью, атрибут [OperationContract] поддерживает свойство Name, которое позволяет указать, каким образом метод C# будет представлен внутри описания WSDL. С учетом этого модифицируем вторую версию InsertCar(), как показано ниже:

```
public interface IAutoLotService
{
  ...
  [OperationContract(Name = "InsertCarWithDetails")]
  void InsertCar(InventoryRecord car);
}
```

Реализация контракта службы

Теперь переименуем файл Service.cs в AutoLotService.cs. Тип AutoLotService реализует интерфейс IAutoLotService (в этот файл кода понадобится импортировать пространства имен AutoLotConnectedLayer и System.Data, а также при необходимости обновить строку подключения):

```
using AutoLotDAL.ConnectedLayer;
using System.Data;

public class AutoLotService : IAutoLotService
{
  private const string ConnString =
    @"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot"+
     ";Integrated Security=True";

  public void InsertCar(int id, string make, string color, string petname)
  {
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    d.InsertAuto(id, color, make, petname);
    d.CloseConnection();
  }

  public void InsertCar(InventoryRecord car)
  {
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    d.InsertAuto(car.ID, car.Color, car.Make, car.PetName);
    d.CloseConnection();
  }

  public InventoryRecord[] GetInventory()
  {
    // Сначала получить DataTable из базы данных.
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    DataTable dt = d.GetAllInventoryAsDataTable();
    d.CloseConnection();

    // Теперь создать List<T> для хранения записей.
    List<InventoryRecord> records = new List<InventoryRecord>();

    //Скопировать содержимое таблицы данных в список List<> специальных контрактов.
    DataTableReader reader = dt.CreateDataReader();
    while (reader.Read())
    {
      InventoryRecord r = new InventoryRecord();
      r.ID = (int)reader["CarID"];
```

```
      r.Color = ((string)reader["Color"]);
      r.Make = ((string)reader["Make"]);
      r.PetName = ((string)reader["PetName"]);
      records.Add(r);
    }
    // Трансформировать List<T> в массив элементов типа InventoryRecord.
    return (InventoryRecord[])records.ToArray();
  }
}
```

В приведенном выше коде нет ничего особенного. Ради простоты мы жестко закодировали значение строки подключения (которая может потребовать корректировки для соответствия текущим настройкам машины) вместо сохранения ее в файле `Web.config`. Учитывая, что библиотека доступа к данным выполняет всю реальную работу по взаимодействию с базой данных `AutoLot`, потребуется лишь передать входные параметры методу `InsertAuto()` класса `InventoryDAL`. Единственное, что здесь представляет интерес — это действие по отображению значений объекта `DataTable` на обобщенный список элементов типа `InventoryRecord` (с применением `DataTableReader`) и затем трансформация `List<T>` в массив объектов типа `InventoryRecord`.

Роль файла *.svc

При создании веб-ориентированной службы WCF вы обнаружите, что проект содержит специфичный файл с расширением `*.svc`. Этот конкретный файл необходим любой службе WCF, размещаемой в IIS; в нем описано имя и местоположение реализации службы внутри точки установки. Поскольку имена начального файла и типов WCF были изменены, потребуется модифицировать содержимое файла `Service.svc`:

```
<%@ ServiceHost Language="C#" Debug="true"
    Service="AutoLotService" CodeBehind="~/App_Code/AutoLotService.cs" %>
```

Содержимое файла Web.config

В файле `Web.config` службы WCF, созданной для HTTP, будет использоваться несколько упрощений WCF, исследованных ранее в главе. Как подробно объясняется далее в книге при рассмотрении ASP.NET, предназначение файла `Web.config` аналогично файлу `*.config` исполняемой сборки; однако он также управляет рядом настроек, специфических для веб. Например, обратите внимание, что службы MEX включены, и не нужно указывать специальный элемент `<endpoint>` вручную:

```
<configuration>
  ...
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <!-- Чтобы избежать раскрытия информации метаданных,
               установите следующее значение в false и удалите
               конечную точку метаданных перед развертыванием. -->
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
          <!-- Для получения деталей исключений при отказах в целях
               отладки установите следующее значение в true.
               Перед развертыванием установите его снова в false
               во избежание раскрытия информации об исключении. -->
          <serviceDebug includeExceptionDetailInFaults="false"/>
```

```
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"
                               multipleSiteBindingsEnabled="true" />
  </system.serviceModel>
  ...
</configuration>
```

Тестирование службы

Теперь для тестирования службы можно построить клиент любого вида, включая передачу конечной точки файла `*.svc` приложению `WcfTestClient.exe`:

```
WcfTestClient http://localhost/AutoLotWCFService/Service.svc
```

Если необходимо создать специальное клиентское приложение, то можно применить диалоговое окно Add Service Reference, как это делалось в примерах проектов `MagicEightBallServiceClient` и `MathClient` ранее в главе.

Исходный код. Проект `AutoLotService` доступен в подкаталоге `Chapter_25`.

На этом исследование API-интерфейса Windows Communication Foundation завершено. Разумеется, эта тема слишком обширна, чтобы ее можно было полностью раскрыть в одной ознакомительной главе, но благодаря изложенному здесь материалу можно продолжить изучение WCF самостоятельно. Дополнительные сведения об инфраструктуре WCF приведены в документации .NET Framework 4.6 SDK.

Резюме

В этой главе вы ознакомились с инфраструктурой Windows Communication Foundation (WCF), которая представляет основной API-интерфейс распределенного программирования на платформе .NET. Как здесь объяснялось, главной мотивацией, создания WCF было предоставление унифицированной объектной модели, которая открывает доступ к нескольким собранным воедино (ранее несвязанным) API-интерфейсам распределенных вычислений. Служба WCF представляется путем указания адресов, привязок и контрактов (обозначаемых аббревиатурой ABC).

Вы также узнали, что типичное приложение WCF предусматривает использование трех взаимосвязанных сборок. В первой сборке определяются контракты и типы службы, которые представляют ее функциональность. Эта сборка затем размещается в специальной исполняемой программе, в виртуальном каталоге IIS либо в Windows-службе. Наконец, клиентская сборка применяет сгенерированный файл кода, в котором определен тип прокси (и настройки внутри конфигурационного файла приложения) для взаимодействия с удаленным типом.

В главе также было показано, как использовать несколько инструментов программирования для WCF, такие как редактор `SvcConfigEditor.exe` (позволяющий модифицировать содержимое файлов `*.config`), приложение `WcfTestClient.exe` (для быстрого тестирования служб WCF) и разнообразные шаблоны проектов Visual Studio. Кроме того, вы узнали о ряде упрощений конфигурации, включая стандартные конечные точки и линии поведения.

ЧАСТЬ VII

Windows Presentation Foundation

В этой части

ГЛАВА 26

Введение в Windows Presentation Foundation и XAML

Когда была выпущена версия 1.0 платформы .NET, программисты, нуждающиеся в построении графических настольных приложений, использовали два API-интерфейса под названиями Windows Forms и GDI+, упакованные главным образом в сборки `System.Windows.Forms.dll` и `System.Drawing.dll`. Наряду с тем, что Windows Forms и GDI+ — великолепные API-интерфейсы для построения традиционных настольных графических пользовательских интерфейсов, начиная с версии .NET 3.0, Microsoft также поставляет альтернативный API-интерфейс для тех же целей, который называется Windows Presentation Foundation (WPF).

Эта вводная глава о WPF начинается с исследования мотивации, лежащей в основе новой инфраструктуры для построения графических пользовательских интерфейсов, что поможет увидеть отличия между моделями программирования Windows Forms/GDI+ и WPF. Затем мы рассмотрим разнообразные типы приложений WPF, поддерживаемые данным API-интерфейсом, а также выясним роль нескольких важных классов, включая `Application`, `Window`, `ContentControl`, `Control`, `UIElement` и `FrameworkElement`. Попутно вы научитесь перехватывать действия мыши и клавиатуры, определять данные на уровне приложения и решать другие распространенные задачи WPF, не применяя ничего кроме кода C#.

В настоящей главе будет представлена грамматика на основе XML, которая называется *расширяемым языком разметки приложений* (Extensible Application Markup Language — XAML). Вы изучите синтаксис и семантику XAML (в том числе синтаксис присоединяемых свойств, роль преобразователей типов и расширений разметки), а также узнаете, как генерировать, загружать и синтаксически анализировать XAML во время выполнения. Кроме того, будет показано, каким образом интегрировать данные XAML в кодовую базу WPF на C# (и связанные с этим преимущества).

Глава завершается исследованием визуальных конструкторов WPF, встроенных в Visual Studio. Будет построен собственный специальный редактор/анализатор XAML, который продемонстрирует способ манипулирования разметкой XAML во время выполнения для создания динамических пользовательских интерфейсов.

Мотивация, лежащая в основе WPF

На протяжении многих лет в Microsoft создавали многочисленные инструменты для построения графических пользовательских интерфейсов (для низкоуровневой разработки на C/C++/Windows API, VB6, MFC и т.д.), предназначенных для настольных приложений. Каждый инструмент предлагает кодовую базу для представления основных аспектов приложения с графическим пользовательским интерфейсом, включая главные окна, диалоговые окна, элементы управления, системы меню и другие базовые аспекты. После начального выпуска платформы .NET инфраструктура Windows Forms быстро стала предпочтительным подходом к разработке пользовательских интерфейсов благодаря своей простой, но очень мощной объектной модели.

Хотя с помощью Windows Forms было успешно создано множество полноценных настольных приложений, дело в том, что эта программная модель до некоторой степени *ассиметрична*. Попросту говоря, сборки `System.Windows.Forms.dll` и `System.Drawing.dll` не обеспечивают прямой поддержки для многих дополнительных технологий, требуемых при построении полнофункционального настольного приложения. Чтобы проиллюстрировать это утверждение, рассмотрим природу разработки графического пользовательского интерфейса до выпуска WPF (табл. 26.1).

Таблица 26.1. Решения, предшествующие WPF, для обеспечения желаемой функциональности

Желаемая функциональность	Технология
Построение окон с элементами управления	Windows Forms
Поддержка двухмерной графики	GDI+ (`System.Drawing.dll`)
Поддержка трехмерной графики	API-интерфейсы DirectX
Поддержка потокового видео	API-интерфейсы Windows Media Player
Поддержка документов нефиксированного формата	Программное манипулирование файлами PDF

Как видите, разработчик, использующий Windows Forms, вынужден заимствовать типы из нескольких несвязанных API-интерфейсов и объектных моделей. Несмотря на то что применение всех разрозненных API-интерфейсов синтаксически похоже (в конце концов, это просто код C#), каждая технология требует радикально иного мышления. Например, навыки, необходимые для создания трехмерной анимации с использованием DirectX, совершенно отличаются от тех, что нужны для привязки данных к экранной сетке. По правде говоря, программисту Windows Forms чрезвычайно трудно в равной степени овладеть природой каждого API-интерфейса.

Унификация несходных API-интерфейсов

Инфраструктура WPF (появившаяся в .NET 3.0) специально создавалась для объединения этих ранее несвязанных задач программирования в единственную унифицированную объектную модель. Таким образом, при разработке трехмерной анимации больше не возникает необходимость в ручном кодировании с применением API-интерфейса DirectX (хотя это можно делать), поскольку нужная функциональность уже встроена в WPF. Чтобы продемонстрировать, насколько все стало яснее, в табл. 26.2 представлена модель разработки настольных приложений, введенная в .NET 3.0.

Таблица 26.2. Решения .NET 3.0 для обеспечения желаемой функциональности

Желаемая функциональность	Технология
Построение форм с элементами управления	WPF
Поддержка двухмерной графики	WPF
Поддержка трехмерной графики	WPF
Поддержка потокового видео	WPF
Поддержка документов нефиксированного формата	WPF

Очевидное преимущество здесь в том, что программисты .NET теперь имеют единый симметричный API-интерфейс для всех распространенных нужд, возникающих во время разработки графических пользовательских интерфейсов настольных приложений. Освоившись с функциональностью основных сборок WPF и грамматикой XAML, вы будете приятно удивлены, насколько быстро с их помощью можно создавать очень сложные пользовательские интерфейсы.

Обеспечение разделения ответственности через XAML

Возможно, одно из наиболее значительных преимуществ заключается в том, что инфраструктура WPF предоставляет способ аккуратного отделения внешнего вида и поведения приложения с графическим пользовательским интерфейсом от программной логики, которая им управляет. Используя язык XAML, пользовательский интерфейс приложения можно определять через *разметку* XML. Такая разметка (в идеале генерируемая с помощью инструментов вроде Microsoft Visual Studio или Microsoft Expression Blend) затем может быть подключена к связанному файлу кода для обеспечения внутренней части функциональности программы.

На заметку! Язык XAML не ограничивается приложениями WPF. Любое приложение может применять XAML для описания дерева объектов .NET, даже если они не имеют никакого отношения к видимому пользовательскому интерфейсу. Например, в API-интерфейсе Windows Workflow Foundation грамматика, основанная на XAML, используется для определения бизнес-процессов и специальных действий. Кроме того, XAML применяют другие инфраструктуры для построения графических пользовательских интерфейсов в .NET, такие как Silverlight (хотя и теряющая популярность, но все еще интенсивно используемая в настоящее время), а также приложения Windows Phone и Windows 10.

По мере погружения в WPF вас может удивить, насколько высокую гибкость обеспечивает эта "настольная разметка". Язык XAML позволяет определять не только простые элементы пользовательского интерфейса (кнопки, таблицы, окна списков и т.д.) в разметке, но также интерактивную двух- и трехмерную графику, анимацию, логику привязки данных и функциональность мультимедиа (наподобие воспроизведения видео).

Кроме того, XAML делает очень легкой настройку визуализации элемента управления. Например, определение круглой кнопки, на которой выполняется анимация логотипа компании, требует всего нескольких строк разметки. Как показано в главе 29, элементы управления WPF могут быть модифицированы посредством стилей и шаблонов, которые позволяют изменять целиком внешний вид приложения с минимальными усилиями. В отличие от разработки с помощью Windows Forms единственной веской причиной для построения специального элемента управления WPF с нуля является необходимость в изменении *поведения* элемента управления (например, добавление специальных методов, свойств или событий либо создание подкласса существующего эле-

мента управления с целью переопределения виртуальных членов). Если нужно просто изменить *внешний вид* элемента управления (как в случае с круглой анимированной кнопкой), то это можно делать полностью через разметку.

Обеспечение оптимизированной модели визуализации

Наборы инструментов для построения графических пользовательских интерфейсов, такие как Windows Forms, MFC или VB6, выполняют все запросы графической визуализации (включая визуализацию элементов управления вроде кнопок и окон со списком) с применением низкоуровневого API-интерфейса на основе C (GDI), который в течение многих лет был частью Windows. Интерфейс GDI обеспечивает адекватную производительность для типовых бизнес-приложений или простых графических программ; однако если приложению с пользовательским интерфейсом нужна была высокопроизводительная графика, то приходилось обращаться к DirectX.

Программная модель WPF полностью отличается тем, что при визуализации графических данных GDI *не* используется. Все операции визуализации (двух- и трехмерная графика, анимация, визуализация элементов управления и т.д.) теперь работают с API-интерфейсом DirectX. Очевидная выгода такого подхода в том, что приложения WPF будут автоматически получать преимущества аппаратной и программной оптимизации. Вдобавок приложения WPF могут задействовать очень развитые графические службы (эффекты размытия, сглаживания, прозрачности и т.п.) без сложностей, присущих программированию напрямую с применением API-интерфейса DirectX.

На заметку! Хотя WPF переносит все запросы визуализации на уровень DirectX, нельзя утверждать, что приложение WPF будет работать настолько же быстро, как приложение, построенное с использованием неуправляемого языка C++ и DirectX. Несмотря на значительные усовершенствования, внесенные в WPF в версии .NET 4.6, если вы намереваетесь строить настольное приложение, которое требует максимально возможной скорости выполнения (вроде трехмерной игры), то неуправляемый C++ и DirectX по-прежнему являются наилучшим подходом.

Упрощение программирования сложных пользовательских интерфейсов

Чтобы подвести итог сказанному до сих пор: Windows Presentation Foundation (WPF) — это API-интерфейс, предназначенный для построения настольных приложений, который интегрирует разнообразные настольные API-интерфейсы в единую объектную модель и обеспечивает четкое разделение ответственности через XAML. В дополнение к указанным важнейшим моментам приложения WPF также выигрывают от очень простого способа интеграции со службами, что исторически было довольно сложным. Ниже кратко перечислены основные функциональные возможности WPF.

- Множество диспетчеров компоновки (намного больше, чем в Windows Forms) для обеспечения исключительно гибкого контроля над размещением и изменением позиций содержимого.
- Применение расширенного механизма привязки данных для связывания содержимого с элементами пользовательского интерфейса разнообразными способами.
- Встроенный механизм стилей, который позволяет определять "темы" для приложения WPF.
- Использование векторной графики, которая поддерживает автоматическое изменение размеров содержимого для соответствия размеру и разрешению экрана, отображающего пользовательский интерфейс приложения.

- Поддержка двух- и трехмерной графики, анимации, а также воспроизведения видео и аудио.
- Развитый типографский API-интерфейс, который поддерживает документы XML Paper Specification (XPS), фиксированные документы (WYSIWYG), документы нефиксированного формата и аннотации в документах (например, API-интерфейс Sticky Notes).
- Поддержка взаимодействия с унаследованными моделями графических пользовательских интерфейсов (такими как Windows Forms, ActiveX и HWND-дескрипторы Win32). Например, в приложение WPF можно встраивать специальные элементы управления Windows Forms и наоборот.

Теперь, имея определенное представление о том, что инфраструктура WPF привносит в платформу, давайте рассмотрим разнообразные типы приложений, которые могут быть созданы с применением этого API-интерфейса. Многие из перечисленных выше возможностей будут подробно исследоваться в последующих главах.

Различные варианты приложений WPF

API-интерфейс WPF может использоваться для построения широкого разнообразия приложений с графическим пользовательским интерфейсом, которые в основном отличаются структурой навигации и моделями развертывания. Ниже будет представлен их высокоуровневый обзор.

Традиционные настольные приложения

Первый (и самый привычный) вариант предусматривает применение WPF для построения традиционной исполняемой сборки, которая запускается на локальной машине. Например, инфраструктуру WPF можно было бы использовать для построения текстового редактора, программы рисования или мультимедийной программы, такой как цифровой музыкальный проигрыватель, средство просмотра фотографий и т.д. Подобно любому другому настольному приложению такие файлы `*.exe` могут устанавливаться традиционными средствами (программами установки, пакетами Windows Installer и т.д.) или же с помощью технологии ClickOnce, которая позволяет распространять и устанавливать настольные приложения через удаленный веб-сервер.

Говоря языком программирования, помимо ожидаемого набора диалоговых окон, панелей инструментов, панелей состояния, систем меню и других элементов пользовательского интерфейса в приложении WPF этого типа будут применяться (минимум) классы `Window` и `Application`.

Несомненно, WPF можно использовать для построения базовых бизнес-приложений, лишенных каких-либо украшений, но инфраструктура WPF действительно будет блестящей в случае *встраивания* таких средств. Взгляните на рис. 26.1, где показан пример настольного приложения WPF для просмотра медицинских карточек пациентов в учреждении здравоохранения.

К сожалению, на печатной странице невозможно отразить полный набор средств данного приложения. Например, после запуска этого приложения вы заметите, что в правом верхнем углу главного окна отображается график реального времени, показывающий синусовый ритм пациента. Если щелкнуть на кнопке Patient Details (Информация о пациенте) в нижнем правом углу, то выполнится анимация, которая зеркально отобразит, повернет и трансформирует пользовательский интерфейс, приведя его к виду как на рис. 26.2.

Можно ли построить то же самое приложение без WPF? Безусловно. Тем не менее, объем кода и его сложность будут намного выше.

Рис. 26.1. Настольное приложение WPF, в котором применяется несколько API-интерфейсов WPF

Рис. 26.2. Трансформация и анимация в WPF реализуется очень легко

На заметку! Примеры приложений WPF доступны для загрузки по адресу `https://msdn.microsoft.com/ru-ru/library/ms771633(v=vs.90).aspx`. Здесь вы найдете многочисленные документы по WPF (и Windows Forms), примеры проектов, демонстрации технологии и ссылки на форумы. К сожалению, на момент написания книги примеры для версии .NET 4.6 отсутствовали. Хорошая новость в том, что команда разработчиков XAML занимается обновлением примеров с учетом .NET 4.6.

Приложения WPF на основе навигации

Приложения WPF могут по выбору использовать структуру на основе навигации, которая позволяет традиционному настольному приложению получить базовое поведение приложения веб-браузера. С применением этой модели можно строить настольное приложение `*.exe` с кнопками "вперед" и "назад", которые позволяют конечному поль-

зователю перемещаться вперед и назад по различным экранам пользовательского интерфейса, называемым *страницами*.

Приложение такого типа поддерживает список всех страниц и предоставляет необходимую инфраструктуру для навигации по ним, передачи данных между страницами (подобно переменным в веб-приложении) и ведения списка хронологии. В качестве примера взгляните на проводник Windows (рис. 26.3), в котором используется функциональность подобного рода. Обратите внимание на кнопки навигации, находящиеся в верхнем левом углу окна.

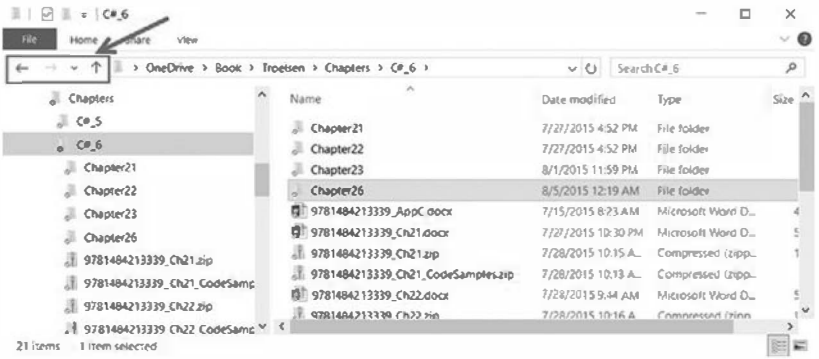


Рис. 26.3. Настольная программа на основе навигации

Несмотря на то что настольное приложение WPF может принимать веб-подобную схему навигации, необходимо понимать, что это всего лишь вопрос дизайна пользовательского интерфейса. Само приложение по-прежнему является просто исполняемой сборкой, функционирующей на настольной машине, и помимо внешнего сходства не имеет ничего общего с веб-приложением. Говоря языком программирования, такая разновидность приложений WPF построена с применением классов вроде `Application`, `Page`, `NavigationWindow` и `Frame`.

Приложения XBAP

Инфраструктура WPF также позволяет строить приложения, которые могут размещаться *внутри* веб-браузера. Такая разновидность приложений WPF называется *браузерными приложениями XAML* (XAML browser application), или XBAP. В рамках этой модели конечный пользователь переходит по заданному URL-адресу, где приложение XBAP (по существу представляющее собой коллекцию объектов `Page`) прозрачно загружается и устанавливается на локальной машине. Однако в отличие от традиционной установки ClickOnce для исполняемого приложения программа XBAP размещается прямо в браузере и принимает встроенную систему навигации браузера. На рис. 26.4 показана программа XBAP в действии.

Одно из преимуществ технологии XBAP связано с тем, что она позволяет создавать сложные пользовательские интерфейсы, которые отличаются намного большей выразительностью, чем типичная веб-страница, построенная с помощью HTML и JavaScript (хотя HTML5 определенно улучшает текущее положение дел). Объект `Page` в XBAP может задействовать те же самые службы, что и настольное приложение WPF, включая анимацию, двух- и трехмерную графику, темы и т.д. В действительности веб-браузер является просто контейнером для объектов `Page`, а *не* средством отображения веб-страниц ASP.NET. Тем не менее, поскольку объекты `Page` развертываются на удаленном веб-сервере, приложения XBAP можно легко сопровождать в разных версиях и обновлять без повторного развертывания исполняемых сборок на пользовательских настольных машинах.

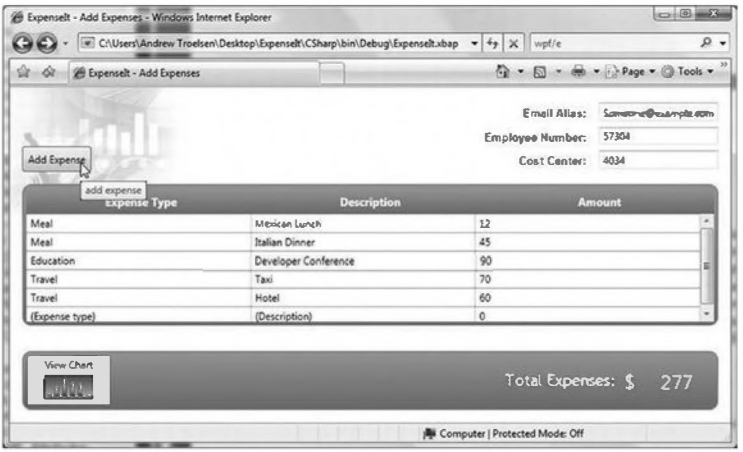


Рис. 26.4. Программы XBAP загружаются на локальную машину и размещаются внутри веб-браузера

Подобно традиционному веб-приложению объекты `Page` можно просто обновлять на веб-сервере, и пользователь всегда будет получать самую актуальную версию, обращаясь по соответствующему URL-адресу.

Недостаток этой разновидности программ WPF заключается в том, что приложения XBAP должны размещаться внутри веб-браузеров Microsoft Internet Explorer или Firefox. Следует отметить, что приложения XBAP не поддерживаются в новом браузере Windows 10, который называется Microsoft Edge; в среде Windows 10 придется использовать Internet Explorer. При развертывании приложений XBAP в корпоративной сети компании совместимость браузеров не должна быть проблемой, т.к. системные администраторы могут диктовать выбор браузера, обязательного для установки на машинах пользователей. Однако если доступ к приложению XBAP открыт внешнему миру, то невозможно гарантировать, что каждый конечный пользователь будет работать с браузером Internet Explorer или Firefox, а потому некоторые внешние пользователи просто не смогут его просмотреть.

Еще одна проблема касается того, что машина, на которой просматривается приложение XBAP, должна иметь установленную локальную копию платформы .NET, т.к. объекты `Page` будут иметь дело с теми же самыми сборками .NET, которые применяются в традиционных приложениях. Учитывая этот специфический момент, приложения XBAP ограничены только системами Windows и не могут просматриваться на системах, работающих под управлением Mac OS X или Linux.

На заметку! Наряду с тем, что в Visual Studio 2015 по-прежнему можно создавать проекты XBAP, они в значительной степени замещены приложениями, основанными на HTML5/JavaScript (такими как ASP.NET MVC).

Отношения между WPF и Silverlight

Инфраструктура WPF и язык XAML также предоставляют фундамент для межплатформенной, межбраузерной, основанной на WPF технологии, которая называется *Silverlight*. На самом высоком уровне Silverlight можно считать конкурентом Adobe Flash с преимуществами использования C# и XAML, а не новым набором инструментов и языков. Silverlight является подмножеством функциональности WPF, применяемой при построении интерактивных подключаемых модулей для крупных страниц на основе HTML. Тем не менее, в действительности Silverlight — это полностью уникальный дистрибутив платформы .NET, в состав которого входят уменьшенные версии среды CLR и библиотек базовых классов .NET.

В отличие от XBAP устанавливать полную версию .NET Framework на машине пользователя не требуется. При условии, что на целевой машине имеется установленная исполняющая среда Silverlight, браузер будет загружать ее и отображать приложения Silverlight автоматически. Лучше всего то, что подключаемые модули Silverlight не ограничены операционными системами Windows. В Microsoft также создали исполняющую среду Silverlight для Mac OS X.

С помощью Silverlight можно строить исключительно многофункциональные (и интерактивные) веб-приложения. Например, как и WPF, технология Silverlight обладает системой векторной графики, поддержкой анимации и поддержкой мультимедиа. Более того, в приложения можно внедрять подмножество библиотек базовых классов .NET. Это подмножество включает API-интерфейсы LINQ, обобщенные коллекции, поддержку WCF и полезный набор функциональности из сборки `mscorlib.dll` (файловый ввод-вывод, манипулирование XML и т.д.).

На заметку! В Microsoft отходят от Silverlight как платформы для разработки. Хотя в Visual Studio 2015 по-прежнему можно создавать проекты Silverlight, потребность в Silverlight в значительной степени вытесняется приложениями, основанными на HTML5/JavaScript. Компания Microsoft обязуется сопровождать технологию Silverlight на протяжении 10 лет после того, как она официально прекратит свое существование, что дает еще около 8 лет (в зависимости от того, когда вы читаете данную книгу) активной поддержки.

Исследование сборок WPF

Независимо от того, какого типа приложение WPF вы собираетесь строить, в конечном итоге WPF — это главным образом коллекция типов, встроенных в сборки WPF. В табл. 26.3 описаны основные сборки, используемые при разработке приложений WPF, на каждую из которых должна быть добавлена ссылка, когда создается новый проект. Как и следовало ожидать, проекты WPF в Visual Studio ссылаются на эти обязательные сборки автоматически.

Таблица 26.3. Основные сборки WPF

Сборка	Описание
`PresentationCore.dll`	В этой сборке определены многочисленные пространства имен, которые образуют фундамент уровня графического пользовательского интерфейса в WPF. Например, она включает поддержку API-интерфейса WPF Ink (для программирования перьевого ввода в Pocket PC и Tablet PC), примитивы анимации и множество типов графической визуализации
`PresentationFramework.dll`	В этой сборке содержится большинство элементов управления WPF, классы `Application` и `Window`, поддержка интерактивной двухмерной графики и многочисленные типы, применяемые для привязки данных
`System.Xaml.dll`	Эта сборка предоставляет пространства имен, которые позволяют программно взаимодействовать с документами XAML во время выполнения. В общем и целом она полезна только при разработке инструментов поддержки WPF или когда нужен абсолютный контроль над разметкой XAML во время выполнения
`WindowsBase.dll`	В этой сборке определены типы, которые формируют инфраструктуру API-интерфейса WPF, включая типы потоков WPF, типы безопасности, разнообразные преобразователи типов и поддержку *свойств зависимости* и *маршрутизируемых событий* (описанных в главе 27)

Коллективно эти четыре сборки определяют несколько новых пространств имен и сотни новых классов, интерфейсов, структур, перечислений и делегатов .NET. За подробными сведениями следует обращаться в документацию .NET Framework 4.6 SDK, а в табл. 26.4 описана роль некоторых (далеко не всех) важных пространств имен.

Таблица 26.4. Основные пространства имен WPF

Пространство имен	Описание
`System.Windows`	Это корневое пространство имен WPF. Здесь находятся основные классы (такие как `Application` и `Window`), которые требуются в любом проекте настольного приложения WPF
`System.Windows.Controls`	Содержит все ожидаемые графические элементы (виджеты) WPF, включая типы для построения систем меню, всплывающие подсказки и многочисленные диспетчеры компоновки
`System.Windows.Data`	Содержит типы для работы с механизмом привязки данных WPF, а также для поддержки шаблонов привязки данных
`System.Windows.Documents`	Содержит типы для работы с API-интерфейсом документов, который позволяет интегрировать в приложения WPF функциональность в стиле PDF через протокол XML Paper Specification (XPS)
`System.Windows.Ink`	Предоставляет поддержку Ink API — интерфейса, который позволяет получать ввод от пера или мыши, реагировать на входные жесты и т.д. Этот API-интерфейс очень полезен при программировании для Tablet PC, но может использоваться также в любых приложениях WPF
`System.Windows.Markup`	В этом пространстве имен определено множество типов, обеспечивающих программный анализ и обработку разметки XAML (и эквивалентного двоичного формата BAML)
`System.Windows.Media`	Это корневое пространство имен для нескольких пространств имен, связанных с мультимедиа. Внутри таких пространств имен определены типы для работы с анимацией, визуализацией трехмерной графики, визуализацией текста и прочими мультимедийными примитивами
`System.Windows.Navigation`	Это пространство имен предоставляет типы для обеспечения логики навигации, применяемой браузерными приложениями XAML (XBAP), а также настольными приложениями, которые требуют страничной модели навигации
`System.Windows.Shapes`	В этом пространстве имен определены классы, которые позволяют визуализировать двухмерную графику, автоматически реагирующую на ввод с помощью мыши

В начале путешествия по программной модели WPF мы исследуем два члена пространства имен `System.Windows`, которые являются общими при традиционной разработке любого настольного приложения: `Application` и `Window`.

На заметку! Если вы создавали пользовательские интерфейсы для настольных приложений с использованием API-интерфейса Windows Forms, то имейте в виду, что сборки `System.Windows.Forms.*` и `System.Drawing.*` никак не связаны с WPF. Эти библиотеки представляют первоначальный инструментальный набор .NET для построения графических пользовательских интерфейсов, т.е. Windows Forms/GDI+.

Роль класса Application

Класс System.Windows.Application представляет глобальный экземпляр выполняющегося приложения WPF. В нем имеется метод Run() (для запуска приложения), комплект событий, которые можно обрабатывать для взаимодействия с приложением на протяжении его времени жизни (наподобие Startup и Exit), и набор членов, специфичных для браузерных приложений XAML (таких как события, инициируемые во время навигации пользователя по страницам). В табл. 26.5 описаны основные свойства класса Application.

Таблица 26.5. Основные свойства класса Application

Свойство	Описание
Current	Это статическое свойство позволяет получать доступ к работающему объекту Application из любого места кода. Может быть очень полезно, когда обычному или диалоговому окну необходим доступ к объекту Application, который его создал, обычно для взаимодействия с переменными или функциональностью уровня приложения
MainWindow	Это свойство позволяет программно получать или устанавливать главное окно приложения
Properties	Это свойство позволяет устанавливать и получать данные, доступные через все аспекты приложения WPF (окна, диалоговые окна и т.д.)
StartupUri	Это свойство получает или устанавливает URI, который указывает окно или страницу для автоматического открытия при запуске приложения
Windows	Это свойство возвращает объект тип WindowCollection, предоставляющий доступ ко всем окнам, которые созданы в потоке, создавшем объект Application. Может быть очень удобным, когда нужно пройти по всем открытым окнам приложения и изменить их состояние (скажем, свернуть все окна)

Построение класса Application

В любом приложении WPF понадобится определить класс, расширяющий Application. Внутри этого класса будет определена точка входа программы (метод Main()), которая создает экземпляр данного подкласса и обычно обрабатывает события Startup и Exit. Вскоре будет рассмотрен полноценный проект, а пока вот краткий пример:

```
// Определить глобальный объект приложения для этой программы WPF.
class MyApp : Application
{
  [STAThread]
  static void Main(string[] args)
  {
    // Создать объект приложения.
    MyApp app = new MyApp();
    // Зарегистрировать события Startup/Exit.
    app.Startup += (s, e) => { /* Запуск приложения */ };
    app.Exit += (s, e) => { /* Завершение приложения */ };
  }
}
```

В обработчике события `Startup` чаще всего будут обрабатываться входные аргументы командной строки и запускаться главное окно программы. Обработчик `Exit`, как и следовало ожидать — это место, куда можно поместить любую необходимую логику завершения программы (например, сохранение пользовательских предпочтений и запись в реестр Windows).

Перечисление элементов коллекции `Windows`

Еще одним интересным свойством класса `Application` является `Windows`, обеспечивающее доступ к коллекции, которая представляет все окна, загруженные в память для текущего приложения WPF. Вспомните, что создаваемые новые объекты `Window` автоматически добавляются в коллекцию `Application.Windows`. Ниже приведен пример метода, который сворачивает все окна приложения (возможно в ответ на нажатие определенного сочетания клавиш или выбор пункта меню конечным пользователем):

```
static void MinimizeAllWindows()
{
  foreach (Window wnd in Application.Current.Windows)
  {
    wnd.WindowState = WindowState.Minimized;
  }
}
```

В рассматриваемом далее примере будет построен завершенный тип, производный от `Application`. А пока давайте выясним основную функциональность типа `Window` и изучим несколько важных базовых классов WPF.

Роль класса `Window`

Класс `System.Windows.Window` (из сборки `PresentationFramework.dll`) представляет одиночное окно, которым владеет производный от `Application` класс, включая все отображаемые главным окном диалоговые окна. Тип `Window` вполне ожидаемо имеет несколько родительских классов, каждый из которых привносит дополнительную функциональность. На рис. 26.5 показана цепочка наследования (и реализуемые интерфейсы) для класса `System.Windows.Window`, как она выглядит в браузере объектов Visual Studio.

По мере чтения этой и последующих глав вы начнете понимать функциональность, предлагаемую многими базовыми классами WPF. Далее приведен краткий обзор функциональности каждого базового класса (полные сведения ищите в документации .NET Framework 4.6 SDK).

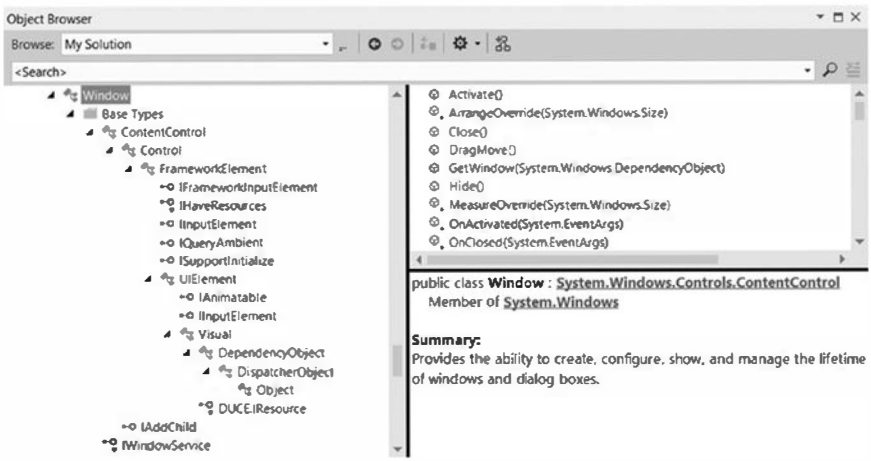


Рис. 26.5. Иерархия наследования класса `Window`

Роль класса System.Windows.Controls.ContentControl

Непосредственным родительским классом Window является класс ContentControl, который вполне можно считать наиболее впечатляющим из всех классов WPF. Этот базовый класс снабжает производные типы способностью размещать в себе одиночный фрагмент *содержимого*, который, попросту говоря, относится к визуальным данным, помещенным внутрь области элемента управления через свойство Content. Модель содержимого WPF позволяет очень легко настраивать базовый вид и поведение элемента управления ContentControl.

Например, когда речь идет о типичном "кнопочном" элементе управления, то обычно предполагается, что его содержимым будет простой строковый литерал (OK, Cancel, Abort и т.д.). Если для описания элемента управления WPF применяется XAML, а значение, которое необходимо присвоить свойству Content, может быть выражено в виде простой строки, то вот как установить свойство Content внутри открывающего определения элемента:

```
<!-- Установка значения Content в открывающем элементе -->
<Button Height="80" Width="100" Content="OK"/>
```

На заметку! Свойство Content можно также устанавливать в коде C#, что позволяет изменять внутренности элемента управления во время выполнения.

Однако содержимое может быть практически любым. Например, пусть нужна "кнопка", которая содержит в себе нечто более интересное, чем простую строку — возможно специальную графику или текстовый фрагмент. В других инфраструктурах для построения пользовательских интерфейсов, таких как Windows Forms, потребовалось бы создать специальный элемент управления, что могло повлечь за собой написание значительного объема кода и сопровождения полностью нового класса. Благодаря модели содержимого WPF в этом нет необходимости.

Когда свойству Content должно быть присвоено значение, которое невозможно выразить в виде простого массива символов, его нельзя присвоить с использованием атрибута в открывающем определении элемента управления. Взамен понадобится определить данные содержимого *неявно* внутри области действия элемента. Например, следующий элемент <Button> включает в качестве содержимого элемент <StackPanel>, который сам имеет уникальные данные (а именно — <Ellipse> и <Label>):

```
<!-- Неявная установка для свойства Content сложных данных -->
<Button Height="80" Width="100">
  <StackPanel>
    <Ellipse Fill="Red" Width="25" Height="25"/>
    <Label Content ="OK!"/>
  </StackPanel>
</Button>
```

Для установки сложного содержимого можно также применять *синтаксис "свойство-элемент"* языка XAML. Взгляните на показанное ниже функционально эквивалентное определение <Button>, которое явно устанавливает свойство Content с помощью синтаксиса "свойство-элемент" (дополнительная информация о XAML будет дана позже в главе, так что пока не обращайте внимания на детали):

```
<!-- Установка свойства Content с использованием синтаксиса "свойство-элемент" -->
<Button Height="80" Width="100">
  <Button.Content>
    <StackPanel>
      <Ellipse Fill="Red" Width="25" Height="25"/>
```

```
      <Label Content ="OK!"/>
    </StackPanel>
  </Button.Content>
</Button>
```

Имейте в виду, что не каждый элемент WPF является производным от класса ConentConrtol, поэтому не все элементы поддерживают такую уникальную модель содержимого (хотя большинство поддерживает). Кроме того, некоторые элементы управления WPF вносят несколько усовершенствований в только что рассмотренную базовую модель содержимого. В главе 27 роль содержимого WPF раскрывается более подробно.

Роль класса System.Windows.Controls.Control

В отличие от ContentControl все элементы управления WPF разделяют в качестве общего родительского класс базовый класс Control. Он предоставляет многочисленные члены, которые необходимы для обеспечения основной функциональности пользовательского интерфейса. Например, в классе Control определены свойства для установки размеров элемента управления, прозрачности, порядка обхода по нажатию клавиши <Tab>, дисплейного курсора, цвета фона и т.д. Более того, этот родительский класс предлагает поддержку *шаблонных служб*. Как объясняется в главе 29, элементы управления WPF могут полностью изменять способ визуализации своего внешнего вида, используя шаблоны и стили. В табл. 26.6 кратко описаны основные члены типа Control, сгруппированные по связанной функциональности.

Таблица 26.6. Основные члены класса Control

Член	Описание
Background, Foreground, BorderBrush, BorderThickness, Padding, HorizontalContentAlignment, VerticalContentAlignment	Эти свойства позволяют устанавливать базовые настройки, касающиеся того, как элемент управления будет визуализироваться и позиционироваться
FontFamily, FontSize, FontStretch, FontWeight	Эти свойства управляют разнообразными настройками шрифтов
IsTabStop, TabIndex	Эти свойства применяются для установки порядка обхода элементов управления в окне по нажатию клавиши <Tab>
MouseDoubleClick, PreviewMouseDoubleClick	Эти события обрабатывают действие двойного щелчка на виджете
Template	Это свойство позволяет получать и устанавливать шаблон элемента, который может быть использован для изменения вывода визуализации виджета

Роль класса System.Windows.FrameworkElement

Базовый класс FrameworkElement предоставляет члены, которые применяются повсюду в инфраструктуре WPF, такие как поддержка раскадровки (для анимации) и привязки данных, а также возможность именования членов (через свойство Name), получения любых ресурсов, определенных производным типом, и установки общих измерений производного типа. Основные члены класса FrameworkElement кратко описаны в табл. 26.7.

Таблица 26.7. Основные члены класса `FrameworkElement`

Член	Описание
`ActualHeight`, `ActualWidth`, `MaxHeight`, `MaxWidth`, `MinHeight`, `MinWidth`, `Height`, `Width`	Эти свойства управляют размерами производного типа
`ContextMenu`	Это свойство получает или устанавливает всплывающее меню, ассоциированное с производным типом
`Cursor`	Это свойство получает или устанавливает курсор мыши, ассоциированный с производным типом
`HorizontalAlignment`, `VerticalAlignment`	Это свойство управляет позиционированием типа внутри контейнера (такого как панель или окно со списком)
`Name`	Это свойство позволяет назначать имя типу, чтобы обращаться к его функциональности в файле кода
`Resources`	Это свойство предоставляет доступ к любым ресурсам, которые определены типом (система ресурсов WPF объясняется в главе 29)
`ToolTip`	Это свойство получает или устанавливает всплывающую подсказку, ассоциированную с производным типом

Роль класса `System.Windows.UIElement`

Из всех типов в цепочке наследования класса `Window` наибольший объем функциональности обеспечивает базовый класс `UIElement`. Его основная задача — предоставить производному типу многочисленные события, чтобы он мог получать фокус и обрабатывать входные запросы. Например, в этом классе предусмотрено множество событий для обслуживания операций перетаскивания, перемещений курсора мыши, клавиатурного ввода, а также ввода с помощью пера (для Pocket PC и Tablet PC).

Модель событий WPF будет подробно описана в главе 27; тем не менее, многие основные события будут выглядеть вполне знакомыми (`MouseMove`, `MouseDown`, `MouseEnter`, `MouseLeave`, `KeyUp` и т.д.). В дополнение к десяткам событий этот родительский класс предлагает свойства, предназначенные для управления фокусом, состоянием доступности, видимостью и логикой проверки попаданий (табл. 26.8).

Таблица 26.8. Основные члены класса `UIElement`

Член	Описание
`Focusable`, `IsFocused`	Эти свойства позволяют устанавливать фокус на заданный производный тип
`IsEnabled`	Это свойство позволяет управлять доступностью заданного производного типа
`IsMouseDirectlyOver`, `IsMouseOver`	Эти свойства предоставляют простой способ выполнения логики проверки попадания
`IsVisible`, `Visibility`	Эти свойства позволяют работать с настройкой видимости производного типа
`RenderTransform`	Это свойство позволяет устанавливать трансформацию, которая будет использоваться при визуализации производного типа

Роль класса System.Windows.Media.Visual

Класс Visual предлагает основную поддержку визуализации в WPF, которая включает проверку попадания для графических данных, трансформацию координат и вычисление ограничивающих прямоугольников. В действительности при рисовании данных на экране класс Visual взаимодействует с подсистемой DirectX. Как будет показано в главе 28, инфраструктура WPF поддерживает три возможных способа визуализации графических данных, каждый из которых отличается функциональностью и производительностью. Применение типа Visual (и его потомков вроде DrawingVisual) является наиболее легковесным путем визуализации графических данных, но также подразумевает написание вручную большого объема кода для учета всех требуемых служб. Более подробно об этом пойдет речь в главе 28.

Роль класса System.Windows.DependencyObject

Инфраструктура WPF поддерживает отдельную разновидность свойств .NET под названием *свойства зависимости*. Выражаясь упрощенно, свойства такого стиля предоставляют дополнительный код, чтобы позволить свойству реагировать на определенные технологии WPF, такие как стили, привязка данных, анимация и т.д. Чтобы тип поддерживал такую схему свойств, он должен быть производным от базового класса DependencyObject. Несмотря на то что свойства зависимости являются ключевым аспектом разработки WPF, большую часть времени их детали скрыты с глаз. В главе 27 мы рассмотрим свойства зависимости более подробно.

Роль класса System.Windows.Threading.DispatcherObject

Последним базовым классом для типа Window (помимо System.Object, который здесь не требует дополнительных пояснений) является DispatherObject. В нем определено одно свойство, представляющее интерес, Dispatcher, которое возвращает ассоциированный объект System.Windows.Threading.Dispatcher. Класс Dispatcher — это точка входа в очередь событий приложения WPF, и он предоставляет базовые конструкции для организации параллелизма и многопоточности.

Построение приложения WPF без XAML

С учетом всей функциональности, предлагаемой родительскими классами типа Window, представить окно в приложении можно, либо напрямую создав объект Window, либо используя этот класс в качестве родительского для строго типизированного наследника. В рассматриваемом далее примере демонстрируются оба подхода. Хотя в большинстве приложений WPF будет применяться XAML, формально так поступать не обязательно. Все, что может быть выражено в XAML, также можно выразить в коде и (по большей части) наоборот. При желании есть возможность создать завершенный проект WPF, используя лежащую в основе объектную модель и процедурный код C#.

В целях иллюстрации давайте построим минимальное, но полное приложение *без* применения XAML, работая с классами Application и Window напрямую. Начнем с создания нового проекта консольного приложения по имени WpfAppAllCode (пока переживать не стоит; шаблон проекта WPF в Visual Studio будет использоваться позже). В окне свойств проекта изменим выходной тип на Windows Application (Приложение Windows), что предотвратит появление окна консоли. Откроем диалоговое окно Add Reference (Добавление ссылки) и добавим ссылки на сборки WindowBase.dll, PresentationCore.dll, System.Xaml.dll и PresentationFramework.dll.

Поместим в начальный файл кода C# приведенный ниже код, который создает окно с ограниченной функциональностью (здесь указаны только те пространства имен, которые *должны* быть импортированы, чтобы код скомпилировался; любые автоматически включенные операторы `using` можно оставить на месте):

```
// Простое приложение WPF, написанное без XAML.
using System;
using System.Windows;
using System.Windows.Controls;

namespace WpfAppAllCode
{
  // В этом первом примере определяется единственный класс
  // для представления самого приложения и главного окна.
  class Program : Application
    {
    [STAThread]
    static void Main(string[] args)
    {
      // Обработать события Startup и Exit и затем запустить приложение.
      Program app = new Program();
      app.Startup += AppStartUp;
      app.Exit += AppExit;
      app.Run();  // Инициирует событие Startup.
    }

    static void AppExit(object sender, ExitEventArgs e)
    {
      MessageBox.Show("App has exited");
    }

    static void AppStartUp(object sender, StartupEventArgs e)
    {
      // Создать объект Window и установить некоторые базовые свойства.
      Window mainWindow = new Window();
      mainWindow.Title = "My First WPF App!";
      mainWindow.Height = 200;
      mainWindow.Width = 300;
      mainWindow.WindowStartupLocation = WindowStartupLocation.CenterScreen;
      mainWindow.Show();
    }
  }
}
```

На заметку! Метод `Main()` в приложении WPF должен быть снабжен атрибутом `[STAThread]`, который обеспечивает безопасность к потокам любых унаследованных COM-объектов, применяемых внутри приложения. Если не аннотировать `Main()` подобным образом, то сгенерируется исключение времени выполнения.

Обратите внимание, что класс `Program` расширяет класс `System.Windows.Application`. Внутри метода `Main()` создается экземпляр объекта приложения и организуется обработка событий `Startup` и `Exit` с использованием синтаксиса группового преобразования методов. Вспомните из главы 10, что такое сокращенное обозначение устраняет необходимость в ручном указании делегатов, применяемых определенным событием. Конечно, при желании лежащие в основе делегаты можно указывать прямо по их именам.

В следующем модифицированном методе `Main()` событие `Startup` работает в сочетании с делегатом `StartupEventHandler`, который может указывать только на методы, принимающие тип `Object` в первом параметре и `StartupEventArgs` во втором. С другой стороны, событие `Exit` работает с делегатом `ExitEventHandler`, который требует, чтобы указанный им метод во втором параметре принимал тип `ExitEventArgs`.

```
[STAThread]
static void Main(string[] args)
{
  // На этот раз указать лежащие в основе делегаты.
  Program app = new Program();
  app.Startup += new StartupEventHandler(AppStartUp);
  app.Exit += new ExitEventHandler(AppExit);
  app.Run(); // Инициирует событие Startup.
}
```

Метод `AppStartUp()` был сконфигурирован для создания объекта `Window`, установки некоторых базовых свойств и вызова метода `Show()` для отображения окна на экране в немодальном режиме (с помощью метода `ShowDialog()` можно открыть модальное диалоговое окно). Метод `AppExit()` использует класс `MessageBox` из WPF для отображения диагностического сообщения, когда приложение завершается.

После компиляции и запуска проекта вы обнаружите очень простое главное окно, которое можно свернуть, развернуть и закрыть. Чтобы немного украсить его, понадобится добавить несколько элементов пользовательского интерфейса. Но прежде чем делать это, давайте перепишем код с применением строго типизированного и хорошо инкапсулированного класса, производного от `Window`.

Создание строго типизированного класса окна

Сейчас производный от `Application` класс при запуске приложения создает экземпляр типа `Window`. В идеале нужно создать класс, унаследованный от `Window`, чтобы инкапсулировать его внешний вид и функциональность. Добавим в проект еще один класс по имени `MainWindow` со следующим определением (понадобится импортировать пространство имен `System.Windows`):

```
class MainWindow : Window
{
  public MainWindow(string windowTitle, int height, int width)
  {
    this.Title = windowTitle;
    this.WindowStartupLocation = WindowStartupLocation.CenterScreen;
    this.Height = height;
    this.Width = width;
  }
}
```

Теперь можно обновить обработчик события `StartUp`, чтобы напрямую создавать экземпляр класса `MainWindow`:

```
static void AppStartUp(object sender, StartupEventArgs e)
{
  // Создать объект MainWindow.
  var main = new MainWindow("My better WPF App!", 200, 300);
  main.Show();
}
```

После компиляции и запуска программы будет получен вывод, идентичный предыдущей версии. Очевидное преимущество новой версии связано с тем, что теперь есть строго типизированный класс, представляющий главное окно, на основе которого можно строить пользовательский интерфейс.

На заметку! Когда создается объект `Window` (или класса, производного от `Window`), он автоматически добавляется во внутреннюю коллекцию окон класса `Application` (посредством логики конструктора в самом классе `Window`). Свойство `Application.Windows` можно использовать для прохода по списку объектов `Window`, находящихся в текущий момент в памяти.

Создание простого пользовательского интерфейса

Добавление элемента пользовательского интерфейса (такого как `Button`) к `Window` в коде C# предусматривает выполнение описанных далее базовых шагов.

1. Определение переменной-члена для представления нужного элемента управления.
2. Конфигурирование внешнего вида и поведения элемента управления при конструировании объекта `Window`.
3. Присваивание элемента управления унаследованному свойству `Content` или в качестве альтернативы его передача как параметра унаследованному методу `AddChild()`.

Вспомните, что модель содержимого элемента управления WPF требует, чтобы в свойстве `Content` указывался одиночный элемент. Естественно, объект `Window`, который содержит только один элемент управления, не особенно полезен. Следовательно, почти в каждом случае "одиночной порцией содержимого", которая присваивается свойству `Content`, на самом деле будет диспетчер компоновки, такой как `DockPanel`, `Grid`, `Canvas` или `StackPanel`. В рамках диспетчера компоновки можно иметь любую комбинацию внутренних элементов, включая другие вложенные диспетчеры компоновки (более подробно этот аспект разработки WPF описан в главе 27).

Пока что мы добавим в производный от `Window` класс один элемент управления `Button`. Щелчок на этой кнопке приводит к закрытию текущего окна, что косвенно завершит приложение, т.к. другие окна в памяти отсутствуют. Взгляните на следующую модификацию класса `MainWindow` (для получения доступа к классу `Button` необходимо импортировать пространство имен `System.Windows.Controls`):

```
class MainWindow : Window
{
  // Наш элемент пользовательского интерфейса.
  private Button btnExitApp = new Button();

  public MainWindow(string windowTitle, int height, int width)
  {
    // Сконфигурировать кнопку и установить дочерний элемент управления.
    btnExitApp.Click += new RoutedEventHandler(btnExitApp_Clicked);
    btnExitApp.Content = "Exit Application";
    btnExitApp.Height = 25;
    btnExitApp.Width = 100;

    // Установить в качестве содержимого окна единственную кнопку.
    this.Content = btnExitApp;

    // Сконфигурировать окно.
    this.Title = windowTitle;
    this.WindowStartupLocation = WindowStartupLocation.CenterScreen;
```

```
    this.Height = height;
    this.Width = width;
    this.Show();
  }
  private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
  {
    // Закрыть окно.
    this.Close();
  }
}
```

Обратите внимание, что событие `Click` кнопки WPF работает в сочетании с делегатом по имени `RoutedEventHandler`, вызывающим очевидный вопрос о том, что собой представляет маршрутизируемое событие (routed event). Модель событий WPF подробно рассматривается в следующей главе, а пока просто имейте в виду, что цели делегата `RoutedEventHandler` должны принимать тип `Object` в первом параметре и `RoutedEventArgs` — во втором.

Рис. 26.6. Простое приложение WPF, реализованное полностью в коде C#

После компиляции и запуска приложения отобразится настроенное окно, показанное на рис. 26.6. Кнопка автоматически располагается в самом центре клиентской области окна, что является стандартным поведением, когда содержимое не помещено в тип панели WPF.

Взаимодействие с данными уровня приложения

Вспомните, что в классе `Application` имеется свойство по имени `Properties`, которое позволяет определить коллекцию пар "имя/значение" через индексатор типа. Поскольку этот индексатор предназначен для оперирования типом `System.Object`, в коллекцию можно сохранять элементы любого вида (в том числе экземпляры пользовательских классов) с целью последующего извлечения по дружественному имени. С применением такого подхода легко совместно использовать данные во всех окнах приложения WPF.

В целях иллюстрации обновим текущий обработчик события `Startup`, чтобы он проверял входящие параметры командной строки на присутствие значения `/GODMODE` (распространенный мошеннический код во многих играх). Если оно найдено, то значение `bool` по имени `GodMode` внутри коллекции свойств устанавливается в `true` (в противном случае оно устанавливается в `false`).

Звучит достаточно просто, но как передать обработчику события `Startup` входные аргументы командной строки (обычно получаемые методом `Main()`)? Один из подходов предусматривает вызов статического метода `Environment.GetCommandLineArgs()`. Однако те же самые аргументы автоматически добавляются во входной параметр `StartupEventArgs` и доступны через свойство `Args`. Таким образом, ниже приведена первая модификация текущей кодовой базы:

```
private static void AppStartUp(object sender, StartupEventArgs e)
{
  // Проверить входные аргументы командной строки
  // на предмет наличия флага /GODMODE.
  Application.Current.Properties["GodMode"] = false;
  foreach(string arg in e.Args)
  {
```

```
    if (arg.ToLower() == "/godmode")
    {
      Application.Current.Properties["GodMode"] = true;
      break;
    }
  }
  // Создать объект MainWindow.
  MainWindow wnd = new MainWindow("My better WPF App!", 200, 300);
}
```

Данные уровня приложения доступны из любого места внутри приложения WPF. Для обращения к ним потребуется лишь получить точку доступа к глобальному объекту приложения (через `Application.Current`) и просмотреть коллекцию. Например, обработчик события `Click` для кнопки можно было бы изменить следующим образом:

```
private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
{
  // Указал ли пользователь /godmode?
  if((bool)Application.Current.Properties["GodMode"])
  {
    MessageBox.Show("Cheater!");
  }
  this.Close();
}
```

Если теперь конечный пользователь запустит программу, как показано ниже:

```
WpfAppAllCode.exe /godmode
```

то при завершении приложения отобразится окно сообщения.

На заметку! Вспомните, что аргументы командной строки можно указывать и внутри Visual Studio. Для этого необходимо дважды щелкнуть на значке Properties (Свойства) в окне Solution Explorer, в открывшемся диалоговом окне перейти на вкладку Debug (Отладка) и ввести `/godmode` в поле Command line arguments (Аргументы командной строки).

Обработка закрытия объекта `Window`

Конечные пользователи могут завершать работу окна с помощью многочисленных встроенных приемов уровня системы (например, щелкнув на кнопке закрытия X на рамке окна) или вызвав метод `Close()` в ответ на некоторое действие с интерактивным элементом (скажем, выбор пункта меню File⇨Exit (Файл⇨Выход)). Инфраструктура WPF предлагает два события, которые можно перехватывать для выяснения, *действительно* ли пользователь намерен закрыть окно и удалить его из памяти. Первое такое событие — `Closing`, которое работает в сочетании с делегатом `CancelEventHandler`.

Указанный делегат ожидает целевой метод, принимающий тип `System.ComponentModel.CancelEventArgs` во втором параметре. Класс `CancelEventArgs` предоставляет свойство `Cancel`, установка которого в `true` предотвращает фактическое закрытие окна (это удобно, когда пользователю должен быть задан вопрос о том, на самом ли деле он желает закрыть окно или сначала нужно сохранить проделанную работу).

Если пользователь действительно хочет закрыть окно, то свойство `CancelEventArgs.Cancel` можно установить в `false` (стандартное значение). В итоге сгенерируется событие `Closed` (которое работает с делегатом `System.EventHandler`), представляющее собой точку, где окно полностью и безвозвратно готово к закрытию.

Модифицируем класс MainWindow для обработки упомянутых двух событий, добавив в текущий код конструктора такие операторы:

```
public MainWindow(string windowTitle, int height, int width)
{
  ...
  this.Closing += MainWindow_Closing;
  this.Closed += MainWindow_Closed;
}
```

Теперь реализуем соответствующие обработчики событий:

```
private void MainWindow_Closing(object sender,
  System.ComponentModel.CancelEventArgs e)
{
  // Выяснить, действительно ли пользователь желает закрыть окно.
  string msg = "Do you want to close without saving?";
  MessageBoxResult result = MessageBox.Show(msg,
    "My App", MessageBoxButton.YesNo, MessageBoxImage.Warning);
  if (result == MessageBoxResult.No)
  {
    // Если пользователю не желает закрывать окно, то отменить закрытие.
    e.Cancel = true;
  }
}
private void MainWindow_Closed(object sender, EventArgs e)
{
  MessageBox.Show("See ya!");
}
```

Запустим программу и попробуем закрыть окно, щелкнув либо на значке X в правом верхнем углу окна, либо на кнопке. Должно появиться диалоговое окно с запросом подтверждения, показанное на рис. 26.7.

Щелчок на кнопке Yes (Да) приведет к завершению приложения, а щелчок на кнопке No (Нет) оставит окно в памяти.

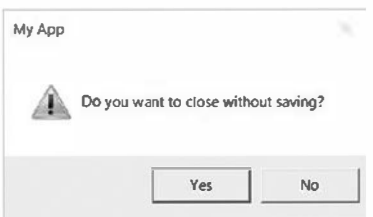


Рис. 26.7. Перехват события Closing окна

Перехват событий мыши

Инфраструктура WPF предоставляет множество событий, которые можно перехватывать для взаимодействия с мышью. В частности, базовый класс UIElement определяет такие связанные с мышью события, как MouseMove, MouseUp, MouseDown, MouseEnter, MouseLeave и т.д.

В качестве примера обработаем событие MouseMove. Это событие работает в сочетании с делегатом System.Windows.Input.MouseEventHandler, который ожидает, что его целевой метод будет принимать тип System.Windows.Input.MouseEventArgs во втором параметре. С применением класса MouseEventArgs можно извлекать позицию (*x*, *y*) курсора мыши и другие важные детали. Взгляните на следующее частичное определение:

```
public class MouseEventArgs : InputEventArgs
{
  ...
  public Point GetPosition(IInputElement relativeTo);
  public MouseButtonState LeftButton { get; }
  public MouseButtonState MiddleButton { get; }
  public MouseDevice MouseDevice { get; }
```

```
  public MouseButtonState RightButton { get; }
  public StylusDevice StylusDevice { get; }
  public MouseButtonState XButton1 { get; }
  public MouseButtonState XButton2 { get; }
}
```

На заметку! Свойства `XButton1` и `XButton2` позволяют взаимодействовать с "расширенными кнопками мыши" (вроде кнопок "вперед" и "назад", которые имеются в некоторых устройствах). Они часто используются для взаимодействия с хронологией навигации в браузере для перемещения между посещенными страницами.

Метод `GetPosition()` позволяет получать значение (x, y) относительно элемента пользовательского интерфейса в окне. Если интересует позиция относительно активного окна, то нужно просто передать `this`. В конструкторе класса `MainWindow` необходимо добавить обработчик события `MouseMove`:

```
public MainWindow(string windowTitle, int height, int width)
{
  ...
  this.MouseMove += MainWindow_MouseMove;
}
```

Ниже показан обработчик события `MouseMove`, который отобразит местоположение курсора мыши в области заголовка окна (обратите внимание, что возвращенный тип `Point` транслируется в строковое значение посредством вызова `ToString()`):

```
private void MainWindow_MouseMove(object sender,
  System.Windows.Input.MouseEventArgs e)
{
  // Отобразить в заголовке окна текущую позицию (x, y) курсора мыши.
  this.Title = e.GetPosition(this).ToString();
}
```

Перехват событий клавиатуры

Обработка клавиатурного ввода для окна, на котором находится фокус, также очень проста. В классе `UIElement` определено несколько событий, которые можно перехватывать для отслеживания нажатий клавиш клавиатуры на активном элементе (например, `KeyUp`, `KeyDown`). События `KeyUp` и `KeyDown` работают с делегатом `System.Windows.Input.KeyEventHandler`, который ожидает во втором параметре тип `KeyEventArgs`, определяющий набор важных открытых свойств:

```
public class KeyEventArgs : KeyboardEventArgs
{
  ...
  public bool IsDown { get; }
  public bool IsRepeat { get; }
  public bool IsToggled { get; }
  public bool IsUp { get; }
  public Key Key { get; }
  public KeyStates KeyStates { get; }
  public Key SystemKey { get; }
}
```

Чтобы проиллюстрировать организацию обработки события `KeyDown` в конструкторе `MainWindow` (как делалось для предыдущих событий), мы реализуем следующий обработчик события, который изменяет содержимое кнопки на информацию о текущей нажатой клавише:

```
private void MainWindow_KeyDown(object sender,
                                System.Windows.Input.KeyEventArgs e)
{
  // Отобразить на кнопке нажатую клавишу.
  btnExitApp.Content = e.Key.ToString();
}
```

На рис. 26.8 показан конечный результат работы первой программы WPF.

Рис. 26.8. Первая программа WPF, не содержащая разметку XAML

К настоящему моменту WPF может показаться всего лишь очередной инфраструктурой для построения графических пользовательских интерфейсов, которая предлагает (более или менее) те же самые службы, что и Windows Forms, MFC или VB6. Если это именно так, то возникает вопрос о смысле наличия еще одного инструментального набора для создания пользовательских интерфейсов. Чтобы оценить уникальность WPF, потребуется освоить основанную на XML грамматику — XAML.

Исходный код. Проект `WpfAppAllCode` доступен в подкаталоге `Chapter_26`.

Построение приложения WPF с использованием только XAML

Типичное приложение WPF *не* будет состоять исключительно из кода, как было в первом примере. Наоборот, файлы кода C# будут сочетаться со связанным исходным файлом XAML и вместе представлять целостность заданного объекта `Window` или `Application`, а также объектов других классов, которые пока не рассматривались, таких как `UserControl` и `Page`.

Подобный подход называется *подходом с файлами кода* к построению приложения WPF, и он будет широко применяться при раскрытии инфраструктуры WPF в оставшемся материале книги. Тем не менее, прежде чем двигаться дальше, в следующем примере мы продемонстрируем построение приложения WPF с использованием только файлов XAML. Хотя такой подход применять не рекомендуется, он поможет лучше понять, каким образом блоки разметки XAML трансформируются в кодовую базу C# и в конечном итоге в сборку .NET.

На заметку! В следующем примере используются приемы XAML, которые пока не рассматривались, так что не переживайте, если встретите незнакомый синтаксис. Можете просто загрузить файлы решения в текстовый редактор и просматривать код строку за строкой, но не применяйте для этого среду Visual Studio! Среда Visual Studio будет автоматически вносить изменения в содержимое и тем самым мешать достижению цели данного раздела. Вы еще потратите *массу* времени на использование Visual Studio 2015 для создания приложений WPF, но только не сейчас.

В общем случае файлы XAML будут содержать разметку, которая описывает внешний вид и поведение окна, тогда как связанные с ними файлы кода C# — логику реализации. Например, файл XAML для объекта `Window` может описывать общую систему разметки, элементы управления внутри системы разметки, а также имена разнообразных обработчиков событий. Связанный файл C# будет содержать логику реализации этих обработчиков событий и любой специальный код, требующийся приложению.

Язык XAML — это основанная на XML грамматика, позволяющая определять состояние (и до некоторой степени функциональность) дерева объектов .NET посредством разметки. Хотя XAML часто применяется при построении пользовательских интерфейсов с помощью WPF, в действительности его можно использовать для описания любого дерева *неабстрактных* типов .NET (включая разработанные вами специальные типы, которые определены в отдельной сборке .NET) при условии, что каждый тип поддерживает стандартный конструктор. Вы увидите, что разметка внутри файла `*.xaml` трансформируется в полноценную объектную модель.

Поскольку грамматика XAML основана на XML, мы получаем все преимущества (и недостатки) языка XML. Положительная сторона в том, что файлы XAML чрезвычайно самоописательны (как любой документ XML). В общем и целом каждый элемент в файле XAML представляет имя типа (такое как `Button`, `Window` или `Application`) внутри заданного пространства имен .NET. Атрибуты в пределах области действия открывающего элемента отображаются на свойства (`Height`, `Width` и т.д.) и события (`Startup`, `Click` и т.п.) указанного типа.

Учитывая тот факт, что XAML является просто декларативным способом определения состояния объекта, виджет WPF можно определить через разметку или процедурный код. Например, следующий код XAML:

```
<!-- Определение WPF-элемента Button в XAML -->
<Button Name = "btnClickMe" Height = "40" Width = "100" Content = "Click Me" />
```

программно может быть представлен так:

```
// Определение того же самого WPF-элемента Button в коде C#.
Button btnClickMe = new Button();
btnClickMe.Height = 40;
btnClickMe.Width = 100;
btnClickMe.Content = "Click Me";
```

Отрицательная сторона связана с тем, что разметка XAML может быть довольно многословной, и она зависит от регистра символов (подобно любому документу XML). Таким образом, сложные определения XAML могут дать в результате значительный объем разметки. Большинству разработчиков не придется вручную создавать полные описания XAML для своих приложений WPF. Основная часть работы (к счастью) будет передаваться инструментам разработки, таким как Visual Studio, Microsoft Expression Blend или другим продуктам от независимых поставщиков. После того как инструмент сгенерирует базовую разметку, определения XAML при необходимости можно точно настроить вручную.

Определение объекта `Window` в XAML

Несмотря на то что инструменты способны генерировать для вас много разметки XAML, важно понимать основы синтаксиса XAML и то, каким образом разметка в итоге трансформируется в допустимую сборку .NET. Чтобы проиллюстрировать XAML в действии, в следующем примере мы построим приложение WPF с применением только пары файлов `*.xaml`.

Первый производный от `Window` класс (`MainWindow`) был определен в коде C# как тип класса, который расширял базовый класс `System.Windows.Window`. Этот класс содержит единственный объект кнопки (`Button`), щелчок на которой приводит к вызову зарегистрированного обработчика события. Определение того же типа `Window` с помощью грамматики XAML может выглядеть так, как показано ниже. Создадим в простом текстовом редакторе (вроде Блокнота) новый файл по имени `MainWindow.xaml`, сохранив его в легко доступном подкаталоге на диске C:, чтобы без проблем обрабатывать в командной строке. Поместим в файл `MainWindow.xaml` приведенную ниже разметку XAML:

```
<!-- Определение объекта Window -->
<Window x:Class="WpfAppAllXaml.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="A Window built using 100% XAML"
  Height="200" Width="300"
  WindowStartupLocation ="CenterScreen">
  <Windows.Content>
    <!-- Это окно содержит единственную кнопку -->
    <Button x:Name="btnExitApp" Width="133" Height="24"
            Content = "Close Window" Click ="btnExitApp_Clicked"/>
  </Window.Content>
  <!-- Реализация обработчика события Click кнопки -->
  <x:Code>
    <![CDATA[
      private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
      {
        this.Close();
      }
    ]]>
  </x:Code>
</Window>
```

Для начала обратите внимание, что корневой элемент `<Window>` использует атрибут `Class` для указания имени класса, который будет сгенерирован при обработке данного файла XAML. Кроме того, атрибут `Class` снабжен префиксом `x:`. Заглянув в открывающий элемент `<Window>`, вы увидите, что этому префиксу дескриптора XML присваивается строка `"http://schemas.microsoft.com/winfx/2006/xaml"` для построения объявления пространства имен XML. Детали определений пространств имен XML станут ясными чуть позже в главе, а пока просто имейте в виду, что всякий раз, когда нужно ссылаться на элемент, определенный в пространстве имен XAML под названием `http://schemas.microsoft.com/winfx/2006/xaml`, должен быть указан префикс `x:`.

Внутри области действия открывающего дескриптора `<Window>` заданы значения для атрибутов `Title`, `Height`, `Width` и `WindowStartupLocation`, которые напрямую отображаются на одноименные свойства, поддерживаемые классом `System.Windows.Window` из сборки `PresentationFramework.dll`.

Далее обратите внимание, что в области определения окна находится разметка, описывающая внешний вид и поведение экземпляра `Button`, который будет применяться для неявной установки свойства `Content` окна. Кроме настройки имени переменной (с использованием `x:Name`) и общих размеров мы также обрабатываем событие `Click` типа `Button`, присвоив метод делегату для его вызова при возникновении события `Click`.

Финальным аспектом файла XAML является элемент `<x:Code>`, который позволяет определять обработчики событий и прочие методы этого класса внутри файла `*.xaml`. В качестве меры безопасности сам код помещен в контекст `CDATA`, чтобы предотвратить попытки анализатора XML напрямую интерпретировать данные (правда, в текущем примере это не обязательно).

Важно отметить, что писать специальную функциональность внутри элемента `<x:Code>` не рекомендуется. Хотя такой "подход с единственным файлом" изолирует все действия в одном месте, встроенный код не обеспечивает четкого разделения ответственности между разметкой пользовательского интерфейса и программной логикой. В большинстве приложений WPF код реализации будет находиться в связанном файле C# (что мы в итоге и сделаем).

Определение объекта Application в XAML

Вспомните, что язык XAML может применяться для определения в разметке любого неабстрактного класса .NET, который поддерживает стандартный конструктор. С учетом этого в разметке можно также определить объект приложения. Взгляните на следующее содержимое нового файла MyApp.xaml:

```
<!-- Похоже, отсутствует метод Main()!
     Однако атрибут StartupUri является
     его функциональным эквивалентом -->
<Application x:Class="WpfAppAllXaml.MyApp"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
</Application>
```

Вы наверняка согласитесь с утверждением, что здесь отображение между производным от Application классом C# и его описанием XAML не так очевидно, как было в случае с определения XAML для MainWindow. В частности, не видно никаких следов метода Main(). Поскольку любой исполняемый файл .NET должен иметь точку входа, вполне корректно предположить, что она будет сгенерирована во время компиляции частично на основе свойства StartupUri. Значение, присвоенное свойству StartupUri, представляет ресурс XAML, подлежащий загрузке при запуске приложения. В этом примере в свойстве StartupUri было указано имя ресурса XAML, определяющего наш начальный объект Window, т.е. MainWindow.xaml.

Несмотря на то что метод Main() автоматически создается на этапе компиляции, допускается использовать элемент <x:Code> для написания других блоков кода C#. Например, если необходимо отобразить сообщение, когда программа завершает работу, можно реализовать обработчик события Exit (обратите внимание, что в открывающем элементе <Application> теперь устанавливается атрибут Exit для перехвата события Exit класса Application):

```
<Application x:Class="WpfAppAllXaml.MyApp"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml" Exit ="AppExit">
  <x:Code>
    <![CDATA[
    private void AppExit(object sender, ExitEventArgs e)
    {
      MessageBox.Show("App has exited");
    }
  ]]>
 </x:Code>
</Application>
```

Обработка файлов XAML с использованием msbuild.exe

К настоящему моменту все готово к трансформации разметки в допустимую сборку .NET. Однако напрямую применять для этого компилятор C# невозможно. Пока что компилятор C# не обладает встроенной возможностью распознавания разметки XAML. Тем не менее, утилите командной строки msbuild.exe известно, как трансформировать разметку XAML в код C# и компилировать этот код на лету, когда ей указаны корректные файлы *.targets.

Утилита msbuild.exe представляет собой инструмент, который будет компилировать код .NET, основываясь на инструкциях в XML-сценарии построения. Как обнаруживается, файлы сценариев построения содержат ту же самую разновидность данных, что и файл *.csproj, сгенерированный Visual Studio. Следовательно, программу .NET можно компилировать в командной строке с помощью msbuild.exe или же в самой среде Visual Studio.

На заметку! Полноценное исследование утилиты msbuild.exe выходит за рамки данной главы. Для получения дополнительных сведений выполните поиск "MSBuild" в документации .NET Framework 4.6 SDK.

Ниже приведен очень простой сценарий WpfAppAllXaml.csproj, который содержит достаточный объем информации для информирования msbuild.exe о том, как трансформировать файлы XAML в связанную кодовую базу C#:

```
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <RootNamespace>WpfAppAllXaml</RootNamespace>
    <AssemblyName>WpfAppAllXaml</AssemblyName>
    <OutputType>winexe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="System.XAML" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>
  <ItemGroup>
    <ApplicationDefinition Include="MyApp.xaml" />
    <Page Include="MainWindow.xaml" />
  </ItemGroup>
  <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
</Project>
```

На заметку! Такой файл *.csproj не может быть загружен в IDE-среду Visual Studio, потому что он содержит только минимальный набор инструкций, необходимых для построения приложения в командной строке.

Элемент <PropertyGroup> используется для указания ряда базовых аспектов сборки, таких как корневое пространство имен, имя результирующей сборки и тип вывода (эквивалент параметра /target:winexe компилятора csc.exe).

Первый элемент <ItemGroup> задает набор внешних сборок для ссылки из текущей сборки, которыми, как видно, являются основные сборки WPF, упомянутые ранее в главе. Второй элемент <ItemGroup> более интересен. Обратите внимание, что атрибуту Include элемента <ApplicationDefinition> присвоен файл *.xaml, в котором определяется объект приложения. Атрибут Include элемента <Page> может применяться для перечисления всех остальных файлов *.xaml, где определены окна (и страницы, что часто происходит при построении браузерных приложений XAML), обрабатываемые объектом Application.

Однако "магия" данного сценария построения скрыта в финальном элементе <Import>. Здесь производится ссылка на файл Microsoft.CSharp.targets, который содержит данные для взаимодействия с самим компилятором C#.

Теперь можно открыть окно командной строки разработчика и обработать данные XAML с помощью `msbuild.exe`. Для этого понадобится перейти в каталог с файлами `MainWindow.xaml`, `MyApp.xaml` и `WpfAppAllXaml.csproj` и ввести следующую команду:

```
msbuild WpfAppAllXaml.csproj
```

После завершения процесса сборки в рабочем каталоге обнаружатся подкаталоги `\bin` и `\obj` (точно как в проекте Visual Studio). В папке `\bin\Debug` находится новая сборка .NET по имени `WpfAppAllXaml.exe`. Если открыть ее в `ildasm.exe`, то можно заметить, что разметка XAML была трансформирована в допустимое исполняемое приложение (рис. 26.9.).

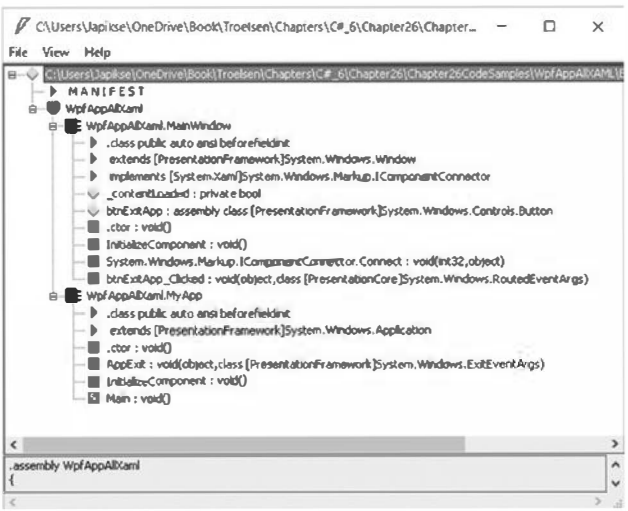


Рис. 26.9. Трансформация разметки XAML в исполняемую сборку .NET

Запустив программу двойным щелчком на исполняемом файле, вы увидите на экране ее главное окно.

Трансформация разметки в сборку .NET

Чтобы понять, *каким образом* разметка была трансформирована в сборку .NET, нужно немного углубиться в процесс `msbuild.exe` и изучить несколько сгенерированных компилятором файлов, в том числе отдельный двоичный ресурс, встроенный в сборку на этапе компиляции. Первая задача связана с исследованием того, как файлы `*.xaml` трансформируются в соответствующую кодовую базу C#.

Отображение разметки XAML окна на код C#

Файлы `*.targets`, указанные в сценарии для `msbuild.exe`, содержат многочисленные инструкции для трансляции элементов XAML в код C#. Когда утилита `msbuild.exe` обрабатывает файл `*.csproj`, она создает два файла `*.g.cs` (где `g` означает auto*generated* (автоматически сгенерированный)), которые сохраняются в папке `\obj\Debug`. На основе имен файлов `*.xaml` результирующие файлы C# получают названия `MainWindow.g.cs` и `MyApp.g.cs`.

Открыв файл `MainWindow.g.cs` в текстовом редакторе, вы найдете там класс по имени `MainWindow`, расширяющий базовый класс `Window`. Имя этого класса является пря-

мым результатом действия атрибута x:Class в открывающем дескрипторе <Window>. Класс также содержит переменную-член типа System.Windows.Controls.Button с именем btnExitApp. В данном случае имя элемента управления основано на значении атрибута x:Name в открывающем объявлении <Button>. Этот класс также содержит обработчик события Click кнопки, btnExitApp_Clicked(). Ниже приведен частичный листинг сгенерированного компилятором файла MainWindow.g.cs:

```
public partial class MainWindow :
  System.Windows.Window, System.Windows.Markup.IComponentConnector
{
  internal System.Windows.Controls.Button btnExitApp;

  private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
  {
    this.Close();
  }
  ...
}
```

В классе MainWindow определена закрытая переменная-член типа bool (по имени _contentLoaded), которая не была напрямую представлена в разметке XAML. Этот член данных используется для того, чтобы гарантировать присваивание содержимого окна только один раз:

```
public partial class MainWindow :
  System.Windows.Window, System.Windows.Markup.IComponentConnector
{
  // Назначение этой переменной-члена поясняется ниже.
  private bool _contentLoaded;
  ...
}
```

Обратите внимание, что сгенерированный компилятором класс также явно реализует интерфейс IComponentConnector из WPF, определенный в пространстве имен System.Windows.Markup. В этом интерфейсе имеется единственный метод Connect(), который реализован для подготовки каждого элемента управления, определенного в разметке, и обеспечения логики событий, как указано в исходном файле MainWindow.xaml. Перед завершением метода Connect() переменная-член _contentLoaded устанавливается в true. Вот как выглядит этот метод:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId,
     object target)
{
  switch (connectionId)
  {
    case 1:
      this.btnExitApp = ((System.Windows.Controls.Button)(target));
      this.btnExitApp.Click += new
        System.Windows.RoutedEventHandler(this.btnExitApp_Clicked);
    return;
  }
  this._contentLoaded = true;
}
```

И, наконец, в классе MainWindow также определен и реализован метод InitializeComponent(). Можно было бы ожидать, что данный метод содержит код, который настраивает внешний вид и поведение каждого элемента управления за счет установки различных свойств (Height, Width, Content и т.д.). Тем не менее, это не так! Как тогда элементы управления получают корректный пользовательский интерфейс?

Логика метода `InitializeComponent()` выясняет местоположение встроенного в сборку ресурса, именованного идентично исходному файлу `*.xaml`:

```
public void InitializeComponent()
{
  if (_contentLoaded)
  {
    return;
  }
  _contentLoaded = true;
  System.Uri resourceLocater = new
    System.Uri("/WpfAppAllXaml;component/mainwindow.xaml",
               System.UriKind.Relative);
  System.Windows.Application.LoadComponent(this, resourceLocater);
}
```

И в этот момент возникает вопрос: *что такое встроенный ресурс?*

Роль BAML

Когда утилита `msbuild.exe` обрабатывает файл `*.csproj`, она генерирует файл с расширением `*.baml`, именованный согласно начальному файлу `MainWindow.xaml`. Таким образом, в папке `\obj\Debug` должен появиться файл под названием `MainWindow.baml` (рис. 26.10).

Name	Date modified	Type	Size
MainWindow.baml	8/5/2015 9:28 PM	BAML File	1 KB
MainWindow.g.cs	8/5/2015 9:28 PM	Visual C# Source fi...	4 KB
MyApp.g.cs	8/5/2015 9:28 PM	Visual C# Source fi...	3 KB
SimpleXamlApp.csproj.FileListAbsolute.txt	8/5/2015 9:14 PM	TXT File	1 KB
SimpleXamlApp.csprojResolveAssembly...	8/5/2015 9:14 PM	CACHE File	2 KB
WpfAppAllXaml.csproj.FileListAbsolute.txt	8/5/2015 9:25 PM	TXT File	2 KB
WpfAppAllXaml.csprojResolveAssembly...	8/5/2015 9:23 PM	CACHE File	2 KB
WpfAppAllXaml.exe	8/5/2015 9:28 PM	Application	7 KB
WpfAppAllXaml.g.resources	8/5/2015 9:28 PM	RESOURCES File	2 KB
WpfAppAllXaml.pdb	8/5/2015 9:28 PM	Program Debug D...	18 KB
WpfAppAllXaml_MarkupCompile.cache	8/5/2015 9:28 PM	CACHE File	1 KB

Рис. 26.10. Файл BAML представляет собой просто компактную двоичную версию файла XAML

Как и можно было предположить, формат BAML (Binary Application Markup Language — двоичный язык разметки приложений) — это компактное двоичное представление исходных данных XAML. Файл `*.baml` встраивается в виде ресурса (через сгенерированный файл `*.g.resources`) в скомпилированную сборку.

Ресурс BAML содержит все данные, необходимые для настройки внешнего вида виджетов пользовательского интерфейса (свойства наподобие `Height` и `Width`).

Здесь важно понимать, что приложение WPF содержит внутри себя двоичное представление (BAML) разметки. Во время выполнения ресурс BAML будет извлечен из контейнера ресурсов и применен для настройки внешнего вида всех окон и элементов управления.

Вдобавок запомните, что имена двоичных ресурсов *идентичны* именам написанных автономных файлов `*.xaml`. Однако это вовсе не означает, что вы должны распространять файлы `*.xaml` вместе со скомпилированной программой WPF. Если только не строится приложение WPF, которое должно динамически загружать и анализировать файлы `*.xaml` во время выполнения, то поставлять исходную разметку никогда не придется.

Отображение разметки XAML приложения на код C#

Последняя порция автоматически сгенерированного кода, которую мы рассмотрим, находится в файле `MyApp.g.cs`. Здесь есть производный от `Application` класс с подходящим методом точки входа `Main()`. В этой реализации `Main()` вызывается метод `InitializeComponent()` на производном от `Application` типе, который, в свою очередь, устанавливает свойство `StartupUri`, позволяя каждому объекту устанавливать корректные настройки свойств на основе двоичного определения XAML:

```
namespace WpfAppAllXaml
{
  public partial class MyApp : System.Windows.Application
  {
    void AppExit(object sender, ExitEventArgs e)
    {
      MessageBox.Show("App has exited");
    }
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    public void InitializeComponent()
    {
      this.Exit += new System.Windows.ExitEventHandler(this.AppExit);
      this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);
    }
    [System.STAThreadAttribute()]
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    public static void Main() {
      WpfAppAllXaml.MyApp app = new WpfAppAllXaml.MyApp();
      app.InitializeComponent();
      app.Run();
    }
  }
}
```

Итоговые замечания о процессе трансформирования XAML в сборку

Итак, к настоящему моменту мы создали полноценную программу WPF с использованием только двух файлов XAML и связанного сценария построения для `msbuild.exe`. Вы видели, что для обработки файлов XAML (и генерации `*.baml`) утилита `msbuild.exe` задействует вспомогательные настройки, определенные внутри файла `*.targets`.

На рис. 26.11 показана общая картина, касающаяся обработки файлов `*.xaml` на этапе компиляции.

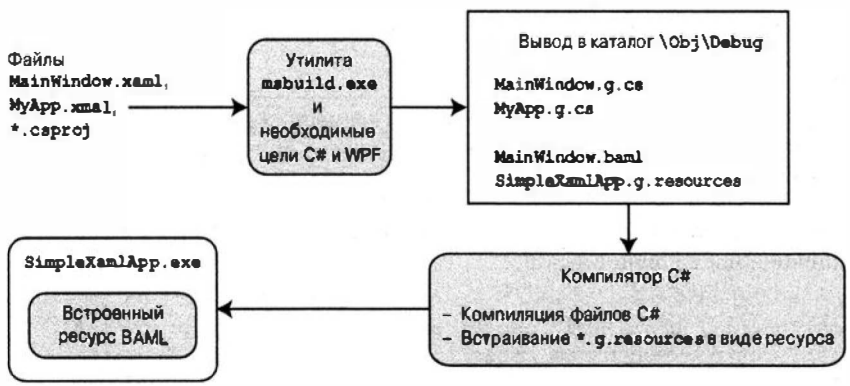


Рис. 26.11. Процесс трансформации XAML в сборку на этапе компиляции

Теперь вы должны лучше представлять, каким образом применяются данные XAML для построения приложения .NET. Далее можно переходить к рассмотрению синтаксиса и семантики самого языка XAML.

Исходный код. Проект `WpfAppAllXaml` доступен в подкаталоге `Chapter_26`.

Синтаксис XAML для WPF

Приложения WPF производственного уровня обычно будут использовать отдельные инструменты для генерации необходимой разметки XAML. Как бы ни были удобны эти инструменты, очень важно понимать общую структуру языка XAML. Для содействия процессу изучения доступен популярный (и бесплатный) инструмент, который позволяет легко экспериментировать с XAML.

Введение в Kaxaml

Если вы только приступаете к изучению грамматики XAML, то очень удобно применять бесплатный инструмент под названием *Kaxaml*. Получить этот популярный редактор/анализатор XAML можно на веб-сайте `http://www.kaxaml.com`.

Редактор Kaxaml полезен тем, что не имеет никакого понятия об исходном коде C#, обработчиках ошибок или логике реализации. Он предлагает намного более простой способ тестирования фрагментов XAML, чем использование полноценного шаблона проекта WPF в Visual Studio. К тому же Kaxaml обладает набором интегрированных инструментов, в том числе средством выбора цвета, диспетчером фрагментов XAML и даже средством "очистки XAML", которое форматирует разметку XAML на основе заданных настроек. Открыв Kaxaml в первый раз, вы найдете в нем простую разметку для элемента управления `<Page>`:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>

  </Grid>
</Page>
```

Подобно `Window` объект `Page` содержит разнообразные диспетчеры компоновки и элементы управления. Тем не менее, в отличие от `Window` объекты `Page` не могут запускаться как отдельные сущности. Взамен они должны помещаться внутрь подходящего хоста, такого как `NavigationWindow`, `Frame` или веб-браузер (и в этом случае просто получается приложение XBAP). Хорошая новость в том, что в элементах `<Page>` и `<Window>` можно вводить идентичную разметку.

На заметку! Если в окне разметки Kaxaml заменить элементы `<Page>` и `</Page>` элементами `<Window>` и `</Window>`, то можно нажать клавишу <F5> и отобразить на экране новое окно.

В качестве начального теста введем следующую разметку в панели XAML, находящейся в нижней части окна Kaxaml:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Кнопка со специальным содержимым -->
```

```
    <Button Height="100" Width="100">
      <Ellipse Fill="Green" Height="50" Width="50"/>
    </Button>
  </Grid>
</Page>
```

В верхней части окна Kaxaml появится визуализированная страница (рис. 26.12).

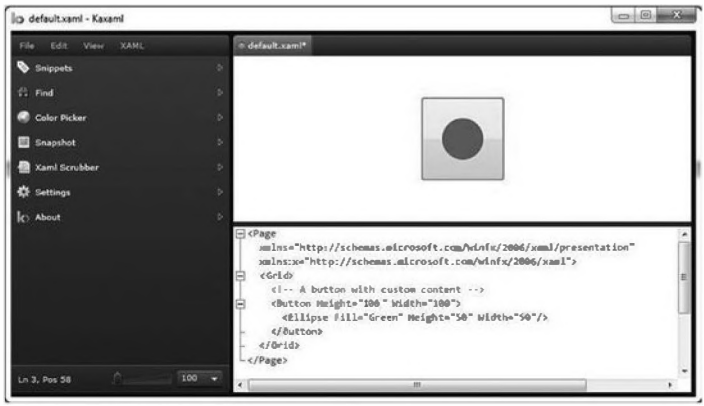


Рис. 26.12. Редактор Kaxaml является очень удобным (и бесплатным) инструментом, применяемым при изучении грамматики XAML

Во время работы с Kaxaml помните, что этот инструмент не позволяет писать разметку, которая влечет за собой любую компиляцию кода (но разрешено использовать `x:Name`). Сюда входит определение атрибута `x:Class` (для указания файла кода), ввод имен обработчиков событий в разметке или применение любых ключевых слов XAML, которые также предусматривают компиляцию кода (вроде `FieldModifier` или `ClassModifier`). Попытка поступить так приводит к ошибке разметки.

Пространства имен XML и "ключевые слова" XAML

Корневой элемент XAML-документа WPF (такой как `<Window>`, `<Page>`, `<UserControl>` или `<Application>`) почти всегда будет ссылаться на два заранее определенных пространства имен XML:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
  </Grid>
</Page>
```

Первое пространство имен XML, `http://schemas.microsoft.com/winfx/2006/xaml/presentation`, отображает множество связанных с WPF пространств имен .NET для использования текущим файлом `*.xaml` (`System.Windows`, `System.Windows.Controls`, `System.Windows.Data`, `System.Windows.Ink`, `System.Windows.Media`, `System.Windows.Navigation` и т.д.).

Это отображение "один ко многим" в действительности жестко закодировано внутри сборок WPF (`WindowsBase.dll`, `PresentationCore.dll` и `PresentationFramework.dll`) с применением атрибута `[XmlnsDefinition]` уровня сборки. Например, если открыть браузер объектов Visual Studio и выбрать сборку `PresentationCore.dll`, то можно увидеть списки, подобные показанному ниже, в котором импортируется пространство имен `System.Windows`:

```
[assembly: XmlnsDefinition(
              "http://schemas.microsoft.com/winfx/2006/xaml/presentation",
              "System.Windows")]
```

Второе пространство имен XML, `http://schemas.microsoft.com/winfx/2006/xaml`, используется для добавления специфичных для XAML "ключевых слов" (термин выбран за неимением лучшего), а также пространства имен `System.Windows.Markup`:

```
[assembly: XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml",
                           "System.Windows.Markup")]
```

Одно из правил любого корректно сформированного документа XML (не забывайте, что грамматика XAML основана на XML) состоит в том, что открывающий корневой элемент назначает одно пространство имен XML в качестве *первичного пространства имен*, которое обычно представляет собой пространство имен, содержащее самые часто применяемые элементы. Если корневой элемент требует включения дополнительных вторичных пространств имен (как видно здесь), то они должны быть определены с использованием уникального префикса (чтобы устранить возможные конфликты имен). По соглашению для префикса применяется просто `x`, однако он может быть любым уникальным маркером, таким как `XamlSpecificStuff`:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:XamlSpecificStuff="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Кнопка со специальным содержимым -->
    <Button XamlSpecificStuff:Name="button1" Height="100" Width="100">
      <Ellipse Fill="Green" Height="50" Width="50"/>
    </Button>
  </Grid>
</Page>
```

Очевидный недостаток определения длинных префиксов для пространств имен XML связан с тем, что `XamlSpecificStuff` придется набирать всякий раз, когда в файле XAML нужно сослаться на один из элементов, определенных в этом пространстве имен XML. Из-за того, что префикс `XamlSpecificStuff` намного длиннее, давайте ограничимся `x`.

Помимо ключевых слов `x:Name`, `x:Class` и `x:Code` пространство имен `http://schemas.microsoft.com/winfx/2006/xaml` также предоставляет доступ к дополнительным ключевым словам XAML, наиболее распространенные из которых кратко описаны в табл. 26.9.

Таблица 26.9. Ключевые слова XAML

Ключевое слово XAML	Описание
`x:Array`	Представляет в XAML тип массива .NET
`x:ClassModifier`	Позволяет определять видимость класса C# (`internal` или `public`), обозначенного ключевым словом `Class`
`x:FieldModifier`	Позволяет определять видимость члена типа (`internal`, `public`, `private` или `protected`) для любого именованного подэлемента корня (например, `<Button>` внутри элемента `<Window>`). *Именованный элемент* определяется с использованием ключевого слова `Name` в XAML
`x:Key`	Позволяет устанавливать значение ключа для элемента XAML, которое будет помещено в элемент словаря
`x:Name`	Позволяет указывать сгенерированное имя C# заданного элемента XAML

Окончание табл. 26.9

Ключевое слово XAML	Описание
`x:Null`	Представляет ссылку `null`
`x:Static`	Позволяет ссылаться на статический член типа
`x:Type`	Эквивалент XAML операции `typeof` языка C# (она будет выдавать объект `System.Type` на основе предоставленного имени)
`x:TypeArguments`	Позволяет устанавливать элемент как обобщенный тип с определенным параметром типа (например, `List<int>` или `List<bool>`)

В дополнение к двум указанным объявлениям пространств имен XML можно, а иногда и нужно, определить дополнительные префиксы дескрипторов в открывающем элементе документа XAML. Обычно так поступают, когда необходимо описать в XAML класс .NET, определенный во внешней сборке.

Например, предположим, что было построено несколько специальных элементов управления WPF, которые упакованы в библиотеку по имени `MyControls.dll`. Если теперь требуется создать новый объект `Window`, в котором применяются созданные элементы, то можно установить специальное пространство имен XML, отображаемое на библиотеку `MyControls.dll`, с использованием маркеров `clr-namespace` и `assembly`. Ниже приведен пример разметки, создающей префикс дескриптора по имени `myCtrls`, который может применяться для доступа к элементам управления в этой библиотеке:

```
<Window x:Class="WpfApplication1.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:myCtrls="clr-namespace:MyControls;assembly=MyControls"
  Title="MainWindow" Height="350" Width="525">
  <Grid>
    <myCtrls:MyCustomControl />
  </Grid>
</Window>
```

Маркеру `clr-namespace` назначается название пространства имен .NET в сборке, в то время как маркер `assembly` устанавливается в дружественное имя внешней сборки `*.dll`. Такой синтаксис можно использовать для любой внешней библиотеки .NET, которой желательно манипулировать внутри разметки. В настоящее время в этом нет необходимости, но в последующих главах понадобится определять специальные объявления пространств имен XML для описания типов в разметке.

На заметку! Если нужно определить в разметке класс, который является частью текущей сборки, но находится в другом пространстве имен .NET, то префикс дескриптора `xmlns` определяется без атрибута `assembly=`:

```
xmlns:myCtrls="clr-namespace:SomeNamespaceInMyApp"
```

Управление видимостью классов и переменных-членов

Многие ключевые слова вы увидите в действии там, где они потребуются в последующих главах; тем не менее, в качестве простого примера взгляните на следующее XAML-определение `<Window>`, в котором применяются ключевые слова `ClassModifier` и `FieldModifier`, а также `x:Name` и `x:Class` (вспомните, что редактор Kaxaml не позволяет использовать ключевые слова XAML, вовлекающие компиляцию, такие как `x:Code`, `x:FieldModifier` или `x:ClassModifier`):

```
<!-- Этот класс теперь будет объявлен как internal в файле *.g.cs -->
<Window x:Class="MyWPFApp.MainWindow" x:ClassModifier ="internal"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <!-- Эта кнопка будет объявлена как public в файле *.g.cs -->
  <Button x:Name ="myButton" x:FieldModifier ="public" Content = "OK"/>
</Window>
```

По умолчанию все определения типов C#/XAML являются открытыми (`public`), а члены — внутренними (`internal`). Однако на основе показанного определения XAML результирующий автоматически сгенерированный файл содержит внутренний тип класса с открытой переменной-членом `Button`:

```
internal partial class MainWindow : System.Windows.Window,
  System.Windows.Markup.IComponentConnector
{
  public System.Windows.Controls.Button myButton;
  ...
}
```

Элементы XAML, атрибуты XAML и преобразователи типов

После установки корневого элемента и необходимых пространств имен XML следующая задача заключается в наполнении корня *дочерним элементом*. В реальном приложении WPF дочерним элементом будет диспетчер компоновки (такой как `Grid` или `StackPanel`), который, в свою очередь, содержит любое количество дополнительных элементов, описывающих пользовательский интерфейс. Эти диспетчеры компоновки рассматриваются в следующей главе, а пока предположим, что элемент `<Window>` будет содержать единственный элемент `Button`.

Как было показано ранее в главе, *элементы* XAML отображаются на типы классов или структур внутри заданного пространства имен .NET, тогда как *атрибуты* в открывающем дескрипторе элемента отображаются на свойства или события конкретного типа. В целях иллюстрации введем в редакторе Kaxaml следующее определение `<Button>`:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Сконфигурировать внешний вид элемента Button -->
    <Button Height="50" Width="100" Content="OK!"
            FontSize="20" Background="Green" Foreground="Yellow"/>
  </Grid>
</Page>
```

Обратите внимание, что значения, присвоенные свойствам, представлены с помощью простого текста. Это может выглядеть как полное несоответствие типам данных, поскольку после создания такого элемента `Button` в коде C# данным свойствам будут присваиваться *не* строковые объекты, а значения специфических типов данных. Например, ниже показано, как та же самая кнопка описана в коде:

```
public void MakeAButton()
{
  Button myBtn = new Button();
  myBtn.Height = 50;
  myBtn.Width = 100;
  myBtn.FontSize = 20;
```

```
  myBtn.Content = "OK!";
  myBtn.Background = new SolidColorBrush(Colors.Green);
  myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}
```

Оказывается, что инфраструктура WPF поставляется с несколькими классами *преобразователей типов*, которые будут применяться для трансформации простых текстовых значений в корректные типы данных. Такой процесс происходит прозрачно (и автоматически).

Тем не менее, нередко возникает потребность в присваивании атрибуту XAML намного более сложного значения, которое невозможно выразить посредством простой строки. Например, пусть необходимо построить специальную кисть для установки свойства `Background` элемента `Button`. Создать такую кисть в коде довольно просто:

```
public void MakeAButton()
{
  ...
  // Необычная кисть для фона.
  LinearGradientBrush fancyBruch =
    new LinearGradientBrush(Colors.DarkGreen, Colors.LightGreen, 45);
  myBtn.Background = fancyBruch;
  myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}
```

Но можно ли представить эту сложную кисть в виде строки? Нет, нельзя! К счастью, в XAML предусмотрен специальный синтаксис, который можно использовать всякий раз, когда нужно присвоить сложный объект в качестве значения свойства; он называется *синтаксисом "свойство-элемент"*.

Понятие синтаксиса "свойство-элемент" в XAML

Синтаксис "свойство-элемент" позволяет присваивать свойству сложные объекты. Ниже показано описание XAML элемента `Button`, в котором для установки свойства `Background` применяется объект `LinearGradientBrush`:

```
<Button Height="50" Width="100" Content="OK!"
        FontSize="20" Foreground="Yellow">
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Color="DarkGreen" Offset="0"/>
      <GradientStop Color="LightGreen" Offset="1"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

Обратите внимание, что внутри дескрипторов `<Button>` и `</Button>` определена вложенная область по имени `<Button.Background>`, а в ней — специальный элемент `<LinearGradientBrush>`. (Пока не беспокойтесь о коде кисти; вы освоите графику WPF в главе 28.)

Вообще говоря, любое свойство может быть установлено с использованием синтаксиса "свойство-элемент", который всегда сводится к следующему шаблону:

```
<ОпределяющийКласс>
  <ОпределяющийКласс.СвойствоОпределяющегоКласса>
    <!-- Значение для свойства определяющего класса -->
  </ОпределяющийКласс.СвойствоОпределяющегоКласса>
</ОпределяющийКласс>
```

Хотя любое свойство *может* быть установлено с применением такого синтаксиса, указание значения в виде простой строки, когда это возможно, будет экономить время ввода. Например, вот намного более многословный способ установки свойства `Width` элемента `Button`:

```
<Button Height="50" Content="OK!"
        FontSize="20" Foreground="Yellow">
  ...
  <Button.Width>
    100
  </Button.Width>
</Button>
```

Понятие присоединяемых свойств XAML

В дополнение к синтаксису "свойство-элемент" в XAML поддерживается специальный синтаксис, используемый для установки значения *присоединяемого свойства*. По существу присоединяемое свойство позволяет дочернему элементу устанавливать значение свойства, которое на самом деле определено в родительском элементе. Общий шаблон, которому нужно следовать, выглядит так:

```
<РодительскийЭлемент>
  <ДочернийЭлемент РодительскийЭлемент.СвойствоРодительскогоЭлемента =
     "Значение">
</РодительскийЭлемент>
```

Самое распространенное применение синтаксиса присоединяемых свойств связано с позиционированием элементов пользовательского интерфейса внутри одного из классов диспетчеров компоновки (`Grid`, `DockPanel` и т.д.). Диспетчеры компоновки более подробно рассматриваются в следующей главе, а пока введем в редакторе Kaxaml следующую разметку:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Canvas Height="200" Width="200" Background="LightBlue">
    <Ellipse Canvas.Top="40" Canvas.Left="40" Height="20" Width="20"
             Fill="DarkBlue"/>
  </Canvas>
</Page>
```

Здесь определен диспетчер компоновки `Canvas`, который содержит элемент `Ellipse`. Обратите внимание, что с помощью синтаксиса присоединяемых свойств элемент `Ellipse` способен информировать свой родительский элемент (`Canvas`) о том, где располагать позицию его левого верхнего угла.

В отношении присоединяемых свойств следует иметь в виду несколько моментов. Прежде всего, это не универсальный синтаксис, который может применяться к *любому* свойству *любого* родительского элемента. Скажем, приведенная далее разметка XAML содержит ошибку:

```
<!-- Попытка установки свойства Background в Canvas через присоединяемое
свойство. Ошибка! -->
<Canvas Height="200" Width="200">
  <Ellipse Canvas.Background="LightBlue"
           Canvas.Top="40" Canvas.Left="90"
           Height="20" Width="20" Fill="DarkBlue"/>
</Canvas>
```

В действительности присоединяемые свойства являются специализированной формой специфичной для WPF концепции, которая называется *свойством зависимости*. Если только свойство не было реализовано в весьма специальной манере, то его значение не может быть установлено с использованием синтаксиса присоединяемых свойств. Свойства зависимости подробно исследуются в главе 27.

На заметку! Инструменты Kaxaml, Visual Studio и Expression Blend (бесплатный инструмент, сопровождающий Visual Studio 2015 и удобный для редактирования основанных на XAML приложений) имеют средство IntelliSense, которое отображает допустимые присоединяемые свойства, доступные для установки заданным элементом.

Понятие расширений разметки XAML

Как уже объяснялось, значения свойств чаще всего представляются в виде простой строки или через синтаксис "свойство-элемент". Однако существует еще один способ указать значение атрибута XAML — применение *расширений разметки*. Расширения разметки позволяют анализатору XAML получать значение для свойства из выделенного внешнего класса. Это может обеспечить большие преимущества, поскольку для получения значений некоторых свойств требуется выполнение множества операторов кода.

Расширения разметки предлагают способ аккуратного расширения грамматики XAML новой функциональностью. Расширение разметки внутренне представлено как класс, производный от `MarkupExtension`. Следует отметить, что необходимость в построении специального расширения разметки возникает крайне редко. Тем не менее, некоторые ключевые слова XAML (вроде `x:Array`, `x:Null`, `x:Static` и `x:Type`) являются замаскированными расширениями разметки!

Расширение разметки помещается между фигурными скобками:

```
<Элемент УстанавливаемоеСвойство = "{РасширениеРазметки}"/>
```

Чтобы увидеть расширение разметки в действии, введем в редакторе Kaxaml следующий код:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:CorLib="clr-namespace:System;assembly=mscorlib">
  <StackPanel>
    <!-- Расширение разметки Static позволяет получать значение
         статического члена класса -->
    <Label Content ="{x:Static CorLib:Environment.OSVersion}"/>
    <Label Content ="{x:Static CorLib:Environment.ProcessorCount}"/>
    <!-- Расширение разметки Type - это версия XAML операции typeof языка C# -->
    <Label Content ="{x:Type Button}" />
    <Label Content ="{x:Type CorLib:Boolean}" />
    <!-- Наполнение элемента ListBox массивом строк -->
    <ListBox Width="200" Height="50">
      <ListBox.ItemsSource>
        <x:Array Type="CorLib:String">
          <CorLib:String>Sun Kil Moon</CorLib:String>
          <CorLib:String>Red House Painters</CorLib:String>
          <CorLib:String>Besnard Lakes</CorLib:String>
        </x:Array>
      </ListBox.ItemsSource>
    </ListBox>
  </StackPanel>
</Page>
```

Прежде всего, обратите внимание, что определение `<Page>` содержит новое объявление пространства имен XML, которое позволяет получать доступ к пространству имен `System` сборки `mscorlib.dll`. Имея это пространство имен XML, сначала с помощью расширения разметки `x:Static` извлекаются значения свойств `OSVersion` и `ProcessorCount` класса `System.Environment`.

Расширение разметки `x:Type` обеспечивает доступ к описанию метаданных указанного элемента. Здесь просто содержимому элементов `Label` присваиваются полностью заданные имена типов `Button` и `System.Boolean` из WPF.

Наиболее интересная часть показанной выше разметки связана с элементом `ListBox`. Его свойство `ItemSource` устанавливается в массив строк, полностью объявленный в разметке. Взгляните, каким образом расширение разметки `x:Array` позволяет указывать набор подэлементов внутри своей области действия:

```
<x:Array Type="CorLib:String">
  <CorLib:String>Sun Kil Moon</CorLib:String>
  <CorLib:String>Red House Painters</CorLib:String>
  <CorLib:String>Besnard Lakes</CorLib:String>
</x:Array>
```

На заметку! Предыдущий пример XAML служит только для иллюстрации расширения разметки в действии. Как будет показано в главе 27, существуют намного более простые способы наполнения элементов управления `ListBox`.

На рис. 26.13 показана разметка этого элемента `<Page>` в редакторе Kaxaml.

Вы уже видели многочисленные примеры, которые демонстрировали основные аспекты синтаксиса XAML. Вы наверняка согласитесь, что XAML очень интересен своей возможностью описывать деревья объектов .NET в декларативной манере. Хотя это исключительно полезно при конфигурировании графических пользовательских интерфейсов, не забывайте о том, что с помощью XAML можно описывать *любой* тип из *любой* сборки при условии, что он является неабстрактным и содержит стандартный конструктор.

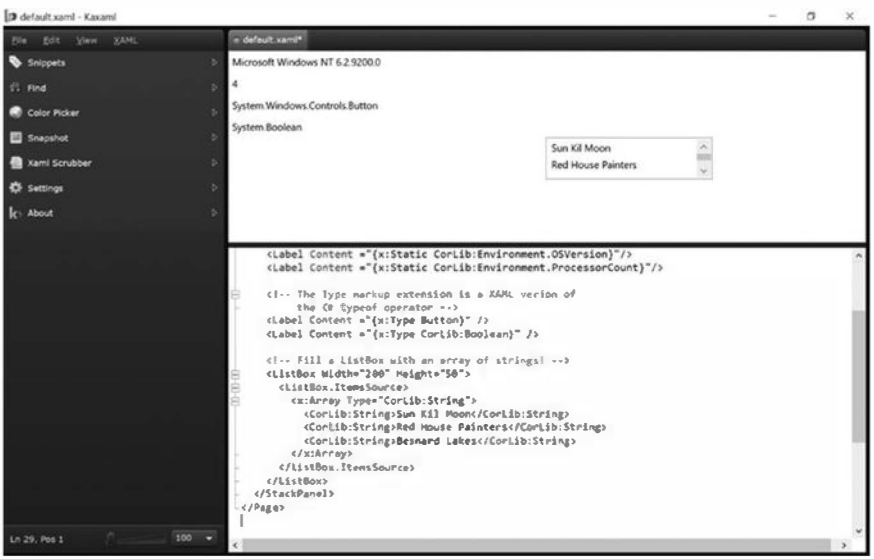


Рис. 26.13. Расширения разметки позволяют устанавливать значения через функциональность выделенного класса

Построение приложения WPF с использованием файлов отделенного кода

Первые два примера в главе отражали крайние случаи разработки приложения WPF, при которых применялся только код C# или только разметка XAML. Однако рекомендованный способ построения любого приложения WPF предусматривает использование подхода с *файлами кода*. В такой модели файлы XAML проекта содержат только разметку, которая описывает общее состояние классов, в то время как файлы кода предоставляют детали реализации.

Добавление файла кода для класса `MainWindow`

Для иллюстрации рассматриваемого подхода мы модифицируем пример `WpfAppAllXaml` с целью применения файлов кода. Скопируем всю папку предыдущего примера в новую папку под названием `WpfAppCodeFiles`. Создадим в этой папке новый файл кода C# по имени `MainWindow.xaml.cs` (по соглашению имя файла отделенного кода C# имеет форму `*.xaml.cs`). Поместим в файл `MainWindow.xaml.cs` показанный ниже код:

```
// MainWindow.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;

namespace WpfAppAllXaml
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      // Обратите внимание, что этот метод определен внутри
      // сгенерированного файла MainWindow.g.cs.
      InitializeComponent();
    }

    private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
    {
      this.Close();
    }
  }
}
```

Здесь определен частичный класс, содержащий логику обработки событий, которая будет объединена с определением частичного класса того же самого типа в файле `*.g.cs`. Учитывая, что метод `InitializeComponent()` определен внутри файла `MainWindow.g.cs`, конструктор окна вызывает его для загрузки и обработки встроенного ресурса BAML.

Файл `MainWindow.xaml` также понадобится изменить; это означает просто удаление в нем всех следов кода C#, которые присутствовали ранее:

```
<Window x:Class="WpfAppAllXaml.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="A Window built using Code Files!"
  Height="200" Width="300"
  WindowStartupLocation ="CenterScreen">
```

```
  <Window.Content>
    <!-- Обработчик событий теперь находится в файле кода -->
    <Button x:Name="btnExitApp" Width="133" Height="24"
            Content = "Close Window" Click ="btnExitApp_Clicked"/>
  </Window.Content>
</Window>
```

Добавление файла кода для класса MyApp

При желании можно было бы также построить файл отделенного кода для типа, производного от Application. Поскольку большинство действий происходит в файле MyApp.g.cs, код внутри файла MyApp.xaml.cs довольно прост:

```
// MyApp.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;
namespace WpfAppAllXaml
{
  public partial class MyApp : Application
  {
    private void AppExit(object sender, ExitEventArgs e)
    {
      MessageBox.Show("App has exited");
    }
  }
}
```

Содержимое файла MyApp.xaml теперь выглядит следующим образом:

```
<Application x:Class="WpfAppAllXaml.MyApp"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml"
  Exit ="AppExit">
</Application>
```

Обработка файлов кода с помощью msbuild.exe

Перед компиляцией файлов с использованием msbuild.exe необходимо обновить файл *.csproj, чтобы посредством элементов <Compile> (которые здесь выделены полужирным) включить в процесс компиляции новые файлы C#:

```
<Project DefaultTargets="Build" xmlns=
  "http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <RootNamespace>WpfAppAllXaml</RootNamespace>
    <AssemblyName>WpfAppAllXaml</AssemblyName>
    <OutputType>winexe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
    <Reference Include="System.Xaml" />
  </ItemGroup>
  <ItemGroup>
```

```
    <ApplicationDefinition Include="MyApp.xaml" />
    <Compile Include = "MainWindow.xaml.cs" />
    <Compile Include = "MyApp.xaml.cs" />
    <Page Include="MainWindow.xaml" />
  </ItemGroup>
  <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
</Project>
```

После передачи сценария сборки утилите `msbuild.exe` с помощью такой команды:

```
msbuild WpfAppAllXaml.csproj
```

получается та же самая исполняемая сборка, что и в приложении `WpfAppAllXaml` (в папке `\bin\Debug`). Тем не менее, если речь идет о разработке, то теперь имеется четкое отделение представления (XAML) от программной логики (C#).

Поскольку это предпочтительный метод разработки, в приложениях WPF, создаваемых с применением Visual Studio (или Expression Blend), всегда используется модель отделенного кода.

Исходный код. Проект `WpfAppCodeFiles` доступен в подкаталоге `Chapter_26`.

Построение приложений WPF с использованием Visual Studio

На протяжении настоящей главы примеры создавались с применением простых текстовых редакторов, компилятора командной строки и редактора Kaxaml. Причиной было желание сосредоточиться на основном синтаксисе приложений WPF, не отвлекаясь на дополнительные украшения, которыми изобилует графический конструктор. Теперь, когда вы видели процесс построения приложений WPF с нуля, давайте посмотрим, каким образом Visual Studio может упростить их создание.

На заметку! Ниже представлены основные особенности использования Visual Studio для построения приложений WPF. В последующих главах при необходимости будут иллюстрироваться дополнительные аспекты этой IDE-среды.

Шаблоны проектов WPF

В диалоговом окне New Project (Новый проект) среды Visual Studio определен набор рабочих пространств проектов WPF, которые расположены в узле Windows корневого узла Visual C#. Здесь на выбор доступны следующие варианты: WPF Application (Приложение WPF), WPF User Control Library (Библиотека пользовательских элементов управления WPF), WPF Custom Control Library (Библиотека специальных элементов управления WPF) и WPF Browser Application (Браузерное приложение WPF), т.е. приложение XBAP. Для начала создадим новый проект приложения WPF по имени `WpfTesterApp` (рис. 26.14).

Кроме установки ссылок на все сборки WPF (`PresentationCore.dll`, `PresentationFramework.dll`, `System.Xaml.dll` и `WindowsBase.dll`) вы также получите начальные классы, производные от `Window` и `Application`, каждый из которых представлен с применением XAML и файла кода C#. На рис. 26.15 показано окно Solution Explorer для нового проекта WPF.

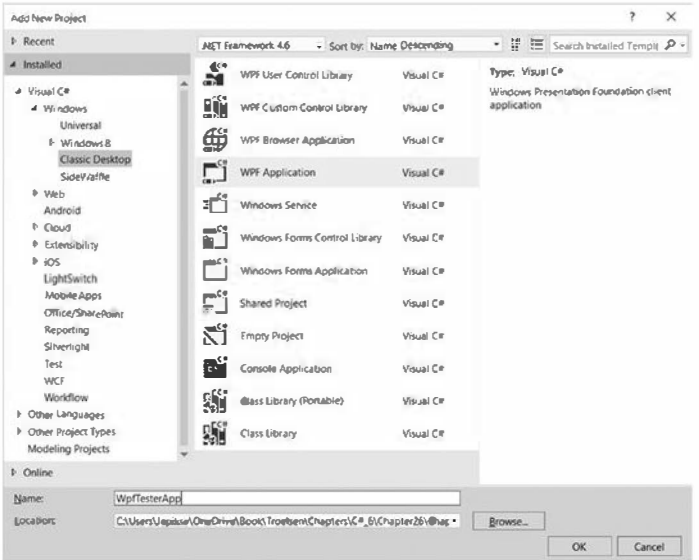


Рис. 26.14. Шаблоны проектов WPF в Visual Studio находятся в узле Windows

Рис. 26.15. Начальные файлы проекта типа WPF Application

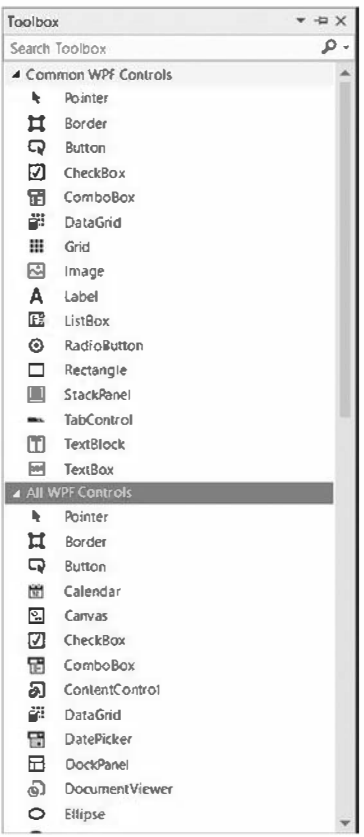


Рис. 26.16. Панель инструментов содержит элементы управления WPF, которые могут быть помещены на поверхность визуального конструктора

Панель инструментов и визуальный конструктор/редактор XAML

В Visual Studio имеется панель инструментов (открываемая через меню View (Вид)), которая содержит многочисленные элементы управления WPF (рис. 26.16).

Используя стандартную операцию перетаскивания с помощью мыши, любой из этих элементов управления можно поместить на поверхность визуального конструктора элемента `Window` или перетащить его на область редактора разметки XAML в нижней части окна визуального конструктора. Начальная разметка XAML будет сгенерирована автоматически. Давайте перетащим посредством мыши элементы управления `Button` и `Calendar` на поверхность визуального конструктора. Обратите внимание на возможность изменения позиции и размера элементов управления (просмотрите результирующую разметку XAML, генерируемую на основе изменений).

В дополнение к построению пользовательского интерфейса с помощью мыши и панели инструментов разметку можно также вводить вручную с применением интегрированного редактора XAML. Как показано на рис. 26.17, здесь обеспечивается поддержка средства IntelliSense, что помогает упростить написание разметки. Для примера можно добавить свойство `Background` в открывающий элемент `<Window>`.

Уделите некоторое время на добавление значений свойств напрямую в редакторе XAML. Обязательно освойте этот аспект визуального конструктора WPF.

Установка свойств с использованием окна Properties

После помещения нескольких элементов управления на поверхность визуального конструктора (или определения их в редакторе вручную) можно открыть окно Properties (Свойства) для установки значений свойств выделенного элемента управления, а также для создания связанных с ним обработчиков событий. В качестве простого примера выберем в визуальном конструкторе ранее добавленный элемент управления `Button`. С применением окна Properties изменим цвет в свойстве `Background` элемента `Button`, используя встроенный редактор кистей (рис. 26.18); редактор кистей более подробно рассматривается в главе 28 во время исследования графики WPF.

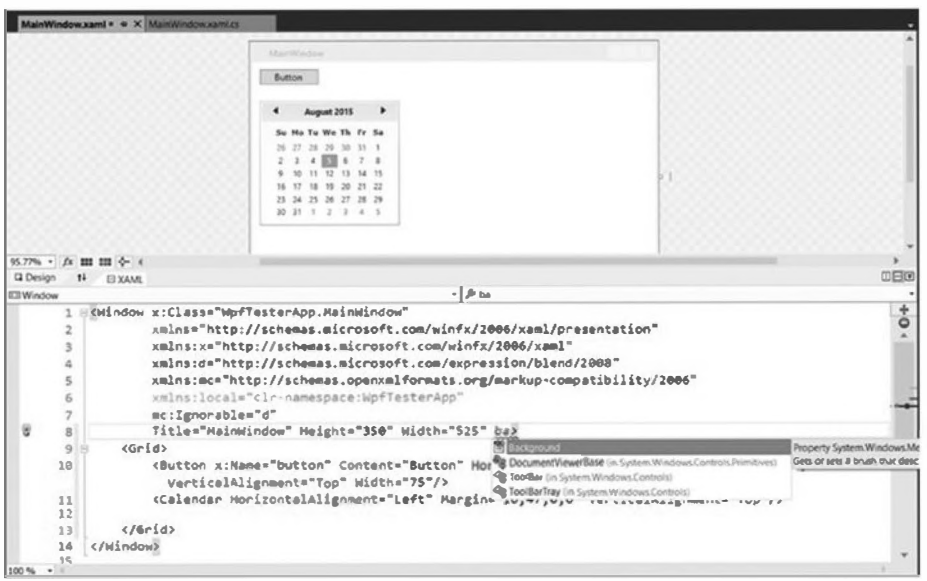


Рис. 26.17. Визуальный конструктор элемента `Window`

На заметку! В верхней части окна Properties имеется текстовая область, предназначенная для поиска. Чтобы быстро найти интересующий элемент, введите имя свойства, которое требуется установить.

После завершения работы с редактором кистей имеет смысл взглянуть на сгенерированную разметку, которая может выглядеть так:

```
<Button x:Name="button" Content="Button" HorizontalAlignment="Left"
        Margin="10,10,0,0" VerticalAlignment="Top" Width="75">
  <Button.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="#FF80EB4F" Offset="0"/>
      <GradientStop Color="#FFCE3058" Offset="1"/>
      <GradientStop Color="#FF8293DD" Offset="0.5"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

Обработка событий с использованием окна Properties

Для организации обработки событий, связанных с определенным элементом управления, также можно применять окно Properties, но на этот раз понадобится щелкнуть на кнопке Events (События), расположенной в верхней правой части окна (кнопка с изображением молнии). Выберем кнопку на поверхности визуального конструктора, если она еще не выбрана, щелкнем на кнопке Events в окне Properties и дважды щелкнем на поле для события `Click`.

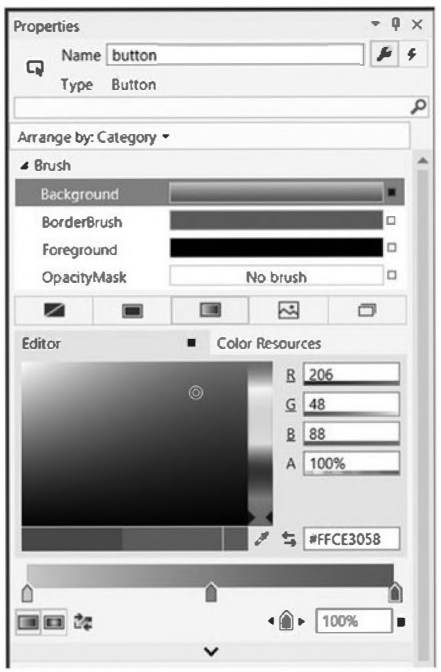


Рис. 26.18. Окно Properties может использоваться для конфигурирования пользовательского интерфейса элемента управления WPF

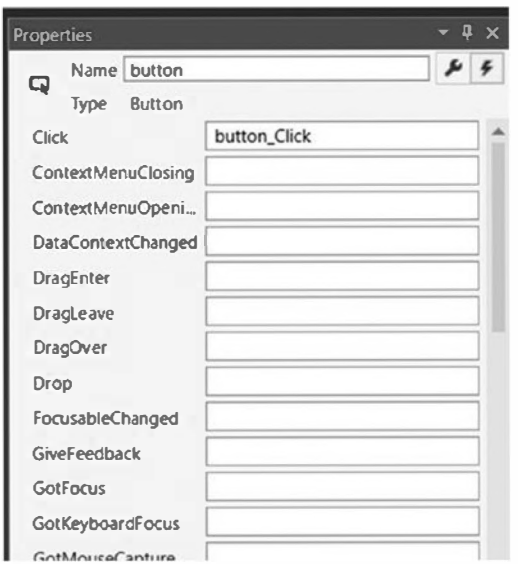


Рис. 26.19. Обработка событий с использованием окна Properties

Среда Visual Studio автоматически построит обработчик событий, имя которого имеет следующую общую форму:

ИмяЭлементаУправления_ИмяСобытия

Так как кнопка не была переименована, в окне Properties отображается сгенерированный обработчик событий по имени `button_Click` (рис. 26.19).

Кроме того, Visual Studio сгенерирует соответствующий обработчик события C# в файле кода для окна. В него можно поместить любой код, который должен выполняться, когда на кнопке произведен щелчок.

В качестве простого примера добавим следующий оператор кода:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
  }

  private void button_Click(object sender, RoutedEventArgs e)
  {
    MessageBox.Show("You clicked the button!");
  }
}
```

Обработка событий в редакторе XAML

Обрабатывать события можно также непосредственно в редакторе XAML. Для примера поместим курсор мыши внутрь элемента `<Window>` и введем имя события `MouseMove`, а за ним знак равенства. Среда Visual Studio отобразит все совместимые обработчики из файла кода (если они существуют), а также пункт Create method (Создать метод), как показано на рис. 26.20.

Позволим IDE-среде создать обработчик события `MouseMove`, введем следующий код и запустим приложение, чтобы увидеть конечный результат:

```
private void MainWindow_MouseMove (object sender, MouseEventArgs e)
{
  this.Title = e.GetPosition(this).ToString();
}
```

Рис. 26.20. Обработка событий с применением редактора XAML

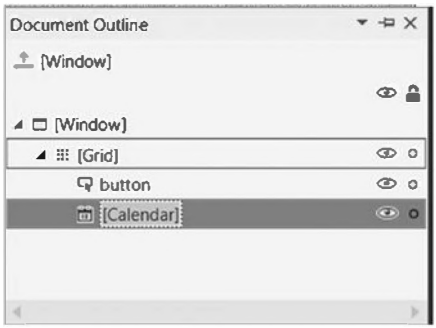


Рис. 26.21. Визуализация разметки XAML в окне Document Outline

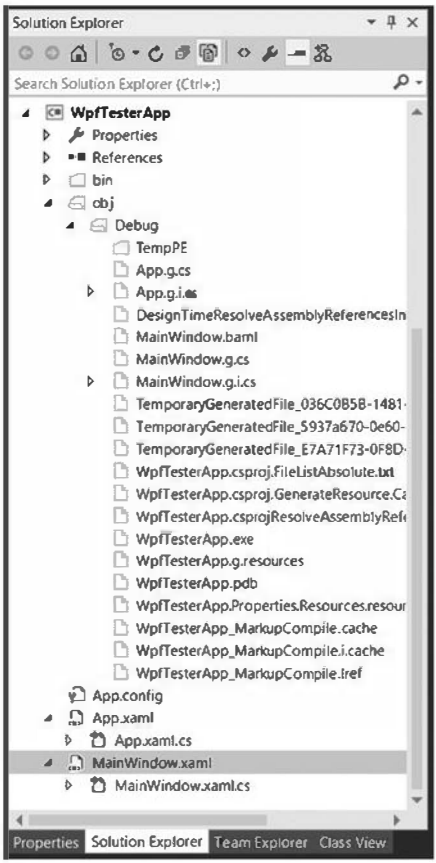


Рис. 26.22. Просмотр выходных файлов проекта WPF в окне Solution Explorer

Окно Document Outline

Во время работы с любым основанным на XAML проектом (приложение WPF, Silverlight, Windows Phone/Windows 10 Mobile или Windows 10 Application) вы определенно будете использовать значительный объем разметки для представления пользовательского интерфейса. Когда вы начнете сталкиваться с более сложной разметкой XAML, может оказаться удобной визуализация разметки для быстрого выбора элементов с целью редактирования в визуальном конструкторе Visual Studio.

В настоящее время наша разметка довольно проста, т.к. было определено лишь несколько элементов управления внутри начального элемента `<Grid>`. Тем не менее, необходимо найти окно Document Outline (Схема документа), которое по умолчанию располагается в нижней левой части окна IDE-среды (если обнаружить его не удается, это окно можно открыть через пункт меню View⇨Other Windows (Вид⇨Другие окна)). При активном окне визуального конструктора XAML (не окне с файлом кода C#) в IDE-среде можно заметить, что в окне Document Outline отображаются вложенные элементы (рис. 26.21).

Этот инструмент также предоставляет способ временного сокрытия заданного элемента (или набора элементов) на поверхности визуального конструктора, а также блокировки элементов с целью предотвращения их дальнейшего редактирования.

В следующей главе вы увидите, что окно Document Outline предоставляет много других возможностей для группирования выбранных элементов внутри новых диспетчеров компоновки (помимо прочих средств).

Просмотр автоматически сгенерированных файлов кода

Перед построением последнего примера главы щелкнем на кнопке Show All Files (Показать все файлы) в окне Solution Explorer (рис. 26.22). Обратите внимание на присутствие файлов `*.baml` и `*.g.cs` (в папке `\obj\Debug`). Добавлять свой код в эти автоматически сгенерированные файлы не рекомендуется, а предыдущие примеры главы должны были прояснить, как обрабатывается разметка XAML.

Построение специального редактора XAML с помощью Visual Studio

Теперь, когда вы ознакомились с базовыми инструментами, применяемыми внутри Visual Studio для проектирования окна WPF, в финальном примере главы будет показано, как построить приложение, которое позволит манипулировать разметкой XAML во время выполнения. Закроем текущий проект и создадим новый проект типа WPF Application по имени `MyXamlPad`. По завершении это приложение будет функционировать аналогично Kaxaml, но без разнообразных украшений. В частности, оно даст возможность вводить правильно сформированную разметку и щелкать на кнопке для динамического преобразования XAML в новый объект `Window`.

Проектирование графического пользовательского интерфейса окна

Инфраструктура WPF поддерживает возможность загрузки, разбора и сохранения описаний XAML программным образом. Это может быть полезно во многих ситуациях. Например, предположим, что есть пять разных файлов XAML, которые описывают внешний вид и поведение объекта `Window`. До тех пор, пока имена каждого элемента управления (и всех необходимых обработчиков событий) идентичны внутри каждого файла, к объекту `Window` можно динамически применять "обложки" (возможно на основе аргумента, передаваемого приложению при запуске).

Взаимодействие с разметкой XAML во время выполнения вращается вокруг типов `XamlReader` и `XamlWriter`, которые определены в пространстве имен `System.Windows.Markup`. Чтобы проиллюстрировать, как программно наполнить объект `Window` из внешнего файла `*.xaml`, создадим приложение, которое имитирует базовую функциональность редактора Kaxaml.

На заметку! Классы `XamlReader` и `XamlWriter` предлагают основную функциональность для манипулирования разметкой XAML во время выполнения. Если когда-нибудь понадобится получить полный контроль над объектной моделью XAML, то придется изучить сборку `System.Xaml.dll`.

Хотя наше приложение определенно не будет настолько полнофункциональным, как Kaxaml, все же оно предоставит возможность вводить допустимую разметку XAML, просматривать результаты и сохранять разметку XAML во внешнем файле. Для начала изменим первоначальное определение XAML элемента `<Window>`, как показано ниже (в этой точке разметку XAML можно ввести вручную, но для генерации обработчиков событий необходимо использовать IDE-среду, как было показано ранее).

На заметку! В следующей главе мы погрузимся в детали работы с элементами управления и панелями, так что пока не беспокойтесь об аспектах, связанных с объявлениями элементов управления.

```
<Window x:Class="MyXamlPad.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="My Custom XAML Editor"
  Height="338" Width="1041"
  Loaded="Window_Loaded" Closed="Window_Closed"
  WindowStartupLocation="CenterScreen">
```

```
  <!-- Использовать DockPanel, а не Grid -->
  <DockPanel LastChildFill="True" >
    <!-- Эта кнопка запустит окно с определенной разметкой XAML -->
    <Button DockPanel.Dock="Top" Name = "btnViewXaml" Width="100" Height="40"
          Content ="View Xaml" Click="btnViewXaml_Click" />
    <!-- Это будет область для ввода -->
    <TextBox AcceptsReturn ="True" Name ="txtXamlData"
             FontSize ="14" Background="Black" Foreground="Yellow"
             BorderBrush ="Blue" VerticalScrollBarVisibility="Auto"
             AcceptsTab="True"/>
  </DockPanel>
</Window>
```

Прежде всего, обратите внимание, что первоначальный элемент `<Grid>` заменен диспетчером компоновки `<DockPanel>`, содержащим элементы `Button` (по имени `btnViewXaml`) и `TextBox` (по имени `txtXamlData`), а также на обработку события `Click` элемента `Button`.

Кроме того, события `Loaded` и `Closed` самого типа `Window` были обработаны внутри открывающего элемента `<Window>` (опять-таки, для генерации обработчиков событий должна применяться IDE-среда, как объяснялось ранее). Если для обработки событий использовался визуальный конструктор, то в файле `MainWindow.Xaml.cs` должен присутствовать следующий код:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
  }
  private void btnViewXaml_Click(object sender, RoutedEventArgs e)
  {
  }
  private void Window_Closed(object sender, EventArgs e)
  {
  }
  private void Window_Loaded(object sender, RoutedEventArgs e)
  {
  }
}
```

Прежде чем продолжить, необходимо импортировать перечисленные ниже пространства имен в файл `MainWindow.xaml.cs`:

```
using System.IO;
using System.Windows.Markup;
```

Реализация события Loaded

Событие `Loaded` главного окна отвечает за выяснение, имеется ли файл `YourXaml.xaml` в папке, содержащей приложение. Если этот файл существует, то данные из него будут прочитаны и помещены в элемент `TextBox` главного окна. В противном случае элемент `TextBox` заполняется начальным стандартным описанием XAML пустого окна (которое представляет собой точно такую же разметку, как начальное определение окна, но только вместо `<Grid>` применяется `<StackPanel>`).

На заметку! Строку, которая предназначена для представления начальной разметки, отображаемой в разрабатываемом редакторе, набирать довольно утомительно из-за потребности в символах отмены, сопровождающих внутренние кавычки, поэтому будьте внимательны.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  // При загрузке главного окна приложения поместить
  // некоторый базовый текст XAML в текстовый блок.
  if (File.Exists("YourXaml.xaml"))
  {
    txtXamlData.Text = File.ReadAllText("YourXaml.xaml");
  }
  else
  {
    txtXamlData.Text =
      "<Window xmlns=\"http://schemas.microsoft.com/winfx/2006/xaml/
presentation\"\n"
     +"xmlns:x=\"http://schemas.microsoft.com/winfx/2006/xaml\"\n"
     +"Height =\"400\" Width =\"500\"
        WindowStartupLocation=\"CenterScreen\">\n"
     +"<StackPanel>\n"
     +"</StackPanel>\n"
     +"</Window>";
  }
}
```

Используя такой подход, приложение будет способно загружать разметку XAML, введенную в предыдущем сеансе, или при необходимости предоставлять стандартный блок разметки. В данный момент можно запустить программу и увидеть внутри объекта `TextBox` разметку, показанную на рис. 26.23.

Рис. 26.23. Первый запуск приложения `MyXamlPad.exe`

Реализация события `Click` объекта `Button`

В результате щелчка на объекте `Button` сначала текущие данные из `TextBox` сохраняются в файле `YourXaml.xaml`. Сохраненные данные будут читаться через метод `File.Open()`, чтобы получить объект `FileStream`. Причина в том, что для представления разметки XAML, подлежащей анализу, метод `XamlReader.Load()` требует тип, производный от `Stream` (а не простой тип `System.String`).

После загрузки описания XAML элемента `<Window>`, который должен конструироваться, мы создадим экземпляр `System.Windiows.Window` на основе находящейся в памяти разметки XAML и отобразим объект `Window` как модальное диалоговое окно:

```
private void btnViewXaml_Click(object sender, RoutedEventArgs e)
{
  // Записать данные из текстового блока в локальный файл *.xaml.
  File.WriteAllText("YourXaml.xaml", txtXamlData.Text);
  // Это окно, к которому будет динамически применяться разметка XAML.
  Window myWindow = null;
  // Открыть локальный файл *.xaml.
  try
  {
    using (Stream sr = File.Open("YourXaml.xaml", FileMode.Open))
    {
      // Подключить разметку XAML к объекту Window.
      myWindow = (Window)XamlReader.Load(sr);
      // Отобразить диалоговое окно и выполнить очистку.
      myWindow.ShowDialog();
      myWindow.Close();
      myWindow = null;
    }
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

Обратите внимание, что большая часть логики помещена внутрь блока `try/catch`. Таким образом, если файл `YourXaml.xaml` содержит некорректно сформированную разметку, то в результирующем окне сообщения отобразится сообщение об ошибке. Например, запустим программу и намеренно внесем ошибку в написание элемента `<StackPanel>`, скажем, добавив в открывающий элемент лишнюю букву `p`. После щелчка на кнопке отобразится сообщение об ошибке, подобное показанному на рис. 26.24.

Рис. 26.24. Перехват ошибок разметки

Реализация события `Closed`

Наконец, событие `Closed` класса `Window` позволяет удостовериться, что данные, введенные в `TextBox`, сохранены в файле `YourXaml.xaml`:

```
private void Window_Closed(object sender, EventArgs e)
{
  // Записать данные из текстового поля в локальный файл *.xaml.
  File.WriteAllText("YourXaml.xaml", txtXamlData.Text);
  Application.Current.Shutdown();
}
```

Тестирование приложения

Запустим приложение и ведем в текстовой области некоторую разметку XAML. Имейте в виду, что (подобно редактору Kaxaml) это приложение не позволяет указывать атрибуты XAML, связанные с генерацией кода (такие как `Class` или обработчики событий). В качестве первого теста введем следующий код XAML внутри области `<StackPanel>`:

```
<Button Height = "100" Width = "100" Content = "Click Me!">
  <Button.Background>
    <LinearGradientBrush StartPoint = "0,0" EndPoint = "1,1">
      <GradientStop Color = "Blue" Offset = "0" />
      <GradientStop Color = "Yellow" Offset = "0.25" />
      <GradientStop Color = "Green" Offset = "0.75" />
      <GradientStop Color = "Pink" Offset = "0.50" />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

После щелчка на кнопке появится окно, визуализирующее определения XAML (или, возможно, окно с сообщением об ошибке анализа — будьте внимательны при вводе).

На рис. 26.25 представлен возможный вывод.

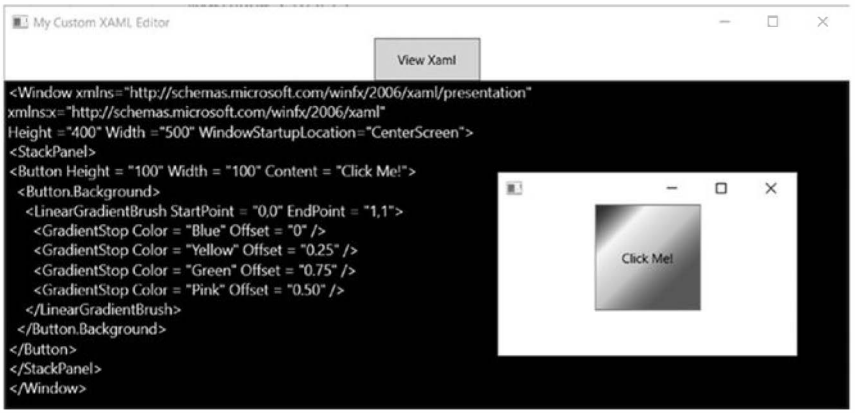


Рис. 26.25. Приложение `MyXamlPad.exe` в действии

Теперь введем показанную ниже разметку XAML сразу после текущего определения `<Button>`:

```
<Label Content = "Interesting...">
  <Label.Triggers>
    <EventTrigger RoutedEvent = "Label.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "FontSize">
            <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
              RepeatBehavior = "Forever"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Label.Triggers>
</Label>
```

Эта разметка является великолепной демонстрацией реальной мощи XAML. При ее тестировании вы обнаружите, что была создана простая анимационная последовательность. Службы анимации (а также графической визуализации) будут детально исследоваться в последующих главах; однако уже сейчас разметку XAML можно подстраивать и смотреть на конечный результат.

Изучение документации WPF

В заключение главы следует отметить, что тематике WPF в документации .NET 4.6 Framework SDK посвящен целый раздел. По мере исследования этого API-интерфейса и чтения остальных глав, раскрывающих инфраструктуру WPF, вы обнаружите настоятельную потребность в обращении к справочной системе. Там вы найдете множество примеров разметки XAML и подробные обучающие руководства по широкому спектру тем, начиная с программирования трехмерной графики и заканчивая сложными операциями привязки данных. Документация WPF доступна по адресу:

```
https://msdn.microsoft.com/ru-ru/library/ms754130(v=vs.110).aspx
```

По мере изучения этой части справочной системы вы столкнетесь с многочисленными примерами разметки XAML, которые можно копировать в буфер обмена и вставлять в созданный специальный редактор XAML. Тем не менее, перед тестированием понадобится заменить корневой элемент `<Page>` элементом `<Window>` (приложение было запрограммировано на отображение полноценных объектов `Window`, а не `Page`), если в примерах используется элемент `<Page>`. До перехода к чтению следующей главы посвятите некоторое время изучению интересующих вас тем и протестируйте дополнительную разметку в специальном инструменте XAML.

Исходный код. Проект `MyXamlPad` доступен в подкаталоге `Chapter_26`.

Резюме

Инфраструктура Windows Presentation Foundation (WPF) — это набор инструментов для построения пользовательских интерфейсов, появившийся в версии .NET 3.0. Основная цель WPF заключается в интеграции и унификации множества ранее разрозненных настольных технологий (двух- и трехмерная графика, разработка окон и элементов управления и т.п.) в единую унифицированную программную модель. Помимо этого в приложениях WPF обычно применяется расширяемый язык разметки приложений (Extensible Application Markup Language — XAML), который позволяет определять внешний вид и поведение элементов WPF через разметку.

Вспомните, что язык XAML позволяет описывать деревья объектов .NET с использованием декларативного синтаксиса. Во время исследования XAML в настоящей главе вы узнали о ряде новых фрагментов синтаксиса, в числе которых синтаксис "свойство-элемент" и присоединяемые свойства, а также о роли преобразователей типов и расширений разметки XAML.

Хотя XAML является ключевым аспектом любого приложения WPF производственного уровня, в первом примере главы было показано, как построить программу WPF с помощью только кода C#. Затем вы научились строить программу WPF с применением только XAML (что не рекомендуется, но полезно в качестве учебного примера). Наконец, вы узнали об использовании файлов отделенного кода, которые позволяют отделять внешний вид и поведение от функциональности.

В последнем примере главы было построено приложение WPF, которое дало возможность программно взаимодействовать с определениями XAML с применением классов `XamlReader` и `XamlWriter`. Попутно вы ознакомились с основными визуальными конструкторами WPF среды Visual Studio. В последующих главах будут приведены дополнительные сведения о визуальных конструкторах WPF.

ГЛАВА 27

Программирование с использованием элементов управления WPF

В главе 26 была представлена основа модели программирования WPF, включая исследование классов `Window` и `Application`, грамматику XAML и использование файлов кода. Кроме того, в ней было дано введение в процесс построения приложений WPF с применением визуальных конструкторов IDE-среды Visual Studio. В настоящей главе мы углубимся в конструирование более сложных графических пользовательских интерфейсов с использованием нескольких новых элементов управления и диспетчеров компоновки, а также по ходу дела выясним дополнительные возможности визуальных конструкторов WPF, доступных в Visual Studio.

Здесь будут рассматриваться некоторые важные темы, связанные с элементами управления WPF, такие как программная модель привязки данных и применение команд управления. Вы узнаете, как работать с интерфейсами Ink API и Documents API, которые позволяют получать ввод от пера (или мыши) и создавать форматированные документы с использованием протокола XML Paper Specification.

На заметку! В предыдущих изданиях этой книги для упрощения процесса построения графических пользовательских интерфейсов с применением инфраструктуры WPF использовался продукт под названием Microsoft Expression Blend. Однако последняя версия Visual Studio предлагает достаточную функциональность, чтобы можно было успешно строить пользовательские интерфейсы WPF для тематики, раскрываемой в книге. При желании освоить работу с Expression Blend обращайтесь к книге Эндрю Троелсена *Expression Blend 4 с примерами на C# для профессионалов* (ИД "Вильямс", 2011 г.)

Обзор основных элементов управления WPF

Если вы не являетесь новичком в области построения графических пользовательских интерфейсов, то общее назначение большинства элементов управления WPF не должно вызывать много вопросов. Независимо от того, какой набор инструментов для создания графических пользовательских интерфейсов вы применяли в прошлом (например, VB 6.0, MFC, Java AWT/Swing, Windows Forms, Mac OS X (Cocoa), GTK+/GTK# и т.п.), основные элементы управления WPF, перечисленные в табл. 27.1, скорее всего, покажутся знакомыми.

Таблица 27.1. Основные элементы управления WPF

Категория элементов управления WPF	Примеры членов	Описание
Основные элементы управления для пользовательского ввода	`Button`, `RadioButton`, `ComboBox`, `CheckBox`, `Calendar`, `DatePicker`, `Expander`, `DataGrid`, `ListBox`, `ListView`, `ToggleButton`, `TreeView`, `ContextMenu`, `ScrollBar`, `Slider`, `TabControl`, `TextBlock`, `TextBox`, `RepeatButton`, `RichTextBox`, `Label`	Инфраструктура WPF предлагает полное семейство элементов управления, которые можно задействовать при построении пользовательских интерфейсов
Боковые элементы окон и элементов управления	`Menu`, `ToolBar`, `StatusBar`, `ToolTip`, `ProgressBar`	Эти элементы пользовательского интерфейса служат для декорирования рамки объекта `Window` компонентами для ввода (наподобие `Menu`) и элементами информирования пользователя (скажем, `StatusBar` и `ToolTip`)
Элементы управления мультимедиа	`Image`, `MediaElement`, `SoundPlayerAction`	Эти элементы управления предоставляют поддержку воспроизведения аудио/видео и вывода изображений
Элементы управления компоновкой	`Border`, `Canvas`, `DockPanel`, `Grid`, `GridView`, `GridSplitter`, `GroupBox`, `Panel`, `TabControl`, `StackPanel`, `Viewbox`, `WrapPanel`	Инфраструктура WPF предлагает множество элементов управления, которые позволяют группировать и организовывать другие элементы для управления компоновкой

Элементы управления Ink API

В дополнение к общепринятым элементам управления WPF, упомянутым в табл. 27.1, инфраструктура WPF определяет элементы управления для работы с интерфейсом Ink API. Данный аспект разработки WPF полезен при построении приложений для Tablet PC, т.к. он позволяет захватывать ввод от пера. Тем не менее, это вовсе не означает, что стандартное настольное приложение не может задействовать Ink API, поскольку те же самые элементы управления могут работать с вводом от мыши.

Пространство имен `System.Windows.Ink` из сборки `PresentationCore.dll` содержит разнообразные поддерживающие типы Ink API (например, `Stroke` и `StrokeCollection`). Однако большинство элементов управления Ink API (наподобие `InkCanvas` и `InkPresenter`) упакованы вместе с общими элементами управления WPF в пространство имен `System.Windows.Controls` внутри сборки `PresentationFramework.dll`. Мы будем работать с интерфейсом Ink API позже в главе.

Элементы управления документов WPF

Вдобавок инфраструктура WPF предоставляет элементы управления для расширенной обработки документов, позволяя строить приложения, которые включают функциональность в стиле Adobe PDF. С применением типов из пространства имен

System.Windows.Documents (также находящегося в сборке PresentationFramework.dll) можно создавать готовые к печати документы, которые поддерживают масштабирование, поиск, пользовательские аннотации ("клейкие" заметки) и другие развитые средства работы с текстом.

Тем не менее, "за кулисами" элементы управления документов не используют API-интерфейсы Adobe PDF, а взамен работают с API-интерфейсом XML Paper Specification (XPS). Конечные пользователи никакой разницы не заметят, потому что документы PDF и XPS имеют практически идентичный вид и поведение. В действительности доступно множество бесплатных утилит, которые позволяют выполнять преобразования между этими двумя файловыми форматами на лету. В следующем примере мы будем иметь дело с некоторыми аспектами элементов управления документами.

Общие диалоговые окна WPF

Инфраструктура WPF также предлагает несколько общих диалоговых окон, таких как OpenFileDialog и SaveFileDialog, которые определены внутри пространства имен Microsoft.Win32 из сборки PresentationFramework.dll. Работа с любым из указанных диалоговых окон сводится к созданию объекта и вызову метода ShowDialog():

```
using Microsoft.Win32;

namespace WpfControls
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
    }
    private void btnShowDlg_Click(object sender, RoutedEventArgs e)
    {
      // Отобразить диалоговое окно сохранения файла.
      SaveFileDialog saveDlg = new SaveFileDialog();
      saveDlg.ShowDialog();
    }
  }
}
```

Как и можно было ожидать, в этих классах поддерживаются разнообразные члены, которые позволяют устанавливать фильтры файлов и пути к каталогам, а также получать доступ к выбранным пользователем файлам. Некоторые диалоговые окна применяются в последующих примерах; кроме того, будет показано, как строить специальные диалоговые окна для получения пользовательского ввода.

Подробные сведения находятся в документации

Несмотря на то что вам могло показаться, целью главы *не* является описание каждого члена абсолютно всех элементов управления WPF. Вместо этого будет дан обзор различных элементов управления с акцентом на лежащей в основе программной модели и ключевых службах, общих для большинства элементов управления WPF.

Полное представление о конкретной функциональности заданного элемента управления дает документация .NET Framework 4.6 SDK — в частности, раздел "Control Library" ("Библиотека элементов управления") справочной системы, доступный по ссылке https://msdn.microsoft.com/en-us/library/ms752324(v=vs.110).aspx.

На заметку! На момент написания главы документация WPF была довольно скудной, как вы обнаружите после перехода по упомянутой выше ссылке. Текущая документация относится к версии .NET 4.5. Хорошая новость в том, что в версии .NET 4.6 инфраструктура WPF претерпела совсем немного изменений, если сравнивать с .NET 4.5 (главным образом они касаются улучшения производительности).

Здесь вы найдете исчерпывающие описания каждого элемента управления, разнообразные примеры кода (на XAML и C#), а также информацию о цепочке наследования, реализованных интерфейсах и примененных атрибутах для любого элемента управления. Обязательно уделите время на исследование элементов управления, рассматриваемых в настоящей главе.

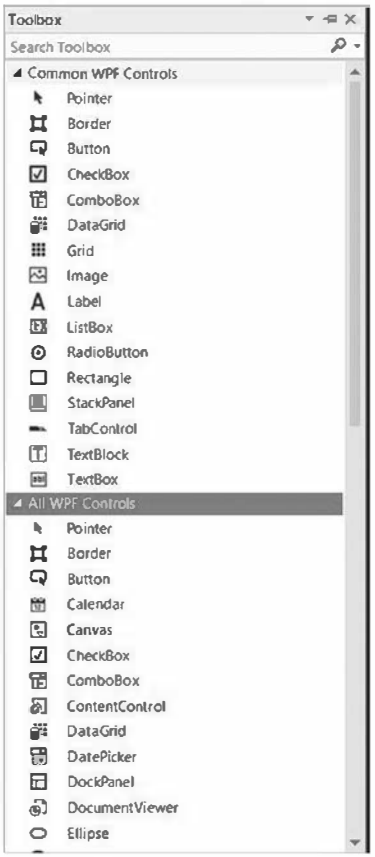


Рис. 27.1. Панель инструментов Visual Studio предоставляет доступ ко многим часто используемым элементам управления WPF

Краткий обзор визуального конструктора WPF в Visual Studio

Большинство стандартных элементов управления WPF упаковано в пространство имен `System.Windows.Controls` внутри сборки `PresentationFramework.dll`. При построении приложения WPF в Visual Studio множество этих элементов находится в панели инструментов при условии, что активным окном является визуальный конструктор WPF (рис. 27.1).

Подобно другим инфраструктурам для построения пользовательских интерфейсов в Visual Studio эти элементы управления можно перетаскивать на поверхность визуального конструктора WPF и конфигурировать их в окне Properties (Свойства), как было показано в главе 26. Хотя Visual Studio сгенерирует приличный объем разметки XAML автоматически, нет ничего необычного в том, чтобы затем редактировать разметку вручную. Давайте рассмотрим основы.

Работа с элементами управления WPF в Visual Studio

Вы можете вспомнить из главы 26, что после помещения элемента управления WPF на поверхность визуального конструктора Visual Studio в окне Properties необходимо установить свойство `x:Name`, т.к. это позволяет обращаться к объекту в связанном файле кода C#. Кроме того, на вкладке Events (События) окна Properties можно генерировать обработчики событий для выбранного элемента управления. Таким образом, с помощью Visual Studio можно было бы сгенерировать следующую разметку для простого элемента управления `Button`:

```
<Button x:Name="btnMyButton" Content="Click Me!" Height="23" Width="140"
        Click="btnMyButton_Click" />
```

Здесь свойство Content элемента Button устанавливается в простую строку "Click Me!". Однако благодаря модели содержимого элементов управления WPF можно создать элемент Button со следующим сложным содержимым:

```
<Button x:Name="btnMyButton" Height="121" Width="156"
Click="btnMyButton_Click">
  <Button.Content>
    <StackPanel Height="95" Width="128" Orientation="Vertical">
      <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
      <Label Width="59" FontSize="20" Content="Click!" Height="36" />
    </StackPanel>
  </Button.Content>
</Button>
```

Вы можете также вспомнить, что непосредственным дочерним элементом производного от ContentControl класса является предполагаемое содержимое, поэтому при указании сложного содержимого определять область <Button.Content> явно не требуется. Можно было бы написать такую разметку:

```
<Button x:Name="btnMyButton" Height="121" Width="156"
Click="btnMyButton_Click">
  <StackPanel Height="95" Width="128" Orientation="Vertical">
    <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
    <Label Width="59" FontSize="20" Content="Click!" Height="36" />
  </StackPanel>
</Button>
```

В любом случае свойство Content кнопки устанавливается в элемент <StackPanel> со связанными элементами. Создавать сложное содержимое такого рода можно также с применением визуального конструктора Visual Studio. После определения диспетчера компоновки для элемента управления содержимым его можно выбирать в визуальном конструкторе в качестве целевого компонента, на который будут перетаскиваться внутренние элементы управления. Каждый из них можно редактировать в окне Properties. Если окно Properties использовалось для обработки события Click элемента управления Button (как было показано в предшествующих объявлениях XAML), то IDE-среда сгенерирует пустой обработчик события, в который можно будет добавить специальный код, например:

```
private void btnMyButton_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show("You clicked the button!");
}
```

Работа с окном Document Outline

Вы должны также иметь в виду, что окно Document Outline (Схема документа) в Visual Studio (которое можно открыть через меню View⇨Other Windows (Вид⇨Другие окна)) будет удобным при проектировании элемента управления WPF со сложным содержимым. На рис. 27.2 показано логическое дерево XAML для создаваемого элемента Window. Щелчок на любом из этих узлов приводит к его автоматическому выбору в визуальном конструкторе для редактирования.

В текущей версии Visual Studio окно Document Outline имеет несколько дополнительных средств, которые вы можете счесть полезными. Справа от любого узла находится значок, напоминающий глазное яблоко. Щелчок на этом значке позволяет скрывать или отображать элемент в визуальном конструкторе, что может быть удобно, когда необходимо сосредоточиться на отдельном сегменте при редактировании (следует отметить,

что элемент *не* будет сокрыт во время выполнения, а только на поверхности визуального конструктора).

Справа от значка с глазным яблоком находится еще один значок, который позволяет "блокировать" элемент в визуальном конструкторе. Как и можно было догадаться, это очень удобно, когда нужно воспрепятствовать случайному изменению разметки XAML для заданного элемента. На самом деле блокировка элемента делает его допускающим только чтение на этапе проектирования (что вполне очевидно не мешает изменять состояние объекта во время выполнения).

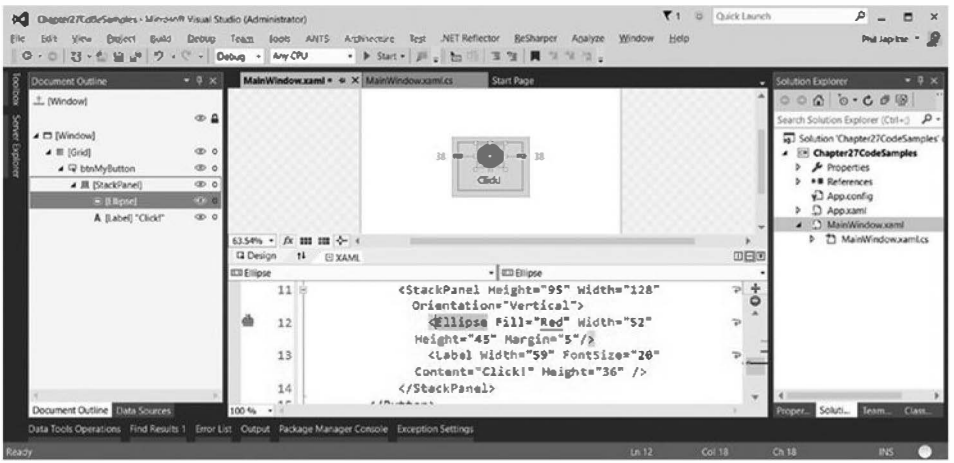


Рис. 27.2. Окно Document Outline в Visual Studio помогает осуществлять навигацию по сложному содержимому

Управление компоновкой содержимого с использованием панелей

Приложение WPF неизменно содержит определенное количество элементов пользовательского интерфейса (например, элементов ввода, графического содержимого, систем меню и строк состояния), которые должны быть хорошо организованы внутри разнообразных окон. После размещения элементов пользовательского интерфейса необходимо гарантировать их запланированное поведение, когда конечный пользователь изменяет размер окна или его части (как в случае окна с разделителем). Чтобы обеспечить сохранение элементами управления WPF своих позиций внутри окна, в котором они находятся, можно задействовать множество *типов панелей* (также известных как *диспетчеры компоновки*).

По умолчанию новый WPF-элемент `Window`, созданный с помощью Visual Studio, будет применять диспетчер компоновки типа `<Grid>` (вскоре он будет описан более подробно). Тем не менее, пока что рассмотрим элемент `Window` без каких-либо объявленных диспетчеров компоновки:

```
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ...
  Title="Fun with Panels!" Height="285" Width="325">
</Window>
```

Когда элемент управления объявляется прямо внутри окна, в котором панели не используются, он позиционируется по центру контейнера. Рассмотрим показанное далее простое объявление окна, содержащего единственный элемент управления `Button`. Независимо от того, как изменяются размеры окна, этот виджет пользовательского интерфейса всегда будет находиться на равном удалении от всех четырех границ клиентской области. Размер элемента `Button` определяется установленными значениями свойств `Height` и `Width` элемента `Button`.

```
<!-- Эта кнопка всегда находится в центре окна -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ...
  Title="Fun with Panels!" Height="285" Width="325">
  <Button x:Name="btnOK" Height = "100"
          Width="80" Content="OK"/>
</Window>
```

Также вспомните, что попытка помещения внутрь области `<Window>` сразу нескольких элементов вызовет ошибки разметки и компиляции. Причина в том, что свойству `Content` окна (или любого потомка `ContentControl`, если уж на то пошло) может быть присвоен только один объект. Следовательно, приведенная ниже разметка XAML приведет к ошибкам разметки и компиляции:

```
<!-- Ошибка! Свойство Content неявно устанавливается более одного раза! -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ...
  Title="Fun with Panels!" Height="285" Width="325">
  <!-- Ошибка! Два непосредственных дочерних элемента в <Window>! -->
  <Label x:Name="lblInstructions" Width="328" Height="27"
         FontSize="15" Content="Enter Information"/>
  <Button x:Name="btnOK" Height = "100" Width="80" Content="OK"/>
</Window>
```

Понятно, что от окна, допускающего наличие только одного элемента управления, мало толку. Когда окно должно содержать несколько элементов, эти элементы потребуется расположить внутри любого числа панелей. В панель будут помещены все элементы пользовательского интерфейса, которые представляют окно, после чего сама панель выступает в качестве единственного объекта, присваиваемого свойству `Content` окна.

Пространство имен `System.Windows.Controls` предлагает многочисленные панели, каждая из которых по-своему обслуживает внутренние элементы. С помощью панелей можно устанавливать поведение элементов управления при изменении размеров окна пользователем — будут они оставаться в те же местах, где были размещены на этапе проектирования, располагаться свободным потоком слева направо или сверху вниз и т.д.

Элементы управления типа панелей также разрешено помещать внутрь других панелей (например, элемент управления `DockPanel` может содержать `StackPanel` со своими элементами), чтобы обеспечить высокую гибкость и степень управления. В табл. 27.2 кратко описаны некоторые распространенные элементы управления типа панелей WPF.

В следующих нескольких разделах вы узнаете, как применять распространенные типы панелей, копируя заранее определенную разметку XAML в приложение `MyXamlPad.exe`, которое было создано в главе 26 (при желании разметку можно также загружать в редактор Kaxaml). Необходимые файлы XAML находятся в подкаталоге `PanelMarkup` внутри `Chapter_27` (рис. 27.3).

Таблица 27.2. Основные элементы управления типа панелей WPF

Элемент управления типа панели	Описание
`Canvas`	Предоставляет классический режим размещения содержимого. Элементы остаются в точности там, куда были помещены на этапе проектирования
`DockPanel`	Привязывает содержимое к указанной стороне панели (`Top` (верхняя), `Bottom` (нижняя), `Left` (левая) или `Right` (правая))
`Grid`	Располагает содержимое внутри серии ячеек, поддерживаемых внутри табличной сетки
`StackPanel`	Укладывает содержимое вертикально или горизонтально, как регламентируется свойством `Orientation`
`WrapPanel`	Позиционирует содержимое слева направо, перенося его на следующую строку по достижении границы панели. Дальнейшее упорядочение происходит последовательно сверху вниз или слева направо в зависимости от значения свойства `Orientation`

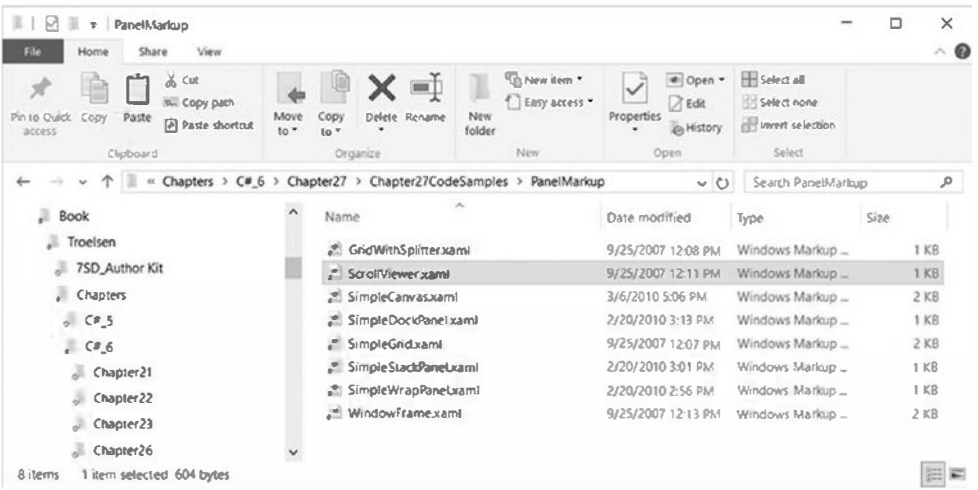


Рис. 27.3. Готовые файлы разметки XAML будут загружаться в приложение `MyXamlPad.exe` для тестирования разнообразных компоновок

Позиционирование содержимого внутри панелей `Canvas`

Вероятно, панель `Canvas` покажется наиболее привычной, т.к. она делает возможным абсолютное позиционирование содержимого пользовательского интерфейса. Если конечный пользователь изменяет размер окна, делая его меньше, чем компоновка, обслуживаемая панелью `Canvas`, то внутреннее содержимое будет невидимым до тех пор, пока контейнер не увеличится до размера, равного или превышающего размер области `Canvas`.

Чтобы добавить содержимое к `Canvas`, сначала нужно определить требуемые элементы управления внутри области между открывающим и закрывающим дескрипторами (`<Canvas>` и `</Canvas>`). Затем для каждого элемента управления необходимо указать левый верхний угол с использованием свойств `Canvas.Top` и `Canvas.Left`; именно здесь должна начинаться визуализация. Нижний правый угол каждого элемента управления можно задать неявно, устанавливая свойства `Canvas.Height` и `Canvas.Width`, либо явно с применением свойств `Canvas.Right` и `Canvas.Bottom`.

Для демонстрации Canvas в действии откроем готовый файл SimpleCanvas.xaml в текстовом редакторе и скопируем его содержимое в приложение MyXamlPad.exe (или в Kaxaml). Определение Canvas должно иметь следующий вид:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ...
  Title="Fun with Panels!" Height="285" Width="325">
  <Canvas Background="LightSteelBlue">
    <Button x:Name="btnOK" Canvas.Left="212" Canvas.Top="203"
            Width="80" Content="OK"/>
    <Label x:Name="lblInstructions" Canvas.Left="17" Canvas.Top="14"
           Width="328" Height="27" FontSize="15"
           Content="Enter Car Information"/>
    <Label x:Name="lblMake" Canvas.Left="17" Canvas.Top="60"
           Content="Make"/>
    <TextBox x:Name="txtMake" Canvas.Left="94" Canvas.Top="60"
             Width="193" Height="25"/>
    <Label x:Name="lblColor" Canvas.Left="17" Canvas.Top="109"
             Content="Color"/>
    <TextBox x:Name="txtColor" Canvas.Left="94" Canvas.Top="107"
             Width="193" Height="25"/>
    <Label x:Name="lblPetName" Canvas.Left="17" Canvas.Top="155"
          Content="Pet Name"/>
    <TextBox x:Name="txtPetName" Canvas.Left="94" Canvas.Top="153"
             Width="193" Height="25"/>
  </Canvas>
</Window>
```

Щелчок на кнопке View Xaml (Показать XAML) приведет к отображению окна, показанного на рис. 27.4.

Обратите внимание, что порядок объявления элементов содержимого внутри Canvas не влияет на расчет местоположения; на самом деле местоположение основано на размере элемента управления и значениях его свойств Canvas.Top, Canvas.Bottom, Canvas.Left и Canvas.Right.

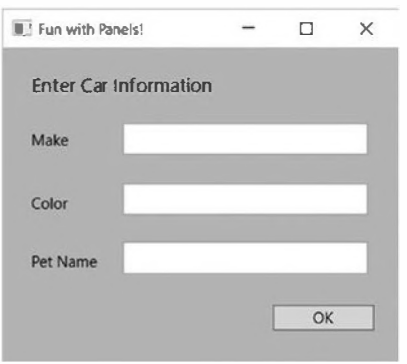


Рис. 27.4. Диспетчер компоновки Canvas обеспечивает абсолютное позиционирование содержимого

На заметку! Если подэлементы внутри Canvas не определяют специфическое местоположение с использованием синтаксиса присоединяемых свойств (например, Canvas.Left и Canvas.Top), то они автоматически прикрепляются к верхнему левому углу Canvas.

Применение типа Canvas может показаться предпочтительным способом организации содержимого (т.к. выглядит настолько знакомым), но данному подходу присущи некоторые ограничения. Во-первых, элементы внутри Canvas не изменяют свои размеры динамически при использовании стилей или шаблонов (скажем, их шрифты остаются незатронутыми). Во-вторых, панель Canvas не пытается сохранять элементы видимыми, когда конечный пользователь уменьшает размер окна.

Пожалуй, наилучшим применением типа Canvas является позиционирование *графического содержимого*. Например, при построении изображения с использованием XAML определенно понадобится сделать так, чтобы все линии, фигуры и текст оставались на

своих местах, а не динамически перемещались в случае изменения пользователем размера окна. Мы еще вернемся к Canvas в главе 28 при обсуждении служб визуализации графики WPF.

Позиционирование содержимого внутри панелей WrapPanel

Панель WrapPanel позволяет определять содержимое, которое будет протекать сквозь панель, когда размер окна изменяется. При позиционировании элементов внутри WrapPanel их координаты верхнего левого и нижнего правого углов не указываются, как это обычно делается в Canvas. Однако для каждого подэлемента допускается определение значений свойств Height и Width (наряду с другими свойствами), чтобы управлять их общим размером в контейнере.

Поскольку содержимое внутри WrapPanel не пристыковывается к заданной стороне панели, порядок объявления элементов играет важную роль (содержимое визуализируется от первого элемента к последнему). В файле SimpleWrapPanel.xaml находится следующая разметка (заключенная внутрь определения <Window>):

```
<WrapPanel Background="LightSteelBlue">
  <Label x:Name="lblInstruction" Width="328"
         Height="27" FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox x:Name="txtMake" Width="193" Height="25"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox x:Name="txtColor" Width="193" Height="25"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName" Width="193" Height="25"/>
  <Button x:Name="btnOK" Width="80" Content="OK"/>
</WrapPanel>
```

Когда эта разметка загружена, при изменении ширины окна содержимое выглядит не особо привлекательно, т.к. оно перетекает слева направо внутри окна (рис. 27.5).

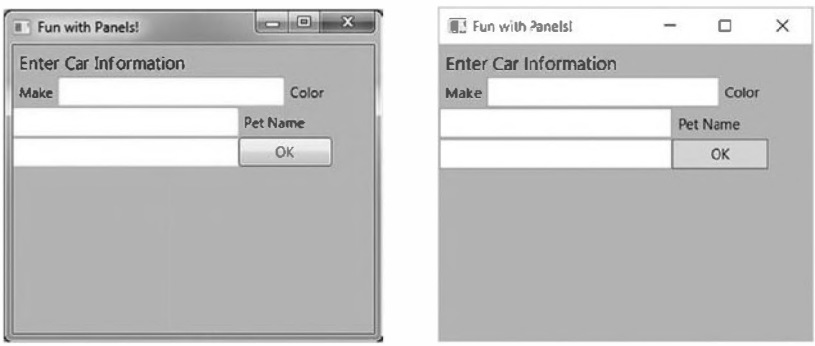


Рис. 27.5. Содержимое в панели WrapPanel ведет себя во многом подобно традиционной странице HTML

По умолчанию содержимое WrapPanel перетекает слева направо. Тем не менее, если изменить значение свойства Orientation на Vertical, то можно заставить содержимое перетекать сверху вниз:

```
<WrapPanel Background="LightSteelBlue" Orientation="Vertical">
```

Панель WrapPanel (как и ряд других типов панелей) может быть объявлена с указанием значений ItemWidth и ItemHeight, которые управляют стандартным размером каждого элемента. Если подэлемент предоставляет собственные значения Height и/или Width, то он будет позиционироваться относительно размера, установленного для него панелью.

Взгляните на следующую разметку:

```
<WrapPanel Background="LightSteelBlue" Orientation="Horizontal"
ItemWidth="200" ItemHeight="30">
  <Label x:Name="lblInstruction"
         FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox x:Name="txtMake"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox x:Name="txtColor"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName"/>
  <Button x:Name="btnOK" Width="80" Content="OK"/>
</WrapPanel>
```

В результате визуализации получается окно, показанное на рис. 27.6 (обратите внимание на размер и позицию элемента управления `Button`, для которого было задано уникальное значение `Width`).

Рис. 27.6. Панель `WrapPanel` может устанавливать ширину и высоту отдельного элемента

После просмотра рис. 27.7 вы наверняка согласитесь с тем, что панель `WrapPanel` — обычно не лучший выбор для организации содержимого непосредственно в окне, поскольку ее элементы могут беспорядочно смешиваться, когда пользователь изменяет размер окна. В большинстве случаев `WrapPanel` будет подэлементом панели другого типа, позволяя небольшой области окна переносить свое содержимое при изменении размера (как, например, элемент управления `ToolBar`).

Позиционирование содержимого внутри панелей `StackPanel`

Подобно `WrapPanel` элемент управления `StackPanel` организует содержимое внутри одиночной строки, которая может быть ориентирована горизонтально или вертикально (по умолчанию) в зависимости от значения, присвоенного свойству `Orientation`. Однако отличие между ними заключается в том, что `StackPanel` не пытается переносить содержимое при изменении размера окна пользователем. Вместо этого элементы в `StackPanel` просто растягиваются (согласно выбранной ориентации), приспосабливаясь к размеру самой панели `StackPanel`. Например, в файле `SimpleStackPanel.xaml` содержится разметка, которая в результате дает вывод, показанный на рис. 27.7:

```
<StackPanel Background="LightSteelBlue">
  <Label x:Name="lblInstruction"
         FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox Name="txtMake"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox x:Name="txtColor"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName"/>
  <Button x:Name="btnOK" Width="80" Content="OK"/>
</StackPanel>
```

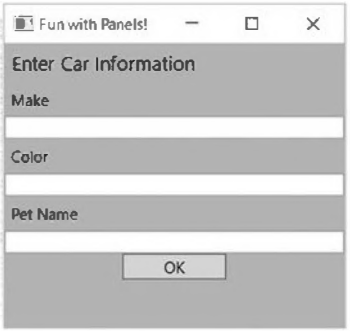


Рис. 27.7. Вертикальное укладывание содержимого

Рис. 27.8. Горизонтальное укладывание содержимого

Если присвоить свойству `Orientation` значение `Horizontal`, то визуализированный вывод станет таким, как на рис. 27.8:

```
<StackPanel Background="LightSteelBlue" Orientation="Horizontal">
```

Опять-таки, подобно `WrapPanel` панель `StackPanel` редко применяется для организации содержимого прямо внутри окна. Панель `StackPanel` должна использоваться как вложенная панель в какой-то главной панели.

Позиционирование содержимого внутри панелей `Grid`

Из всех панелей, предоставляемых API-интерфейсами WPF, панель `Grid`, несомненно, является самой гибкой. Аналогично таблице HTML панель `Grid` может состоять из набора ячеек, каждая из которых имеет свое содержимое. При определении `Grid` выполняются перечисленные ниже шаги.

1. Определение и конфигурирование каждой колонки.
2. Определение и конфигурирование каждой строки.
3. Назначение содержимого каждой ячейке сетки с применением синтаксиса присоединяемых свойств.

На заметку! Если не определить какие-либо строки и колонки, то по умолчанию элемент `<Grid>` будет состоять из единственной ячейки, которая заполняет всю поверхность окна. Кроме того, если не указать ячейку для подэлемента `<Grid>`, то он автоматически разместится в колонке 0 и строке 0.

Первые два шага (определение колонок и строк) выполняются с использованием элементов `<Grid.ColumnDefinitions>` и `<Grid.RowDefinitions>`, которые содержат коллекции элементов `<ColumnDefinition>` и `<RowDefinition>` соответственно. Каждая ячейка внутри сетки на самом деле является настоящим объектом .NET, так что можно желаемым образом настраивать внешний вид и поведение каждого элемента.

Ниже представлено простое определение `<Grid>` (из файла `SimpleGrid.xaml`), которое организует содержимое пользовательского интерфейса, как показано на рис. 27.9:

```
<Grid ShowGridLines ="True" Background ="LightSteelBlue">
  <!-- Определить строки и колонки -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
```

```
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>

  <!-- Добавить элементы в ячейки сетки -->
  <Label x:Name="lblInstruction" Grid.Column ="0" Grid.Row ="0"
        FontSize="15" Content="Enter Car Information"/>
  <Button x:Name="btnOK" Height ="30" Grid.Column ="0"
         Grid.Row ="0" Content="OK"/>
  <Label x:Name="lblMake" Grid.Column ="1"
        Grid.Row ="0" Content="Make"/>
  <TextBox x:Name="txtMake" Grid.Column ="1"
          Grid.Row ="0" Width="193" Height="25"/>
  <Label x:Name="lblColor" Grid.Column ="0"
        Grid.Row ="1" Content="Color"/>
  <TextBox x:Name="txtColor" Width="193" Height="25"
          Grid.Column ="0" Grid.Row ="1" />

  <!--Добавить цвет к ячейке с именем, просто чтобы сделать картину интереснее-->
  <Rectangle Fill ="LightGreen" Grid.Column ="1" Grid.Row ="1" />
  <Label x:Name="lblPetName" Grid.Column ="1" Grid.Row ="1" Content="Pet Name"/>
  <TextBox x:Name="txtPetName" Grid.Column ="1" Grid.Row ="1"
           Width="193" Height="25"/>
</Grid>
```

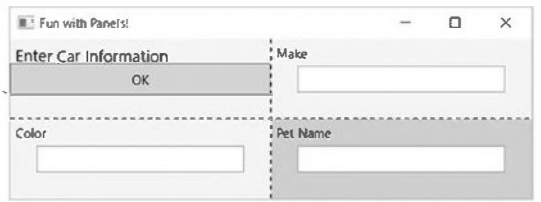


Рис. 27.9. Панель `Grid` в действии

Обратите внимание, что каждый элемент (включая элемент `Rectangle` светло-зеленого цвета) прикрепляется к ячейке сетки с применением присоединяемых свойств `Grid.Row` и `Grid.Column`. По умолчанию порядок ячеек начинается с левой верхней, которая указывается с использованием `Grid.Column="0"` и `Grid.Row="0"`. Учитывая, что сетка состоит всего из четырех ячеек, нижняя правая ячейка может быть идентифицирована как `Grid.Column="1"` и `Grid.Row="1"`.

Панели `Grid` с типами `GridSplitter`

Панели `Grid` также могут поддерживать *разделители*. Как вам возможно известно, разделители позволяют конечному пользователю изменять размеры колонок и строк сетки. При этом содержимое каждой ячейки с изменяемым размером реорганизует себя на основе находящихся в нем элементов. Добавить разделители к `Grid` довольно просто: для этого необходимо определить элемент управления `<GridSplitter>` и с применением синтаксиса присоединяемых свойств указать строку или колонку, на которую он воздействует.

Имейте в виду, что для обеспечения видимости на экране разделителя потребуется присвоить значение свойству `Width` или `Height` (в зависимости от вертикального или горизонтального разделения). Ниже показана простая панель `Grid` с разделителем на первой колонке (`Grid.Column="0"`) из файла `GridWithSplitter.xaml`:

```
<Grid Background ="LightSteelBlue">
  <!-- Определить колонки -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width ="Auto"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <!-- Добавить метку в ячейку 0 -->
  <Label x:Name="lblLeft" Background ="GreenYellow"
       Grid.Column="0" Content ="Left!"/>
  <!-- Определить разделитель -->
  <GridSplitter Grid.Column ="0" Width ="5"/>
  <!-- Добавить метку в ячейку 1 -->
  <Label x:Name="lblRight" Grid.Column ="1" Content ="Right!"/>
</Grid>
```

Прежде всего, обратите внимание, что колонка, которая будет поддерживать разделитель, имеет свойство `Width`, установленное в `Auto`. Кроме того, элемент `<GridSplitter>` использует синтаксис присоединяемых свойств для установки колонки, с которой он работает. В выводе (рис. 27.10) можно заметить 5-пиксельный разделитель, который позволяет изменять размер каждого элемента `Label` (на него указывает стрелка). Из-за того, что для элементов `Label` не было задано свойство `Height` или `Width`, они заполняют всю ячейку.

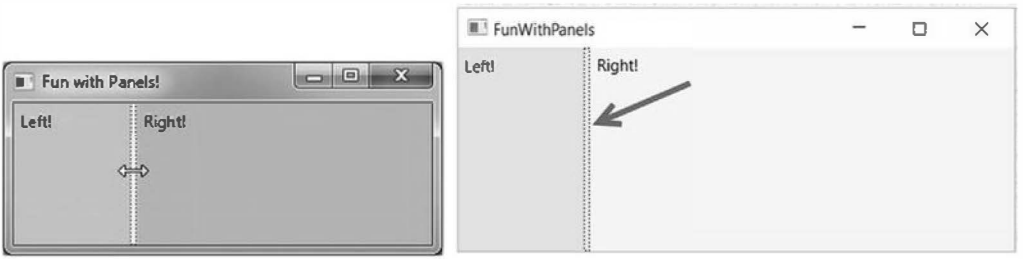


Рис. 27.10. Панель `Grid` с разделителем

Позиционирование содержимого внутри панелей `DockPanel`

Панель `DockPanel` обычно применяется в качестве контейнера, который содержит любое количество дополнительных панелей для группирования связанного содержимого. Панели `DockPanel` используют синтаксис присоединяемых свойств (как было показано в типах `Canvas` и `Grid`) для управления тем, куда будет пристыковываться каждый элемент внутри `DockPanel`.

В файле `SimpleDockPanel.xaml` определена следующая простая панель `DockPanel`, которая дает результат, показанный на рис. 27.11:

```
<DockPanel LastChildFill ="True">
  <!-- Стыковать элементы к панели -->
  <Label x:Name="lblInstruction" DockPanel.Dock ="Top"
         FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" DockPanel.Dock ="Left" Content="Make"/>
  <Label x:Name="lblColor" DockPanel.Dock ="Right" Content="Color"/>
  <Label x:Name="lblPetName" DockPanel.Dock ="Bottom" Content="Pet Name"/>
  <Button x:Name="btnOK" Content="OK"/>
</DockPanel>
```

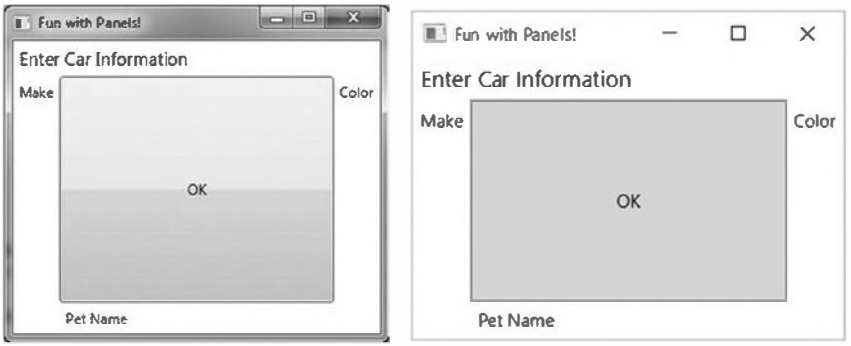


Рис. 27.11. Простая панель `DockPanel`

На заметку! Если добавить множество элементов к одной стороне `DockPanel`, то они выстроятся вдоль указанной грани в порядке их объявления.

Преимущество применения типов `DockPanel` заключается в том, что при изменении пользователем размера окна каждый элемент остается прикрепленным к указанной (посредством `DockPanel.Dock`) стороне панели. Также обратите внимание, что внутри открывающего дескриптора `<DockPanel>` в этом примере атрибут `LastChildFill` установлен в `true`. Поскольку элемент `Button` на самом деле является "последним дочерним" элементом в контейнере, он будет растянут, чтобы занять все оставшееся пространство.

Включение прокрутки в типах панелей

Полезно упомянуть, что в рамках инфраструктуры WPF поставляется класс `ScrollViewer`, который обеспечивает автоматическое поведение прокрутки данных внутри объектов панелей. Вот как он определяется в файле `ScrollViewer.xaml`:

```
<ScrollViewer>
  <StackPanel>
    <Button Content ="First" Background = "Green" Height ="40"/>
    <Button Content ="Second" Background = "Red" Height ="40"/>
    <Button Content ="Third" Background = "Pink" Height ="40"/>
    <Button Content ="Fourth" Background = "Yellow" Height ="40"/>
    <Button Content ="Fifth" Background = "Blue" Height ="40"/>
  </StackPanel>
</ScrollViewer>
```

Результат визуализации приведенного определения XAML представлен на рис. 27.12 (обратите внимание на отображение справа в окне линейки прокрутки, т.к. размера окна не хватает, чтобы показать все пять кнопок).

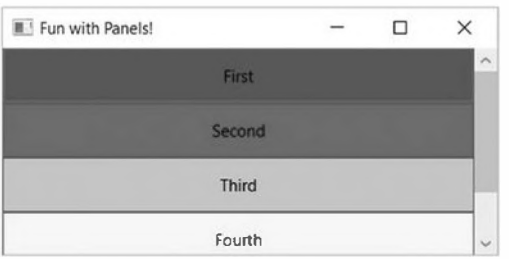


Рис. 27.12. Работа с классом `ScrollViewer`

Как и можно было ожидать, каждая панель предоставляет многочисленные члены, позволяющие точно настраивать размещение содержимого. В качестве связанного замечания: многие элементы управления WPF поддерживают два удобных свойства (`Padding` и `Margin`), которые предоставляют самому элементу возможность информирования панели о том, как с ним

следует обращаться. В частности, свойство `Padding` управляет тем, сколько свободного пространства должно окружать внутренний элемент управления, а свойство `Margin` контролирует объем дополнительного пространства вне элемента управления.

На этом краткий экскурс в основные типы панелей WPF и различные способы позиционирования их содержимого завершен. Далее мы покажем, как использовать визуальные конструкторы Visual Studio для создания компоновок.

Конфигурирование панелей с использованием визуальных конструкторов Visual Studio

Теперь, когда вы ознакомились с разметкой XAML, применяемой при определении ряда общих диспетчеров компоновки, полезно знать, что IDE-среда Visual Studio предлагает очень хорошую поддержку для конструирования компоновок. Ключевым компонентом является окно Document Outline, описанное ранее в главе. Чтобы проиллюстрировать некоторые основы, мы создадим новый проект WPF Application (Приложение WPF) по имени `VisualLayoutTesterApp`.

В первоначальной разметке для `Window` по умолчанию используется диспетчер компоновки `Grid`:

```
<Window x:Class="VisualLayoutTesterApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:VisualLayoutTesterApp"
    mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

На тот случай, если диспетчер компоновки `Grid` устраивает, отметим, что в визуальном конструкторе можно легко разделять и менять размеры ячеек сетки, как показано на рис. 27.13. Для этого понадобится выбрать компонент `Grid` в окне Document Outline после чего щелкнуть на границе сетки, чтобы создать новые строки и колонки.

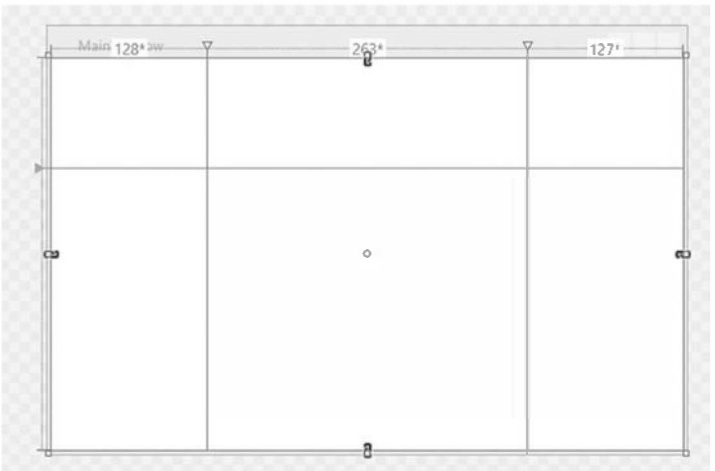


Рис. 27.13. Элемент управления `Grid` может быть разделен на ячейки с применением визуального конструктора IDE-среды

Теперь предположим, что определена сетка с каким-то числом ячеек. Затем можно перетаскивать элементы управления в интересующую ячейку сетки и IDE-среда автоматически установит свойства `Grid.Row` и `Grid.Column` элемента управления. Вот как может выглядеть разметка, сгенерированная IDE-средой после перетаскивания элемента `Button` в предопределенную ячейку:

```
<Button x:Name="button" Content="Button" Grid.Column="1"
HorizontalAlignment="Left" Margin="21,21.4,0,0" Grid.Row="1"
VerticalAlignment="Top" Width="75"/>
```

Пусть, например, было решено вообще не использовать элемент `Grid`. Щелчок правой кнопкой мыши на любом узле разметки в окне Document Outline приводит к открытию контекстного меню, которое содержит пункт, позволяющий заменить текущий контейнер другим (рис. 27.14). Делать это следует с особой осторожностью, т.к. в результате (скорее всего) радикально изменятся позиции элементов управления, чтобы удовлетворять правилам нового типа панели.

Еще один удобный прием связан с возможностью выбора в визуальном конструкторе набора элементов управления и последующего их группирования внутри нового вложенного диспетчера компоновки. Предположим, что имеется панель `Canvas`, в которой определен набор произвольных объектов (при желании можно преобразовать первоначальную панель `Grid` в `Canvas` с применением приема, продемонстрированного на рис. 27.14). Теперь выделим множество элементов на поверхности визуального конструктора, щелкая на каждом элементе левой кнопкой мыши при нажатой клавише <Ctrl>. Если затем щелкнуть правой кнопкой мыши на выделенном наборе элементов, то с помощью открывшегося контекстного меню их можно сгруппировать в новую вложенную панель (рис. 27.15).

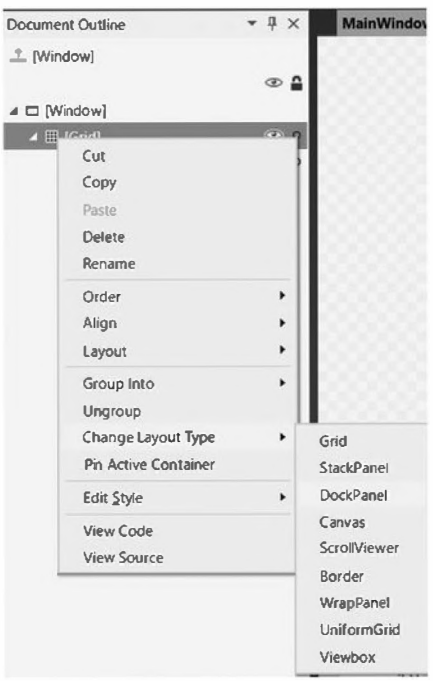


Рис. 27.14. Окно Document Outline позволяет выполнять преобразование в другие типы панелей

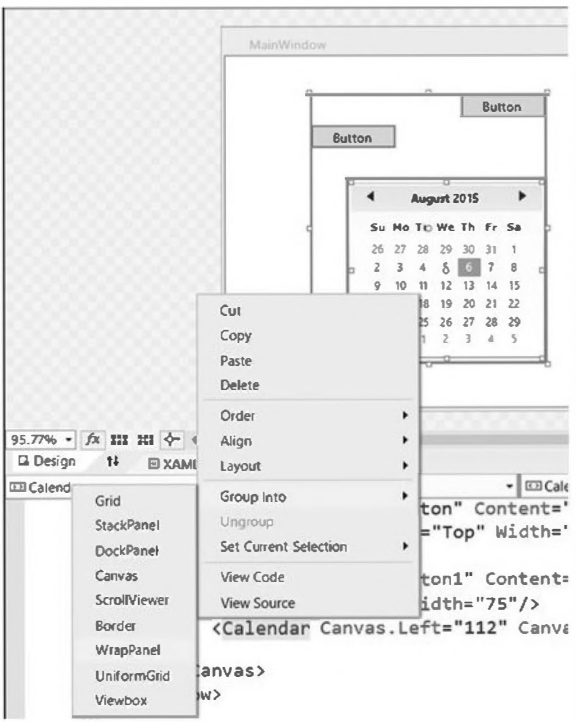


Рис. 27.15. Группирование элементов в новую вложенную панель

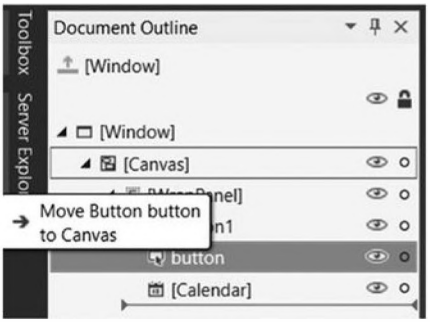


Рис. 27.16. Перемещение элементов с помощью окна Document Outline

После этого снова нужно заглянуть в окно Document Outline, чтобы проконтролировать вложенную систему компоновки. Так как строятся полнофункциональные окна WPF, скорее всего, всегда нужно будет использовать систему вложенных компоновок, а не просто выбирать единственную панель для отображения всего пользовательского интерфейса (на самом деле в оставшихся примерах приложений WPF обычно так и будет делаться). В качестве финального замечания следует указать, что все узлы в окне Document Outline поддерживают перетаскивание. Например, если требуется переместить элемент управления, который в текущий момент находится внутри `Canvas`, в родительскую панель, то это можно сделать так, как иллюстрируется на рис. 27.16.

В последующих главах, посвященных WPF, будут представлены дополнительные ускоренные приемы для работы с компоновкой там, где они возможны. Тем не менее, определенно полезно посвятить какое-то время самостоятельному экспериментированию и проверке разнообразных средств. В следующем примере этой главы будет демонстрироваться построение вложенного диспетчера компоновки для специального приложения обработки текста (с проверкой правописания).

Построение окна с использованием вложенных панелей

Как упоминалось ранее, в типичном окне WPF для получения желаемой системы компоновки применяется не единственный элемент управления типа панели, а одни панели вкладываются внутрь других. Начнем с создания нового проекта WPF Application по имени `MyWordPad`.

Нашей целью является конструирование компоновки, в которой главное окно имеет расположенную в верхней части систему меню, под ней — панель инструментов и в нижней части окна — строку состояния. Строка состояния будет содержать область для хранения текстовых подсказок, которые отображаются при выборе пользователем пункта меню (или кнопки в панели инструментов). Система меню и панель инструментов предоставят триггеры пользовательского интерфейса для закрытия приложения и отображения вариантов правописания в виджете `Expander`. На рис. 27.17 показана начальная компоновка; здесь также отображаются предполагаемые варианты правописания для "WPF".

Рис. 27.17. Использование вложенных панелей для формирования пользовательского интерфейса окна

Обратите внимание, что две кнопки панели инструментов содержат не ожидаемые изображения, а простые текстовые значения. Этого явно не достаточно для приложения производственного уровня, но назначение изображений кнопкам панели инстру-

ментов обычно предусматривает применение встроенных ресурсов, которые рассматриваются в главе 28 (так что пока обойдемся текстом). Кроме того, при наведении на кнопку Check (Проверить) курсор мыши изменяется, и в единственной панели строки состояния отображается полезное сообщение пользовательского интерфейса.

Чтобы приступить к построению описанного пользовательского интерфейса, модифицируем начальное определение XAML типа `Window` для использования дочернего элемента `<DockPanel>` вместо стандартного `<Grid>`:

```
<Window x:Class="MyWordPad.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:MyWordPad"
    mc:Ignorable="d"
    Title="My Spell Checker" Height="350" Width="525">
  <!-- Эта панель устанавливает содержимое окна -->
  <DockPanel>
  </DockPanel>
</Window>
```

Построение системы меню

Системы меню в WPF представлены классом `Menu`, который поддерживает коллекцию объектов `MenuItem`. При построении системы меню в XAML можно заставить каждый объект `MenuItem` обрабатывать разнообразные события, наиболее примечательным из которых является `Click`, возникающее при выборе подэлемента конечным пользователем. В рассматриваемом примере будут созданы два пункта меню верхнего уровня (File (Файл) и Tools (Сервис); меню Edit (Правка) строится позже), которые будут содержать в себе подэлементы Exit (Выход) и Spelling Hints (Подсказки по правописанию) соответственно.

В дополнение к обработке события `Click` для каждого подэлемента необходимо также обработать события `MouseEnter` и `MouseExit`, которые применяются для установки текста в строке состояния. Добавим в контекст элемента `<DockPanel>` следующую разметку (для обработки каждого события можно использовать окно Properties (Свойства) среды Visual Studio, как было описано в главе 26):

```
<!-- Стыковать систему меню к верхней части -->
<Menu DockPanel.Dock ="Top"
      HorizontalAlignment="Left" Background="White" BorderBrush ="Black">
  <MenuItem Header="_File">
    <Separator/>
    <MenuItem Header ="_Exit" MouseEnter ="MouseEnterExitArea"
              MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
    </MenuItem>
    <MenuItem Header="_Tools">
      <MenuItem Header ="_Spelling Hints"
          MouseEnter ="MouseEnterToolsHintsArea"
          MouseLeave ="MouseLeaveArea" Click ="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```

Обратите внимание, что система меню стыкована к верхней части `DockPanel`. Кроме того, элемент `<Separator>` применяется для добавления в систему меню тонкой горизонтальной линии прямо перед пунктом `Exit`. Значения `Header` для каждого `MenuItem`

содержат символ подчеркивания (например, _Exit). Подобным образом указывается символ, который будет подчеркиваться, когда конечный пользователь нажмет клавишу <Alt> (для ввода клавиатурного сокращения). Символ подчеркивания используется вместо символа & в Windows Forms, т.к. язык XAML основан на XML, а символ & имеет особый смысл в XML.

После построения системы меню необходимо реализовать различные обработчики событий. Прежде всего, есть обработчик пункта меню File⇨Exit (Файл⇨Выход), FileExit_Click(), который просто закрывает окно, что, в свою очередь, приводит к завершению приложения, поскольку это окно самого высшего уровня. Обработчики событий MouseEnter и MouseExit для каждого подэлемента будут в итоге обновлять строку состояния; однако пока мы просто оставим их пустыми. Наконец, обработчик ToolsSpellingHints_Click() для пункта меню Tools⇨Spelling Hints также оставим пока пустым. Ниже показаны текущие обновления файла отделенного кода:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
  }
  protected void FileExit_Click(object sender, RoutedEventArgs args)
  {
    // Закрыть это окно.
    this.Close();
  }
  protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
  {
  }
  protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
  {
  }
  protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
  {
  }
  protected void MouseLeaveArea(object sender, RoutedEventArgs args)
  {
  }
}
```

Визуальное построение меню

Наряду с тем, что всегда полезно знать, как вручную определять элементы в XAML, это может быть слегка утомительным. В Visual Studio поддерживается возможность визуального конструирования систем меню, панелей инструментов, строк состояния и многих других элементов управления пользовательского интерфейса. В качестве краткого примера предположим, что внутри нового Window имеется новый элемент управления Menu (можно вставить тестовое окно через пункт меню Project⇨Add Window (Проект⇨Добавить окно) и следовать примеру). Если теперь щелкнуть правой кнопкой мыши на элементе управления Menu, то в контекстном меню появится пункт Add MenuItem (Добавить пункт меню), как показано на рис. 27.18.

После добавления набора пунктов верхнего уровня можно переходить к добавлению пунктов подменю, разделителей, развертыванию и свертыванию самого меню и выполнению других связанных с меню операций, для чего снова щелкать правой кнопкой мыши и выбирать соответствующие пункты в контекстном меню. На рис. 27.19 демонстрируется один из возможных способов визуального проектирования простой системы меню (обязательно следует просмотреть сгенерированную разметку XAML).

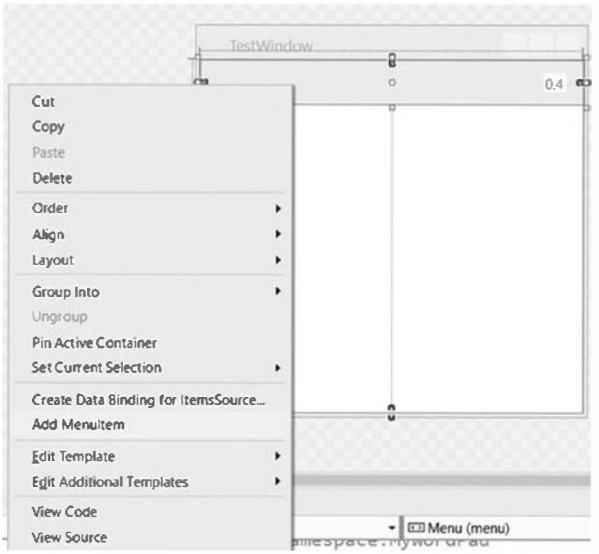


Рис. 27.18. Визуальное добавление элементов в объект `Menu`

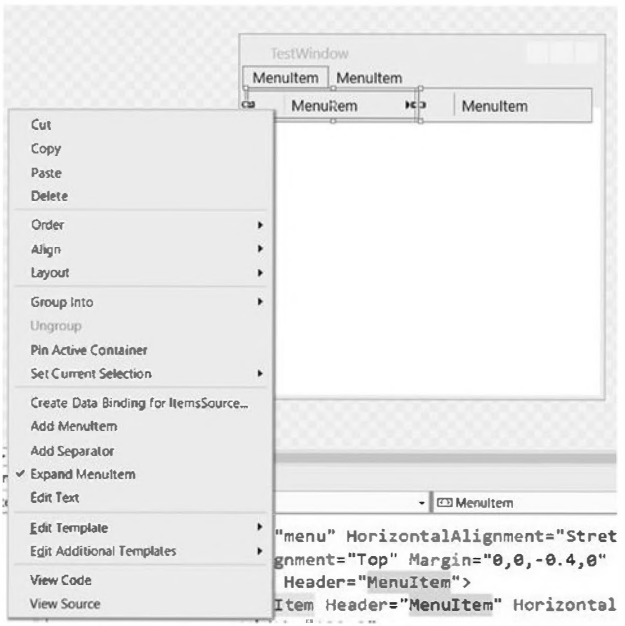


Рис. 27.19. Визуальное добавление элементов в объект `MenuItem`

По мере исследования оставшейся части примера `MyWordPad` обычно будет показана финальная сгенерированная разметка XAML; тем не менее, выделите некоторое время на экспериментирование с визуальными конструкторами.

Построение панели инструментов

Панели инструментов (представляемые в WPF классом `ToolBar`) обычно предлагают альтернативный способ активизации пунктов меню.

Поместите следующую разметку непосредственно после закрывающего дескриптора определения <Menu>:

```
<!-- Разместить панель инструментов ниже области меню -->
<ToolBar DockPanel.Dock ="Top" >
  <Button Content ="Exit" MouseEnter ="MouseEnterExitArea"
          MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
  <Separator/>
  <Button Content ="Check" MouseEnter ="MouseEnterToolsHintsArea"
          MouseLeave ="MouseLeaveArea" Click ="ToolsSpellingHints_Click"
          Cursor="Help" />
</ToolBar>
```

Наш элемент управления ToolBar состоит из двух элементов управления Button, которые предназначены для обработки тех же самых событий теми же методами из файла кода. С помощью такого приема можно дублировать обработчики для обслуживания и пунктов меню, и кнопок панели управления. Хотя в этой панели применяются типичные нажимаемые кнопки, вы должны принимать во внимание, что тип ToolBar "является" ContentControl, а потому на его поверхность можно помещать любые типы (скажем, раскрывающиеся списки, изображения и графику). Еще один интересный аспект здесь связан с тем, что кнопка Check поддерживает специальный курсор мыши посредством свойства Cursor.

На заметку! Элемент ToolBar может быть дополнительно помещен внутрь элемента <ToolBarTray>, который управляет компоновкой, стыковкой и перетаскиванием для набора объектов ToolBar. За подробной информацией обращайтесь в документацию .NET Framework 4.6 SDK.

Построение строки состояния

Элемент управления StatusBar прикрепляется к нижней части <DockPanel> и содержит единственный элемент управления <TextBlock>, который ранее в главе не использовался. Элемент TextBlock можно применять для хранения текста с форматированием вроде выделения полужирным и подчеркиванием, добавления разрывов строк и т.д. Поместим приведенную ниже разметку сразу после предыдущего определения элемента управления ToolBar:

```
<!-- Разместить строку состояния внизу -->
<StatusBar DockPanel.Dock ="Bottom" Background="Beige" >
  <StatusBarItem>
    <TextBlock Name="statBarText" Text="Ready"/>
  </StatusBarItem>
</StatusBar>
```

Завершение проектирования пользовательского интерфейса

Финальный аспект проектирования нашего пользовательского интерфейса связан с определением поддерживающего разделители элемента Grid, в котором определены две колонки. Слева находится элемент управления Expander, который будет отображать список предполагаемых вариантов правописания, помещенный внутрь <StackPanel>, а справа — элемент TextBox с поддержкой многострочного текста, линеек прокрутки и включенной проверкой орфографии. Элемент <Grid> может быть целиком размещен в левой части родительской панели <DockPanel>. Чтобы завершить определение пользовательского интерфейса окна, добавим следующую разметку XAML непосредственно под разметкой, описывающей StatusBar:

```
<Grid DockPanel.Dock ="Left" Background ="AliceBlue">
  <!-- Определить строки и колонки -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <GridSplitter Grid.Column ="0" Width ="5" Background ="Gray" />
  <StackPanel Grid.Column="0" VerticalAlignment ="Stretch" >
    <Label Name="lblSpellingInstructions" FontSize="14" Margin="10,10,0,0">
     Spelling Hints
    </Label>
    <Expander Name="expanderSpelling" Header ="Try these!"
              Margin="10,10,10,10">
      <!-- Будет заполняться программно -->
      <Label Name ="lblSpellingHints" FontSize ="12"/>
    </Expander>
  </StackPanel>
  <!-- Это будет областью для ввода -->
  <TextBox  Grid.Column ="1"
            SpellCheck.IsEnabled ="True"
            AcceptsReturn ="True"
            Name ="txtData" FontSize ="14"
            BorderBrush ="Blue"
            VerticalScrollBarVisibility="Auto"
            HorizontalScrollBarVisibility="Auto">
  </TextBox>
</Grid>
```

Реализация обработчиков событий MouseEnter/MouseLeave

К этому моменту пользовательский интерфейс окна готов. Понадобится лишь предоставить реализации оставшихся обработчиков событий. Начнем с обновления файла кода C# так, чтобы каждый из обработчиков событий MouseEnter и MouseLeave устанавливал в текстовой панели строки состояния подходящее сообщение, которое окажет помощь конечному пользователю:

```
public partial class MainWindow : System.Windows.Window
{
  ...
  protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
  {
    statBarText.Text = "Exit the Application";
  }
  protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
  {
    statBarText.Text = "Show Spelling Suggestions";
  }
  protected void MouseLeaveArea(object sender, RoutedEventArgs args)
  {
    statBarText.Text = "Ready";
  }
}
```

Теперь приложение можно запустить. Текст в строке состояния должен изменяться в зависимости от того, над каким пунктом меню или кнопкой панели инструментов находится курсор.

Реализация логики проверки правописания

Инфраструктура WPF имеет встроенную поддержку проверки правописания, независимую от продуктов Microsoft Office. Это значит, что использовать уровень взаимодействия с COM для обращения к функции проверки правописания Microsoft Word не понадобится; та же самая функциональность добавляется с помощью всего нескольких строк кода.

Вспомните, что при определении элемента управления <TextBox> свойство SpellCheck.IsEnabled устанавливается в true. В результате неправильно написанные слова подчеркиваются красной волнистой линией, как это происходит в Microsoft Office. Более того, лежащая в основе программная модель предоставляет доступ к механизму проверки правописания, который позволяет получить список предполагаемых вариантов для слов, написанных с ошибкой. Добавим в метод ToolsSpellingHints_Click() следующий код:

```
protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
{
  string spellingHints = string.Empty;
  // Попробовать получить ошибку правописания в текущем положении курсора ввода
  SpellingError error = txtData.GetSpellingError(txtData.CaretIndex);
  if (error != null)
  {
    // Построить строку с предполагаемыми вариантами правописания.
    foreach (string s in error.Suggestions)
    {
      spellingHints += $"{s}\n";
    }
    // Отобразить предполагаемые варианты и раскрыть элемент Expander.
    lblSpellingHints.Content = spellingHints;
    expanderSpelling.IsExpanded = true;
  }
}
```

Приведенный выше код довольно прост. С применением свойства CaretIndex извлекается объект SpellingError и вычисляется текущее положение курсора ввода в текстовом поле. Если в указанном месте присутствует ошибка (т.е. значение error не равно null), то осуществляется проход в цикле по списку предполагаемых вариантов с использованием свойства Suggestions. После того, как все предполагаемые варианты для неправильно написанного слова получены, они помещаются в элемент Label внутри элемента Expander.

Вот и все! С помощью всего нескольких строк процедурного кода (и приличной порции разметки XAML) заложены основы для функционирования текстового процессора. После изучения *управляющих команд* мы добавим дополнительные возможности.

Понятие команд WPF

Инфраструктура Windows Presentation Foundation предоставляет поддержку того, что может считаться *независимыми от элементов управления событиями*, посредством *архитектуры команд*. Обычное событие .NET определяется внутри некоторого базового класса и может использоваться только этим классом или его потомками. Следовательно, нормальные события .NET тесно привязаны к классу, в котором они определены.

В отличие от этого команды WPF представляют собой похожие на события сущности, которые не зависят от специфического элемента управления и во многих случаях могут успешно применяться к многочисленным (и на вид несвязанным) типам элементов управления. Вот лишь несколько примеров: WPF поддерживает команды копирования, вырезания и вставки, которые могут использоваться в разнообразных элементах пользовательского интерфейса (пункты меню, кнопки панели инструментов, специальные кнопки), а также клавиатурные комбинации (<Ctrl+C>, <Ctrl+V> и т.д.).

В то время как другие инструментальные наборы для построения пользовательских интерфейсов (вроде Windows Forms) предлагают для таких целей стандартные события, в результате получается избыточный и трудный в сопровождении код. Внутри модели WPF в качестве альтернативы можно применять команды. Итогом обычно оказывается более компактная и гибкая кодовая база.

Внутренние объекты команд

Инфраструктура WPF поставляется с множеством встроенных команд, каждую из которых можно ассоциировать с соответствующей клавиатурной комбинацией (или другим входным жестом). С точки зрения программирования команда WPF — это любой объект, поддерживающий свойство (часто называемое `Command`), которое возвращает объект, реализующий показанный ниже интерфейс `ICommand`:

```
public interface ICommand
{
  // Возникает, когда происходят изменения, влияющие
  // на то, должна ли выполняться команда.
  event EventHandler CanExecuteChanged;
  // Определяет метод, который выясняет, может ли
  // команда выполняться в ее текущем состоянии.
  bool CanExecute(object parameter);
  // Определяет метод для вызова при обращении к команде.
  void Execute(object parameter);
}
```

В WPF определены разнообразные классы команд, которые открывают доступ к примерно сотне готовых объектов команд. В таких классах определены многочисленные свойства, представляющие специфические объекты команд, каждый из которых реализует интерфейс `ICommand`. В табл. 27.3 кратко описаны избранные стандартные объекты команд (более подробную информацию ищите в документации .NET Framework 4.6 SDK).

Таблица 27.3. Внутренние объекты команд WPF

Класс WPF	Объекты команд	Описание
`ApplicationCommands`	`Close`, `Copy`, `Cut`, `Delete`, `Find`, `Open`, `Paste`, `Save`, `SaveAs`, `Redo`, `Undo`	Разнообразные команды уровня приложения
`ComponentCommands`	`MoveDown`, `MoveFocusBack`, `MoveLeft`, `MoveRight`, `ScrollToEnd`, `ScrollToHome`	Разнообразные команды, которые являются общими для компонентов пользовательского интерфейса
`MediaCommands`	`BoostBase`, `ChannelUp`, `ChannelDown`, `FastForward`, `NextTrack`, `Play`, `Rewind`, `Select`, `Stop`	Разнообразные команды, связанные с мультимедиа
`NavigationCommands`	`BrowseBack`, `BrowseForward`, `Favorites`, `LastPage`, `NextPage`, `Zoom`	Разнообразные команды, связанные с навигационной моделью WPF
`EditingCommands`	`AlignCenter`, `CorrectSpellingError`, `DecreaseFontSize`, `EnterLineBreak`, `EnterParagraphBreak`, `MoveDownByLine`, `MoveRightByWord`	Разнообразные команды, связанные с интерфейсом Documents API в WPF

Подключение команд к свойству Command

Для подключения любого свойства команд WPF к элементу пользовательского интерфейса, который поддерживает свойство `Command` (такому как `Button` или `MenuItem`), потребуется сделать совсем немного. В качестве примера модифицируем текущую систему меню, добавив новый пункт верхнего уровня по имени Edit (Правка) с тремя подэлементами, которые позволяют копировать, вставлять и вырезать текстовые данные:

```
<Menu DockPanel.Dock ="Top"
      HorizontalAlignment="Left"
      Background="White" BorderBrush ="Black">
  <MenuItem Header="_File" Click ="FileExit_Click" >
    <MenuItem Header ="_Exit" MouseEnter ="MouseEnterExitArea"
              MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
  </MenuItem>
  <!-- Новые пункты меню с командами -->
  <MenuItem Header="_Edit">
    <MenuItem Command ="ApplicationCommands.Copy"/>
    <MenuItem Command ="ApplicationCommands.Cut"/>
    <MenuItem Command ="ApplicationCommands.Paste"/>
  </MenuItem>
  <MenuItem Header="_Tools">
    <MenuItem Header ="_Spelling Hints"
              MouseEnter ="MouseEnterToolsHintsArea"
              MouseLeave ="MouseLeaveArea"
              Click ="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```

Обратите внимание, что свойству `Command` каждого подэлемента в меню Edit присвоено некоторое значение. В результате пункты меню автоматически получают корректные имена и горячие клавиши (например, <Ctrl+C> для операции вырезания) в пользовательском интерфейсе меню, и приложение теперь способно *копировать*, *вырезать* и *вставлять* текст без необходимости в написании процедурного кода.

Если запустить приложение и выделить какую-то часть текста, то сразу же можно использовать новые пункты меню. Вдобавок приложение также оснащено возможностью реагирования на стандартную операцию щелчка правой кнопкой мыши, предлагая пользователю те же самые пункты в контекстном меню (рис. 27.20).

Подключение команд к произвольным действиям

Если объект команды нужно подключить к произвольному событию (специфичному для приложения), то придется прибегнуть к написанию процедурного кода. Задача несложная, но требует чуть больше логики, чем можно видеть в XAML. Например, пусть необходимо, чтобы все окно реагировало на нажатие клавиши <F1>, активизируя ассоциированную с ним справочную систему. Также предположим, что в файле кода для главного окна определен новый метод по имени `SetF1CommandBinding()`, который вызывается внутри конструктора после вызова `InitializeComponent()`:

```
public MainWindow()
{
  InitializeComponent();
  SetF1CommandBinding();
}
```

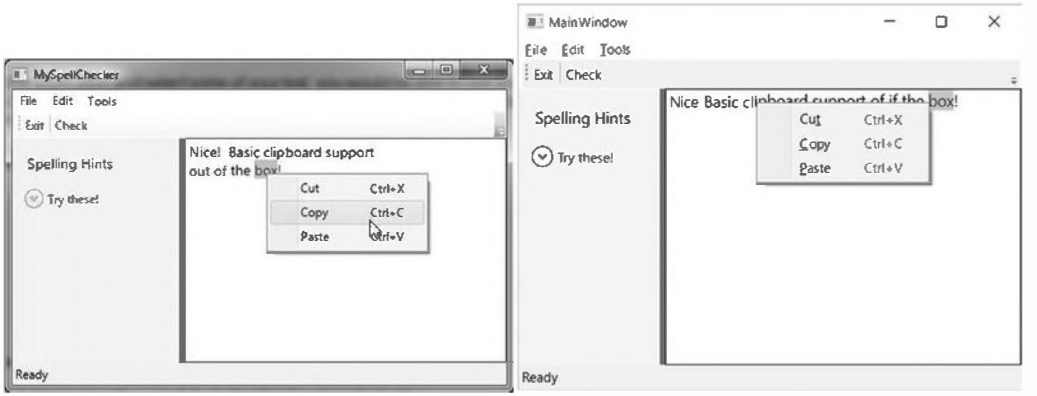


Рис. 27.20. Объекты команд предлагают полезный набор встроенной функциональности

Метод `SetF1CommandBinding()` будет программно создавать новый объект `CommandBinding`, который можно применять всякий раз, когда нужно привязать объект команды к заданному обработчику событий в приложении. Сконфигурируем объект `CommandBinding` для работы с командой `ApplicationCommands.Help`, которая автоматически выдается по нажатию клавиши <F1>:

```
private void SetF1CommandBinding()
{
  CommandBinding helpBinding = new CommandBinding(ApplicationCommands.Help);
  helpBinding.CanExecute += CanHelpExecute;
  helpBinding.Executed += HelpExecuted;
  CommandBindings.Add(helpBinding);
}
```

Большинство объектов `CommandBinding` будет обрабатывать событие `CanExecute` (которое позволяет указать, инициируется ли команда для конкретной операции программы) и событие `Executed` (где можно определить код, который должен быть выполнен после того, как команда произошла). Добавим следующие обработчики событий к нашему производному от `Window` типу (форматы методов регламентируются ассоциированными делегатами):

```
private void CanHelpExecute(object sender, CanExecuteRoutedEventArgs e)
{
  // Если нужно предотвратить выполнение команды,
  // то можно установить CanExecute в false.
  e.CanExecute = true;
}
private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
  MessageBox.Show("Look, it is not that difficult. Just type something!",
                  "Help!");
}
```

В предыдущем фрагменте кода метод `CanHelpExecute()` реализован так, что справка по нажатию <F1> всегда разрешена; это делается путем возвращения `true`. Однако если в определенных ситуациях справочная система отображаться не должна, то необходимо предпринять соответствующую проверку и возвращать `false`. Наша "справочная система", отображаемая внутри `HelpExecute()`, представляет собой всего лишь обычное окно сообщения. Теперь можно запустить приложение. После нажатия <F1> появляется наша упрощенная справочная система (рис. 27.21).

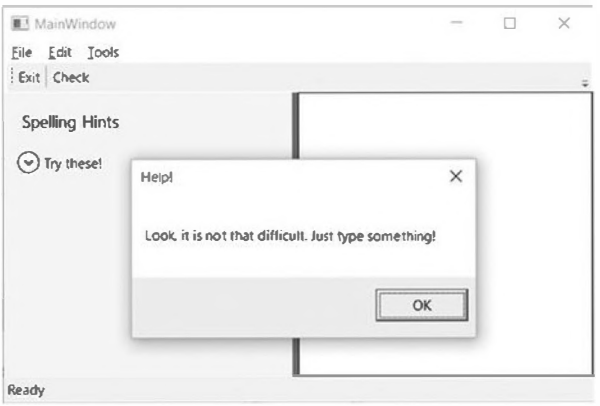


Рис. 27.21. Специальная справочная система

Работа с командами Open и Save

Чтобы завершить текущий пример, мы добавим функциональность сохранения текстовых данных во внешнем файле и открытия файлов `*.txt` для редактирования. Можно пойти длинным путем, вручную добавив программную логику, которая включает и отключает пункты меню в зависимости от того, имеются ли данные внутри `TextBox`. Тем не менее, для сокращения усилий можно прибегнуть к услугам команд.

Начнем с обновления элемента `<MenuItem>`, который представляет меню File верхнего уровня, путем добавления двух новых подменю, использующих объекты `Save` и `Open` класса `ApplicationCommands`:

```
<MenuItem Header="_File">
  <MenuItem Command ="ApplicationCommands.Open"/>
  <MenuItem Command ="ApplicationCommands.Save"/>
  <Separator/>
  <MenuItem Header ="_Exit" MouseEnter ="MouseEnterExitArea"
            MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
</MenuItem>
```

Вспомните, что все объекты команд реализуют интерфейс `ICommand`, в котором определены два события (`CanExecute` и `Executed`). Теперь необходимо разрешить окну выполнять указанные команды, предварительно проверив возможность делать это в текущих обстоятельствах; таким образом, определим обработчик события для запуска специального кода.

Понадобится наполнить коллекцию `CommandBindings`, поддерживаемую окном. В разметке XAML потребуется применить синтаксис "свойство-элемент" для определения области `<Window.CommandBindings>`, в которую помещаются два определения `<CommandBinding>`. Модифицируем определение `<Window>`, как показано ниже:

```
<Window x:Class="MyWordPad.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MySpellChecker" Height="331" Width="508"
  WindowStartupLocation ="CenterScreen" >
  <!-- Это информирует Window о том, какие обработчики вызывать
       при поступлении команд Open и Save -->
  <Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
          Executed="OpenCmdExecuted" CanExecute="OpenCmdCanExecute"/>
```

```
    <CommandBinding Command="ApplicationCommands.Save"
                    Executed="SaveCmdExecuted" CanExecute="SaveCmdCanExecute"/>
  </Window.CommandBindings>

  <!-- Эта панель устанавливает содержимое окна -->
  <DockPanel>
  ...
  </DockPanel>
</Window>
```

Щелкнем правой кнопкой мыши на каждом из атрибутов `Executed` и `CanExecute` в редакторе XAML и выберем в контекстном меню пункт Navigate to Event Handler (Перейти к обработчику события). Как объяснялось в главе 26, это автоматически сгенерирует заготовку кода для обработчика события. Теперь в файле кода C# для окна должны присутствовать четыре пустых обработчика событий.

Реализация обработчиков события `CanExecute` будет сообщать окну, что можно инициировать соответствующие события `Executed` в любой момент, для чего свойство `CanExecute` входного объекта `CanExecuteRoutedEventArgs` устанавливается в `true`:

```
private void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = true;
}
private void SaveCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = true;
}
```

Обработчики события `Executed` выполняют действительную работу по отображению диалоговых окон открытия и сохранения файла; они также отправляют данные из `TextBox` в файл. Начнем с импортирования пространств имен `System.IO` и `Microsoft.Win32` в файл кода. Завершенный код прост:

```
private void OpenCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
  // Создать диалоговое окно открытия файла и показать в нем
  // только текстовые файлы.
  var openDlg = new OpenFileDialog { Filter = "Text Files |*.txt"};

  // Был ли совершен щелчок на кнопке OK?
  if (true == openDlg.ShowDialog())
  {
    // Загрузить содержимое выбранного файла.
    string dataFromFile = File.ReadAllText(openDlg.FileName);

    // Отобразить строку в TextBox.
    txtData.Text = dataFromFile;
  }
}
private void SaveCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
  var saveDlg = new SaveFileDialog { Filter = "Text Files |*.txt"};
  // Был ли совершен щелчок на кнопке OK?
  if (true == saveDlg.ShowDialog())
  {
    // Сохранить данные из TextBox в указанном файле.
    File.WriteAllText(saveDlg.FileName, txtData.Text);
  }
}
```

На заметку! Система команд WPF более подробно рассматривается в главе 30. Там будут создаваться специальные команды на основе `ICommand` и `RelayCommands`.

Итак, пример и начальное знакомство с элементами управления WPF завершены. Вы узнали, как работать с базовыми командами, системами меню, строками состояния, панелями инструментов, вложенными панелями и несколькими элементами пользовательского интерфейса (вроде `TextBox` и `Expander`). Следующий пример будет иметь дело с более экзотическими элементами управления, а также с рядом важных служб WPF.

Исходный код. Проект `MyWordPad` доступен в подкаталоге `Chapter_27`.

Понятие маршрутизируемых событий

Вы могли заметить, что в предыдущем примере кода передавался параметр `RoutedEventArgs` вместо `EventArgs`. Модель маршрутизируемых событий является усовершенствованием стандартной модели событий CLR и спроектирована для обеспечения возможности обработки событий в манере, подходящей описанию XAML дерева объектов. Предположим, что имеется новый проект WPF Application по имени `WPFRoutedEvents`. Модифицируем описание XAML начального окна, добавив следующий элемент управления `<Button>`, который определяет сложное содержимое:

```
<Button Name="btnClickMe" Height="75" Width = "250"
        Click ="btnClickMe_Clicked">
  <StackPanel Orientation ="Horizontal">
    <Label Height="50" FontSize ="20">Fancy Button!</Label>
    <Canvas Height ="50" Width ="100" >
    <Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
             Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
    <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
             Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
  </StackPanel>
</Button>
```

Обратите внимание, что в открывающем определении элемента `<Button>` было обработано событие `Click` за счет указания имени метода, который должен вызываться при возникновении события. Событие `Click` работает с делегатом `RoutedEventHandler`, который ожидает обработчика события, принимающего `object` в первом параметре и `System.Winodws.RoutedEventArgs` во втором. Реализуем этот обработчик:

```
public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
{
  // Делать что-нибудь, когда на кнопке произведен щелчок.
  MessageBox.Show("Clicked the button");
}
```

После запуска приложения окно сообщения будет отображаться независимо от того, на какой части содержимого кнопки был выполнен щелчок (зеленый элемент `Ellipse`, желтый элемент `Ellipse`, элемент `Label` или поверхность элемента `Button`). В принципе это хорошо. Только представьте, насколько громоздким оказалась бы обработка событий WPF, если бы пришлось обрабатывать событие `Click` для каждого из упомянутых подэлементов. Дело не только в том, что создание отдельных обработчиков событий для каждого аспекта `Button` является трудоемкой задачей, а еще и в том, что в результате получился бы сложный в сопровождении код.

К счастью, *маршрутизируемые события* WPF позаботятся об автоматическом вызове единственного обработчика события `Click` вне зависимости от того, на какой части кнопки был совершен щелчок. Выражаясь просто, модель маршрутизируемых событий автоматически распространяет событие вверх (или вниз) по дереву объектов в поисках подходящего обработчика.

Точнее говоря, маршрутизируемое событие может использовать три *стратегии маршрутизации*. Если событие перемещается от точки возникновения вверх к другим областям определений внутри дерева объектов, то его называют *пузырьковым событием*. И наоборот, если событие перемещается от самого внешнего элемента (например, `Window`) вниз к точке возникновения, то его называют *туннельным событием*. Наконец, если событие инициируется и обрабатывается только элементом, внутри которого оно возникло (что можно было бы описать как нормальное событие CLR), то его называют *прямым событием*.

Роль пузырьковых маршрутизируемых событий

В текущем примере, когда пользователь щелкает на внутреннем овале желтого цвета, событие `Click` поднимается на следующий уровень области определения (`Canvas`), затем на `StackPanel` и, в конце концов, на уровень `Button`, где и обрабатывается. Подобным же образом, если пользователь щелкает на `Label`, то событие всплывает на уровень `StackPanel` и в итоге попадает в элемент `Button`.

Благодаря такому шаблону пузырьковых маршрутизируемых событий не придется беспокоиться о регистрации специфичных обработчиков события `Click` для всех членов составного элемента управления. Однако если необходимо выполнить специальную логику обработки щелчков для нескольких элементов внутри того же самого дерева объектов, то это вполне можно делать.

В целях иллюстрации предположим, что щелчок на элементе управления `outerEllipse` должен быть обработан в уникальной манере. Сначала обработаем событие `MouseDown` для этого подэлемента (графически визуализируемые типы вроде `Ellipse` не поддерживают событие `Click`, но могут отслеживать действия кнопки мыши через события `MouseDown`, `MouseUp` и т.д.):

```
<Button Name="btnClickMe" Height="75" Width = "250"
        Click ="btnClickMe_Clicked">
  <StackPanel Orientation ="Horizontal">
    <Label Height="50" FontSize ="20">Fancy Button!</Label>
    <Canvas Height ="50" Width ="100" >
    <Ellipse Name = "outerEllipse" Fill ="Green"
             Height ="25" MouseDown ="outerEllipse_MouseDown"
             Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
    <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
             Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
  </StackPanel>
</Button>
```

Затем реализуем подходящий обработчик событий, который для демонстрационных целей будет просто изменять свойство `Title` главного окна:

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
  // Изменить заголовок окна.
  this.Title = "You clicked the outer ellipse!";
}
```

После этого можно выполнять разные действия в зависимости от того, на чем конкретно щелкнул конечный пользователь (на внешнем эллипсе или в любом другом месте внутри области кнопки).

На заметку! Пузырьковые маршрутизируемые события всегда перемещаются из точки возникновения до *следующей определяющей области*. Таким образом, в рассмотренном примере щелчок на элементе `innerEllipse` привел бы к попаданию события в контейнер `Canvas`, а *не* в элемент `outerEllipse`, потому что оба элемента являются типами `Ellipse` внутри области определения `Canvas`.

Продолжение или прекращение пузырькового распространения

При текущем состоянии приложения, когда пользователь щелкает на объекте `outerEllipse`, запускается зарегистрированный обработчик события `MouseDown` для этого объекта `Ellipse`, после чего событие всплывет до события `Click` кнопки. Чтобы информировать WPF о необходимости останова пузырькового распространения по дереву объектов, понадобится установить свойство `Handled` параметра `MouseButtonEventArgs` в `true`:

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
  // Изменить заголовок окна.
  this.Title = "You clicked the outer ellipse!";

  // Остановить пузырьковое распространение.
  e.Handled = true;
}
```

Заголовок окна изменится, но окно `MessageBox`, отображаемое обработчиком события `Click` элемента `Button`, не появится. По существу пузырьковые маршрутизируемые события позволяют сложной группе содержимого действовать либо как единый логический элемент (например, `Button`), либо как отдельные элементы (скажем, `Ellipse` внутри `Button`).

Роль туннельных маршрутизируемых событий

Строго говоря, маршрутизируемые события по своей природе могут быть *пузырьковыми* (как только что было описано) или *туннельными*. Туннельные события (имена которых начинаются с префикса `Preview` наподобие `PreviewMouseDown`) спускаются от самого верхнего элемента во внутренние области определения дерева объектов. В общем и целом для каждого пузырькового события в библиотеках базовых классов WPF предусмотрено связанное туннельное событие, которое возникает *перед* его пузырьковым аналогом. Например, перед возникновением пузырькового события `MouseDown` сначала инициируется туннельное событие `PreviewMouseDown`.

Обработка туннельных событий выглядит очень похожей на обработку любых других событий: нужно просто указать имя обработчика события в разметке XAML (или при необходимости применить соответствующий синтаксис обработки событий C# в файле кода) и реализовать этот обработчик в файле кода. Для демонстрации взаимодействия туннельных и пузырьковых событий начнем с организации обработки события `PreviewMouseDown` для объекта `outerEllipse`:

```
<Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
         MouseDown ="outerEllipse_MouseDown"
         PreviewMouseDown ="outerEllipse_PreviewMouseDown"
         Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
```

Затем модифицируем текущее определение класса C#, обновив каждый обработчик события (для всех объектов) за счет добавления данных о событии в переменную-член _mouseActivity типа string с использованием входного объекта аргументов события. В результате появится возможность наблюдать за потоком событий, появляющихся в фоновом режиме.

```
public partial class MainWindow : Window
{
  string _mouseActivity = string.Empty;
  public MainWindow()
  {
    InitializeComponent();
  }
  public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
  {
    AddEventInfo(sender, e);
    MessageBox.Show(_mouseActivity, "Your Event Info");
    // Очистить строку для следующего цикла.
    _mouseActivity = "";
  }
  private void AddEventInfo(object sender, RoutedEventArgs e)
  {
    _mouseActivity += string.Format(
      "{0} sent a {1} event named {2}.\n", sender,
      e.RoutedEvent.RoutingStrategy,
      e.RoutedEvent.Name);
  }
  private void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
  {
    AddEventInfo(sender, e);
  }
  private void outerEllipse_PreviewMouseDown(object sender, MouseButtonEventArgs e)
  {
    AddEventInfo(sender, e);
  }
}
```

Обратите внимание, что ни в одном обработчике событий пузырьковое распространение не останавливается. После запуска приложения отобразится окно с уникальным сообщением, которое зависит от места на кнопке, где был произведен щелчок.

На рис. 27.22 показан результат щелчка на внешнем объекте Ellipse.

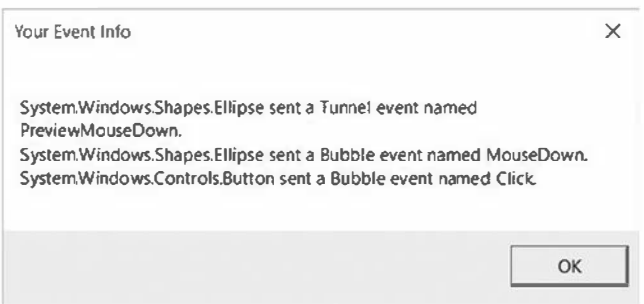


Рис. 27.22. Сначала возникает туннельное событие, а затем пузырьковое

Итак, почему события WPF обычно встречаются парами (одно туннельное и одно пузырьковое)? Ответ можно сформулировать так: благодаря предварительному просмотру событий появляется возможность выполнения любой специальной логики (про-

верки достоверности данных, отключения пузырькового распространения и т.п.) перед запуском пузырькового аналога. В качестве примера предположим, что есть элемент `TextBox`, который должен содержать только числовые данные. Можно было бы обработать событие `PreviewKeyDown`; если выяснится, что пользователь ввел нечисловые данные, то пузырьковое событие легко отменить, установив свойство `Handled` в `true`.

Как несложно было предположить, при построении специального элемента управления, который поддерживает специальные события, событие допускается реализовать так, чтобы оно могло распространяться пузырьковым (или туннельным) образом по дереву разметки XAML. В настоящей главе мы не рассматриваем процесс создания специальных маршрутизируемых событий (хотя он не особо отличается от построения специального свойства зависимости). Если интересно, загляните в раздел "Routed Events Overview" ("Обзор маршрутизируемых событий") документации .NET Framework 4.6 SDK, где предлагается несколько обучающих руководств, которые помогут в освоении этой темы.

Исходный код. Проект `WPFRoutedEvents` доступен в подкаталоге `Chapter_27`.

Более глубокий взгляд на API-интерфейсы и элементы управления WPF

В оставшемся материале главы будет построено новое приложение WPF с применением Visual Studio. Целью является создание пользовательского интерфейса, который состоит из виджета `TabControl`, содержащего набор вкладок. Каждая вкладка будет иллюстрировать несколько новых элементов управления WPF и интересные API-интерфейсы, которые могут быть задействованы в разрабатываемых проектах. Попутно вы также узнаете о дополнительных возможностях визуальных конструкторов WPF из Visual Studio.

Работа с элементом управления `TabControl`

Первым делом создадим новый проект WPF Application по имени `WpfControlsAndAPIs`. Как упоминалось ранее, начальное окно будет содержать элемент управления `TabControl` с четырьмя вкладками, каждая из которых отображает набор связанных элементов управления и/или API-интерфейсов WPF. Перетащим элемент управления `TabControl` из панели инструментов Visual Studio на поверхность визуального конструктора, изменим размер компонента так, чтобы он занимал большую часть области отображения, и переименуем его в `myTabSystem`.

Вы заметите, что два элемента типа вкладок предоставляются автоматически. Для добавления дополнительных вкладок нужно просто щелкнуть правой кнопкой мыши на узле `TabControl` в окне Document Outline и выбрать в контекстном меню пункт Add TabItem (Добавить `TabItem`); можно также щелкнуть правой кнопкой мыши на самом элементе `TabControl` в визуальном конструкторе и выбрать тот же самый пункт меню. Добавим две дополнительных вкладки, используя любой из подходов (на рис. 27.23 демонстрируется подход с визуальным конструктором).

По очереди выберем каждый элемент управления `TabItem` (на поверхности визуального конструктора или в окне Document Outline) и изменим его свойство `Header`, указывая `Ink API`, `Documents`, `Data Binding` и `DataGrid`. Окно визуального конструктора должно выглядеть подобным тому, что показано на рис. 27.24.

Снова по очереди щелкнем на каждой вкладке и с помощью окна Properties назначим ей подходящее уникальное имя. Имейте в виду, что выбранная для редактирования вкладка становится активной, и ее содержимое можно формировать, перетаскивая элементы управления из панели инструментов.

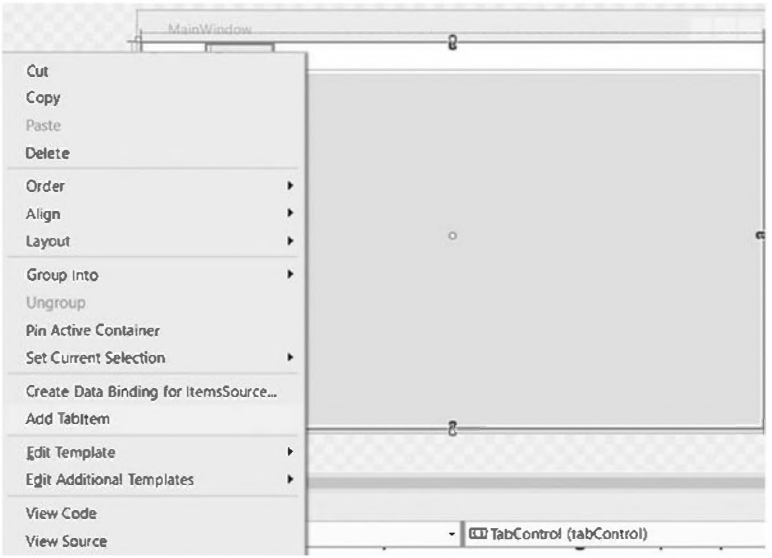


Рис. 27.23. Визуальное добавление элементов `TabItem`

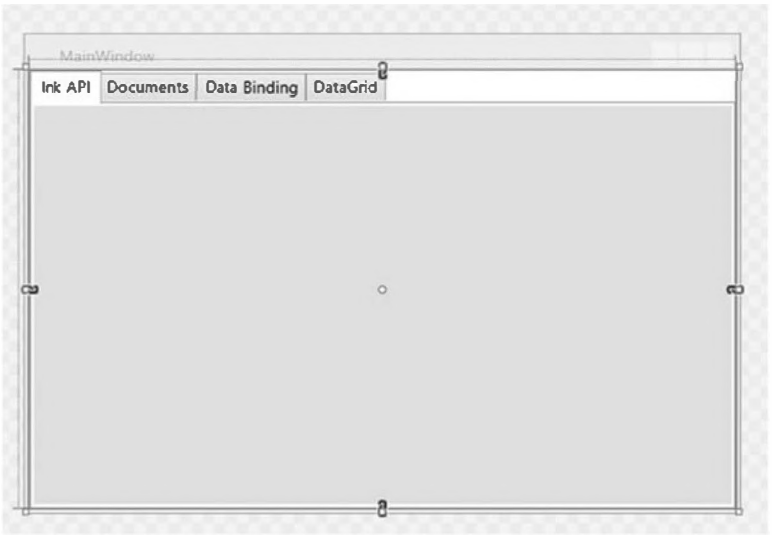


Рис. 27.24. Начальная компоновка системы вкладок

Перед тем, как приступить к проектированию вкладок, давайте взглянем на разметку XAML, которую сгенерировала IDE-среда. Разметка должна выглядеть похожей на приведенную ниже (отличия могут быть только в установленных свойствах):

```
<TabControl x:Name="myTabControl" HorizontalAlignment="Left" Height="280"
            Margin="10,10,0,0" VerticalAlignment="Top" Width="489">
   <TabItem Header="Ink API">
      <Grid Background="#FFE5E5E5"/>
   </TabItem>
   <TabItem Header="Documents">
      <Grid Background="#FFE5E5E5"/>
   </TabItem>
```

```
    <TabItem Header="Data Binding" HorizontalAlignment="Left" Height="20"
             VerticalAlignment="Top" Width="95" Margin="-2,-2,-36,0">
       <Grid Background="#FFE5E5E5"/>
    </TabItem>
    <TabItem Header="DataGrid" HorizontalAlignment="Left" Height="20"
             VerticalAlignment="Top" Width="74" Margin="-2,-2,-15,0">
       <Grid Background="#FFE5E5E5"/>
    </TabItem>
</TabControl>
```

Располагая определением основного элемента управления `TabControl`, можно проработать детали каждой вкладки, одновременно изучая дополнительные средства API-интерфейса WPF.

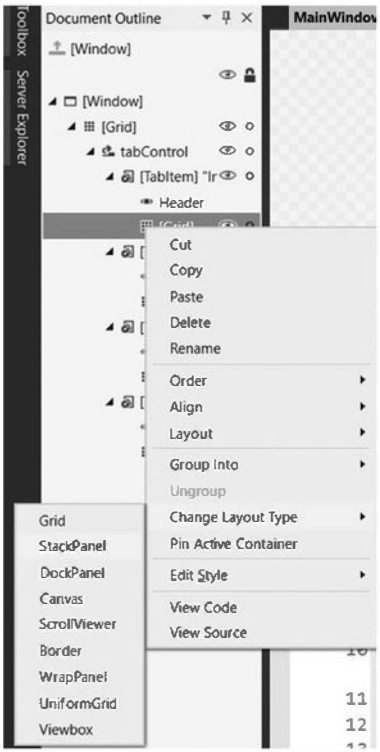


Рис. 27.25. Смена диспетчера компоновки для первой вкладки

Построение вкладки Ink API

Первая вкладка раскрывает общую роль интерфейса Ink API, который позволяет легко встраивать в программу функциональность рисования. Конечно, его применение не ограничивается приложениями для рисования; этот API-интерфейс можно использовать для разнообразных целей, включая фиксацию рукописного ввода посредством пера на Tablet PC.

Начнем с нахождения в окне Document Outline узла, представляющего вкладку `Ink API`, и раскроем его. Вы должны увидеть, что стандартным диспетчером компоновки для данного элемента управления `TabItem` является `<Grid>`. Щелкнем правой кнопкой мыши на нем и поменяем диспетчер компоновки на `StackPanel` (рис. 27.25).

Проектирование панели инструментов

Удостоверившись в том, что узел `StackPanel` в текущий момент выбран в окне Document Outline, вставим новый элемент управления `ToolBar` по имени `inkToolbar`. Выберем узел `inkToolbar` для редактирования и установим свойство `Height` элемента `Toolbar` в `60` (значение свойства `Width` оставим без изменений). В области Common (Общие) окна Properties щелкнем на кнопке с многоточием возле свойства Items (Collection), как показано на рис. 27.26.

В результате щелчка на упомянутой кнопке откроется диалоговое окно, которое позволяет выбрать элементы управления для добавления в `ToolBar`. Щелкнем на раскрывающемся списке, расположенном по центру в нижней части диалогового окна, и добавим три элемента управления `RadioButton`. Посредством встроенного в диалоговое окно редактора свойств для каждого элемента `RadioButton` установим свойство `Height` в `50`, а свойство `Width` в `100` (эти свойства находятся в области Layout (Компоновка)). Также установим свойства `Content` (в области Common) элементов `RadioButton` соответственно в `Ink Mode!`, `Erase Mode!` и `Select Mode!` (рис. 27.27). После добавления трех элементов управления `RadioButton` добавим элемент управления `Separator` с применением раскрывающегося списка в редакторе Items (Элементы).

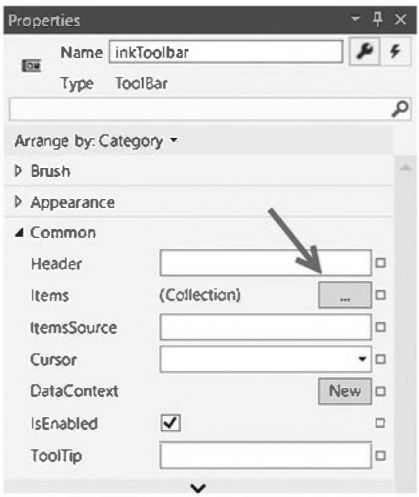


Рис. 27.26. Заполнение `ToolBar` элементами начинается здесь

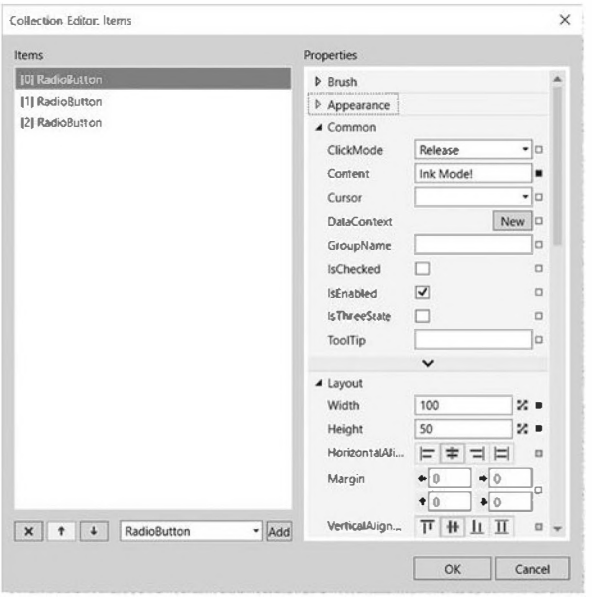


Рис. 27.27. Конфигурирование элементов управления `RadioButton`

Теперь осталось добавить финальный элемент управления `ComboBox` (не `ComboBoxItem`), присутствующий в раскрывающемся списке. Когда необходимо вставить нестандартные элементы управления в редакторе Items, следует просто выбрать в раскрывающемся списке вариант <Other Type> (<Другой тип>). Откроется редактор Select Object (Выберите объект), в котором можно ввести имя желаемого элемента управления. Удостоверившись в том, что флажок Show all assemblies (Показать все сборки) отмечен, найдем интересующий элемент управления (рис. 27.28).

Установим свойство `Width` элемента `ComboBox` в `100` и добавим в `ComboBox` три объекта `ComboBoxItem` с применением свойства Items(Collection) в области Common окна Properties. Укажем для свойства `Content` элементов `ComboBoxItem` строки `Red`, `Green` и `Blue`.

Закроем редактор, чтобы возвратиться в визуальный конструктор окна. Последней задачей в этом разделе будет использование свойства `Name` для назначения имен переменных новым элементам. Назовем три элемента `RadioButton` так: `inkRadio`, `selectRadio` и `eraseRadio`. Элементу `ComboBox` назначим имя `comboColors`. Когда все сделано, разметка XAML для первого элемента управления `TabItem` должна выглядеть следующим образом (может понадобиться скорректировать ширину и высоту):

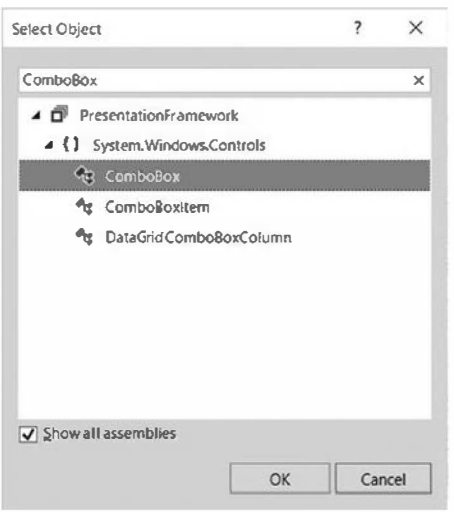


Рис. 27.28. Использование редактора Select Object для добавления уникальных элементов в панель инструментов

```
<TabItem Header="Ink API">
  <StackPanel Background="#FFE5E5E5">
    <ToolBar x:Name="inkToolbar" HorizontalAlignment="Left"
             Width="479" Height="60">
      <RadioButton x:Name="inkRadio" Content="Ink Mode!"
                   Height="50" Width="100"/>
      <RadioButton x:Name="selectRadio" Content="Erase Mode!"
                   Height="50" Width="100"/>
      <RadioButton x:Name="eraseRadio" Content="Select Mode!"
                   Height="50" Width="100"/>
      <Separator/>
      <ComboBox x:Name="comboColors"
                Width="100">
        <ComboBoxItem Content="Red"/>
        <ComboBoxItem Content="Green"/>
        <ComboBoxItem Content="Blue"/>
      </ComboBox>
    </ToolBar>
  </StackPanel>
</TabItem>
```

На заметку! При построении панели инструментов с помощью IDE-среды вы можете обнаружить, что решить задачу удастся намного быстрее, если просто вручную редактировать разметку XAML. Если вам удобно вводить код разметки напрямую, то можете так и поступать. Тем не менее, настоятельно рекомендуется выделить время на освоение окна Properties в Visual Studio. Вы увидите, что в нем предлагается несколько удобных расширенных возможностей.

Элемент управления RadioButton

В данном примере необходимо, чтобы добавленные три элемента управления `RadioButton` были взаимно исключающими. В других инфраструктурах для построения графических пользовательских интерфейсов такие связанные элементы требуют помещения в одну групповую рамку. Поступать подобным образом в WPF нет нужды. Взамен элементам управления просто назначается то же самое *групповое имя*. Это удобно, поскольку связанные элементы не обязаны физически находиться внутри одной области, а могут располагаться где угодно в окне.

В визуальном конструкторе понадобится выбрать каждый элемент `RadioButton` (для выбора всех трех элементов на них нужно щелкать при нажатой клавише <Shift>) и установить свойство `GroupName` (в области Common окна Properties) в `InkMode`.

Когда элемент управления `RadioButton` не размещен внутри родительского элемента управления типа панели, он будет принимать вид, идентичный элементу управления `Button`! Однако в отличие от `Button`, класс `RadioButton` имеет свойство `IsChecked`, значения которого переключаются между `true` и `false`, когда конечный пользователь щелкает на элементе пользовательского интерфейса. К тому же элемент управления `RadioButton` предоставляет два события (`Checked` и `Unchecked`), которые можно применять для перехвата такого изменения состояния.

Чтобы сконфигурировать внешний вид элементов управления `RadioButton`, подобный типичным переключателям, выделим их в визуальном конструкторе, последовательно щелкая при нажатой клавише <Shift>, щелкнем правой кнопкой мыши и выберем в контекстном меню пункт Group Into⇨Border (Сгруппировать в⇨`Border`), как показано на рис. 27.29.

Теперь все готово к тестированию программы, для чего понадобится нажать клавишу <F5>. Должны отобразиться три взаимно исключающих переключателя и раскрывающийся список с тремя элементами (рис. 27.30).

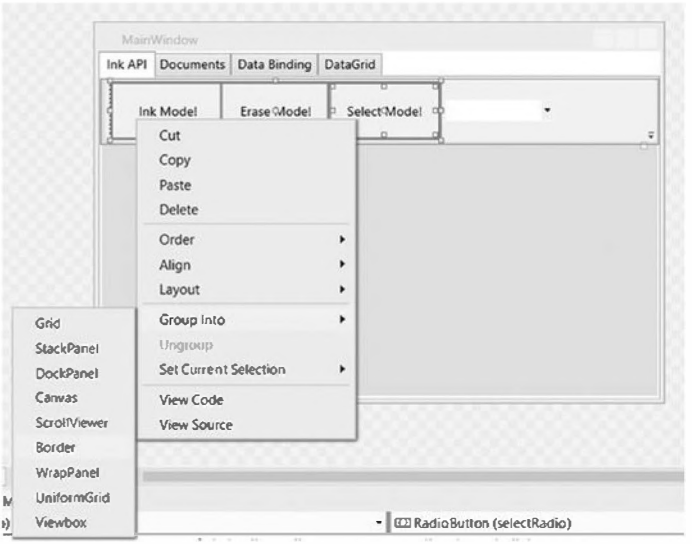


Рис. 27.29. Группирование элементов в элемент управления `Border`

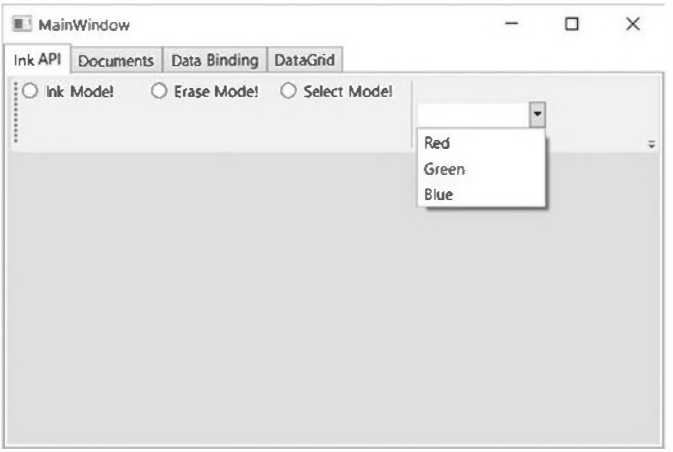


Рис. 27.30. Завершенная система панели инструментов

Обработка событий для вкладки `Ink API`

Следующая задача для вкладки `Ink API` связана с обработкой события `Click` для каждого элемента управления `RadioButton`. Как это делалось в других проектах WPF, нужно просто щелкнуть на значке с изображением молнии в окне Properties среды Visual Studio и ввести имена обработчиков событий. С помощью такого приема свяжем событие `Click` каждого элемента управления `RadioButton` с тем же самым обработчиком по имени `RadioButtonClicked`. После обработки всех трех событий `Click` обработаем событие `SelectionChanged` элемента управления `ComboBox`, используя обработчик по имени `ColorChanged`. В результате должен получиться такой код C#:

```
public partial class MainWindow : Window
{
  public MainWindow()
```

```
    {
      this.InitializeComponent();
      // Вставить здесь код, требуемый при создании объекта.
    }
    private void RadioButtonClicked(object sender,RoutedEventArgs e)
    {
      // TODO: Добавить сюда реализацию обработчика событий.
    }
    private void ColorChanged(object sender,SelectionChangedEventArgs e)
    {
      // TODO: Добавить сюда реализацию обработчика событий.
    }
}
```

Эти обработчики будут реализованы позже, так что пока оставим их пустыми.

Элемент управления `InkCanvas`

Для завершения пользовательского интерфейса этой вкладки необходимо поместить элемент управления `InkCanvas` в `StackPanel`, чтобы он появился под только что созданным элементом `ToolBar`. К сожалению, панель инструментов Visual Studio по умолчанию *не* отображает все возможные компоненты WPF. Хотя можно было бы просто ввести необходимую разметку XAML, вы должны знать, что на самом деле есть возможность обновлять элементы, подлежащие отображению в панели инструментов.

Для этого щелкнем правой кнопкой мыши где-нибудь в области панели инструментов и выберем в контекстном меню пункт Choose Items (Выбрать элементы). Вскоре появится список возможных компонентов для добавления в панель инструментов. Нас интересует добавление элемента управления `InkCanvas` (рис. 27.31).

Выберем `StackPanel` для объекта `tabInk` в окне Document Outline и затем добавим элемент управления `InkCanvas` по имени `myInkCanvas`. Растянем новый элемент управления так, чтобы он занял большую часть области вкладки. Кроме того, с помощью редактора Brushes (Кисти) можно задать для `InkCanvas` уникальный цвет фона (более подробно редактор Brushes рассматривается в следующей главе). После этого запустим программу, нажав <F5>. Поверхность холста уже позволяет рисовать графические данные за счет нажатия левой кнопки и перемещения курсора мыши (рис. 27.32).

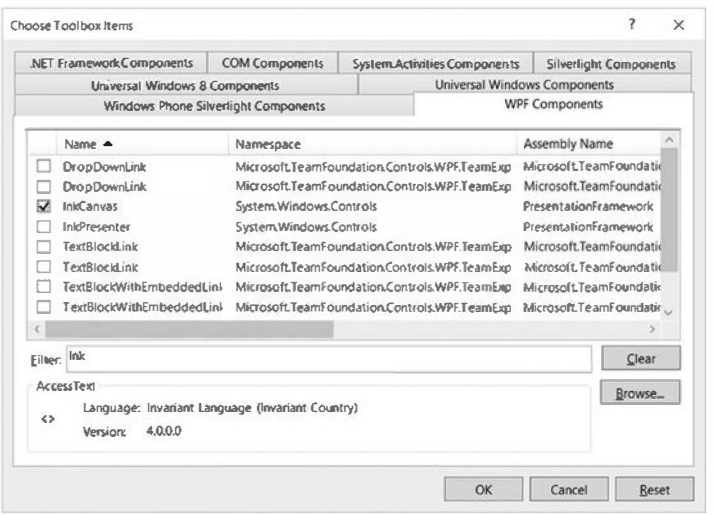


Рис. 27.31. Добавление новых компонентов в панель инструментов Visual Studio

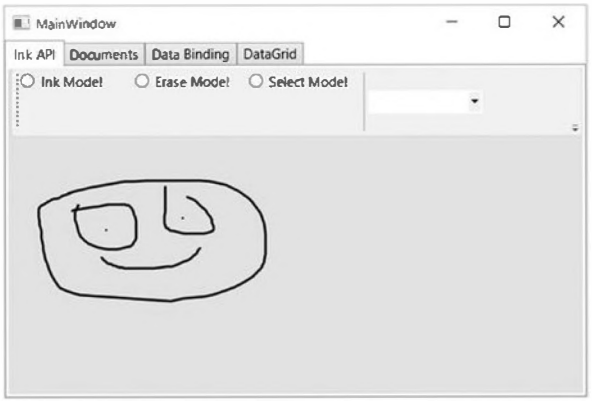


Рис. 27.32. Элемент управления `InkCanvas` в действии

Элемент управления `InkCanvas` обеспечивает нечто большее, чем просто рисование линий с помощью мыши (или пера); он также поддерживает несколько уникальных режимов редактирования, управляемых свойством `EditingMode`. Этому свойству можно присвоить любое значение из связанного перечисления `InkCanvasEditingMode`. В данном примере нас интересует режим `Ink`, принятый по умолчанию, который только что демонстрировался; режим `Select`, позволяющий пользователю выбирать с помощью мыши область для перемещения или изменения размера; и режим `EraseByStroke`, который удаляет предыдущий штрих мыши.

На заметку! *Штрих* — это визуализация, которая происходит во время одиночной операции нажатия и отпускания кнопки мыши. Элемент управления `InkCanvas` сохраняет все штрихи в объекте `StrokeCollection`, который доступен с применением свойства `Strokes`.

Обновим обработчик `RadioButtonClicked()` следующей логикой, которая помещает `InkCanvas` в корректный режим в зависимости от выбранного переключателя `RadioButton`:

```
private void RadioButtonClicked(object sender,RoutedEventArgs e)
{
  // В зависимости от того, какая кнопка отправила событие,
  // поместить InkCanvas в нужный режим оперирования.
  switch((sender as RadioButton)?.Content.ToString())
  {
    // Эти строки должны совпадать со значениями свойства Content
    // каждого элемента RadioButton.
    case "Ink Mode!":
      this.myInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
    break;
    case "Erase Mode!":
      this.myInkCanvas.EditingMode = InkCanvasEditingMode.EraseByStroke;
    break;
    case "Select Mode!":
      this.myInkCanvas.EditingMode = InkCanvasEditingMode.Select;
    break;
  }
}
```

Вдобавок установим `Ink` как стандартный режим в конструкторе окна. Там же установим стандартный выбор для `ComboBox` (более подробно этот элемент рассматривается в следующем разделе):

```
public MainWindow()
{
  this.InitializeComponent();
  // Установить режим Ink в качестве стандартного.
  this.myInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
  this.inkRadio.IsChecked = true;
  this.comboColors.SelectedIndex = 0;
}
```

Теперь запустим программу еще раз, нажав <F5>. Войдем в режим `Ink` и нарисуем что-нибудь. Затем перейдем в режим `Erase` и сотрем ранее нарисованное (курсор мыши автоматически примет вид стирающей резинки). Наконец, переключимся в режим `Select` и выберем несколько линий, используя мышь в качестве лассо.

Охватив элемент, его можно перемещать по поверхности холста, а также изменять размеры. На рис. 27.33 демонстрируются разные режимы в действии.

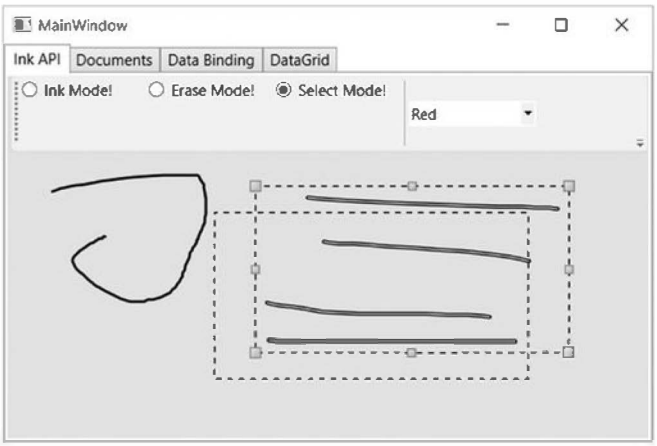


Рис. 27.33. Разные режимы редактирования элемента `InkCanvas` в действии

Элемент управления ComboBox

После заполнения элемента управления `ComboBox` (или `ListBox`) есть три способа определения выбранного в них элемента. Во-первых, когда необходимо найти числовой индекс выбранного элемента, должно применяться свойство `SelectedIndex` (отсчет начинается с нуля; значение `-1` представляет отсутствие выбора). Во-вторых, если требуется получить объект, выбранный внутри списка, то подойдет свойство `SelectedItem`. В-третьих, свойство `SelectedValue` позволяет получить значение выбранного объекта (обычно с помощью вызова `ToString()`).

Последний фрагмент кода, который понадобится добавить для данной вкладки, отвечает за изменение цвета штрихов, нарисованных в `InkCanvas`.

Свойство `DefaultDrawingAttributes` элемента `InkCanvas` возвращает объект `DrawingAttributes`, который позволяет конфигурировать многочисленные аспекты пера, включая его размер и цвет (помимо других настроек). Модифицируем код C# следующей реализацией метода `ColorChanged()`:

```
private void ColorChanged(object sender, SelectionChangedEventArgs e)
{
  // Получить выбранный элемент в раскрывающемся списке.
  string colorToUse =
    (this.comboColors.SelectedItem as ComboBoxItem)?.Content.ToString();
```

```
  // Изменить цвет, используемый для визуализации штрихов.
  this.myInkCanvas.DefaultDrawingAttributes.Color =
    (Color)ColorConverter.ConvertFromString(colorToUse);
}
```

Вспомните, что ComboBox содержит коллекцию ComboBoxItems. В сгенерированной разметке XAML присутствует такое определение:

```
<ComboBox x:Name="comboColors" Width="100" SelectionChanged="ColorChanged">
  <ComboBoxItem Content="Red"/>
  <ComboBoxItem Content="Green"/>
  <ComboBoxItem Content="Blue"/>
</ComboBox>
```

В результате обращения к свойству SelectedItem получается выбранный элемент ComboBoxItem, который хранится как экземпляр общего типа Object. После приведения Object к ComboBoxItem извлекается значение Content, которое будет строкой Red, Green или Blue. Эта строка затем преобразуется в объект Color с применением удобного служебного класса ColorConverter. Снова запустим программу. Теперь должна появиться возможность переключения между цветами при визуализации изображения.

Обратите внимание, что элементы управления ComboBox и ListBox могут также иметь сложное содержимое, а не только список текстовых данных. Чтобы получить представление о некоторых возможностях, откроем редактор XAML для окна и изменим определение элемента управления ComboBox, поместив в него набор элементов <StackPanel>, каждый из которых содержит <Ellipse> и <Label> (свойство Width элемента ComboBox установлено в 200):

```
<ComboBox x:Name="comboColors" Width="200" SelectionChanged="ColorChanged">
  <StackPanel Orientation ="Horizontal" Tag="Red">
    <Ellipse Fill ="Red" Height ="50" Width ="50"/>
    <Label FontSize ="20" HorizontalAlignment="Center"
           VerticalAlignment="Center" Content="Red"/>
  </StackPanel>

  <StackPanel Orientation ="Horizontal" Tag="Green">
    <Ellipse Fill ="Green" Height ="50" Width ="50"/>
    <Label FontSize ="20" HorizontalAlignment="Center"
           VerticalAlignment="Center" Content="Green"/>
  </StackPanel>

  <StackPanel Orientation ="Horizontal" Tag="Blue">
    <Ellipse Fill ="Blue" Height ="50" Width ="50"/>
    <Label FontSize ="20" HorizontalAlignment="Center"
           VerticalAlignment="Center" Content="Blue"/>
  </StackPanel>
</ComboBox>
```

В определении каждого элемента StackPanel присваивается значение свойству Tag, что является быстрым и удобным способом выявления, какой стек элементов был выбран пользователем (для этого существуют и лучшие способы, но пока достаточно такого). С указанной поправкой необходимо изменить реализацию метода ColorChanged():

```
private void ColorChanged(object sender, SelectionChangedEventArgs e)
{
  // Получить свойство Tag выбранного элемента StackPanel.
  string colorToUse = (this.comboColors.SelectedItem
     as StackPanel).Tag.ToString();
  ...
}
```

После запуска программы элемент управления ComboBox будет выглядеть так, как показано на рис. 27.34.

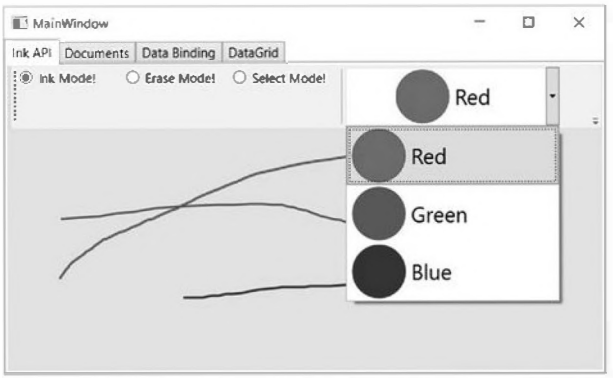


Рис. 27.34. Специальный элемент ComboBox, реализованный с помощью модели содержимого WPF

Сохранение, загрузка и очистка данных InkCanvas

Последняя часть вкладки Ink API позволит сохранять и загружать данные контейнера InkCanvas, а также очищать его содержимое. К настоящему моменту вы уже должны более уверенно чувствовать себя при проектировании пользовательских интерфейсов, поэтому инструкции будут краткими и по существу.

Начнем с импортирования пространств имен System.IO и System.Windows.Ink в файл кода. Затем добавим в ToolBar еще три элемента управления Button с именами btnSave, btnLoad и btnClear. После этого для каждого добавленного элемента обработаем событие Click и реализуем обработчики следующим образом:

```
private void SaveData(object sender, RoutedEventArgs e)
{
  // Сохранить все данные InkCanvas в локальном файле.
  using (FileStream fs = new FileStream("StrokeData.bin", FileMode.Create))
  {
    this.myInkCanvas.Strokes.Save(fs);
    fs.Close();
  }
}
private void LoadData(object sender, RoutedEventArgs e)
{
  // Наполнить StrokeCollection из файла.
  using(FileStream fs = new FileStream("StrokeData.bin",
    FileMode.Open, FileAccess.Read))
  {
    StrokeCollection strokes = new StrokeCollection(fs);
    this.myInkCanvas.Strokes = strokes;
  }
}
private void Clear(object sender, RoutedEventArgs e)
{
  // Очистить все штрихи.
  this.myInkCanvas.Strokes.Clear();
}
```

Теперь должна появиться возможность сохранения данных в файле, загрузки из файла и очистки InkCanvas от всех данных. На этом работа с первой вкладкой элемен-

та управления TabControl завершена, равно как и исследование интерфейса Ink API. Конечно, о технологии Ink API можно рассказать еще много чего, но теперь вы должны обладать достаточными знаниями, чтобы продолжить изучение темы самостоятельно. Далее мы приступаем к рассмотрению интерфейса Documents API в WPF.

Введение в интерфейс Documents API

Инфраструктура WPF поставляется со многими элементами управления, позволяющими вводить или отображать простые блоки текста, в число которых входят Label, TextBox, TextBlock и PasswordBox. Несмотря на удобство таких элементов управления, определенные приложения WPF требуют использования аккуратно сформатированных текстовых данных вроде тех, что можно обнаружить в файле Adobe PDF. Функциональность подобного рода в WPF предоставляет интерфейс Documents API; тем не менее, вместо файлового формата PDF в нем применяется формат XML Paper Specification (XPS).

Интерфейс Documents API можно использовать для конструирования готового к печати документа, задействовав несколько классов из пространства имен System.Windows.Documents. Здесь находятся типы, которые представляют части документа XPS: List, Paragraph, Section, Table, LineBreak, Figure, Floater и Span.

Блочные элементы и встроенные элементы

Формально элементы, добавляемые в документ XPS, относятся к одной из двух категорий: *блочные элементы* и *встроенные элементы*. Первая категория, блочные элементы, состоит из классов, которые расширяют базовый класс System.Windows.Documents.Block. Примерами блочных элементов могут служить List, Paragraph, BlockUIContainer, Section и Table. Классы из этой категории применяются для группирования вместе другого содержимого (например, в виде списка, содержащего данные абзаца, и абзаца, содержащего подабзацы для разного форматирования текста).

Вторая категория, встроенные элементы, состоит из классов, расширяющих базовый класс System.Windows.Documents.Inline. Встроенные элементы вкладываются внутрь блочного элемента (или, возможно, внутрь другого встроенного элемента внутри блочного элемента). Распространенными встроенными элементами являются Run, Span, LineBreak, Figure и Floater.

Данные классы имеют имена, которые можно встретить при построении форматированного документа в профессиональном редакторе. Как и любой другой элемент управления WPF, эти классы допускают конфигурирование в разметке XAML или в коде. Следовательно, можно либо объявлять пустой элемент <Paragraph>, который наполняется во время выполнения (далее в примере будет показано, как это делается), либо определять заполненный элемент <Paragraph> со статическим текстом.

Диспетчеры компоновки документа

Может показаться, что встроенные и блочные элементы разрешено помещать прямо в контейнер типа панели, такой как Grid, но на самом деле их необходимо упаковывать в элементы <FlowDocument> или <FixedDocument>.

Размещать элементы внутри FlowDocument идеально, когда нужно позволить конечному пользователю изменять способ представления данных на экране. Пользователь может делать это путем изменения масштаба текста или способа отображения данных (например, в виде одной длинной страницы или в виде пары колонок). Элемент FixedDocument лучше использовать для представления готовых к печати (WYSIWYG), неизменяемых документов.

В рассматриваемом примере мы будем иметь дело только с контейнером FlowDocument. После вставки встроенных и блочных элементов в объект FlowDocument

он будет помещен внутрь одного из поддерживающих XPS диспетчеров компоновки, которые кратко описаны в табл. 27.4.

Таблица 27.4. Диспетчеры компоновки XPS

Элемент управления типа панели	Описание
`FlowDocumentReader`	Отображает данные в элементе `FlowDocument` и добавляет поддержку масштабирования, поиска и компоновки содержимого в разнообразных формах
`FlowDocumentScrollViewer`	Отображает данные в элементе `FlowDocument`, но данные представлены как единый документ, просматриваемый с применением линеек прокрутки. Этот контейнер не поддерживает масштабирование, поиск или альтернативные режимы компоновки
`RichTextBox`	Отображает данные в элементе `FlowDocument` и добавляет поддержку редактирования со стороны пользователя
`FlowDocumentPageViewer`	Отображает документ постранично, т.е. одну страницу за раз. Данные можно масштабировать, но поиск отсутствует

Самый полнофункциональный способ отображения элемента `FlowDocument` предусматривает его помещение внутрь диспетчера `FlowDocumentReader`. В результате пользователь получает возможность изменять компоновку, искать слова в документе и масштабировать отображение данных с помощью предоставленного пользовательского интерфейса. Одно из ограничений этого контейнера (а также контейнеров `FlowDocumentScrollViewer` и `FlowDocumentPageViewer`) связано с тем, что отображаемое содержимое допускает только чтение. Однако если необходимо разрешить конечному пользователю вводить новую информацию в `FlowDocument`, то его можно упаковать в элемент управления `RichTextBox`.

Построение вкладки `Documents`

Щелкнем на вкладке `Documents` элемента `TabItem` и откроем ее в визуальном конструкторе для редактирования. Здесь уже должен присутствовать стандартный элемент управления `<Grid>`, который является непосредственным дочерним элементом `TabItem`; с помощью окна Document Outline заменим его контейнером `StackPanel`. На вкладке `Documents` будет отображаться элемент `FlowDocument`, который позволит пользователю выделять текст и добавлять аннотации, используя интерфейс Sticky Notes API ("клейкие" заметки).

Начнем с определения следующего элемента управления `ToolBar`, который имеет три простых (неименованных) элемента управления `Button`. Позже к этим элементам управления будут привязаны команды, так что ссылаться на них в коде не придется (можно либо ввести код разметки XAML вручную, либо применить IDE-среду).

```
<TabItem x:Name="tabDocuments" Header="Documents"
         VerticalAlignment="Bottom" Height="20">
  <StackPanel>
    <ToolBar>
      <Button BorderBrush="Green" Content="Add Sticky Note"/>
      <Button BorderBrush="Green" Content="Delete Sticky Notes"/>
      <Button BorderBrush="Green" Content="Highlight Text"/>
    </ToolBar>
  </StackPanel>
</TabItem>
```

При желании панель инструментов Visual Studio можно модифицировать, включив в нее элемент управления FlowDocumentReader (с использованием того же самого приема, как при добавлении InkCanvas), или обновить текущую разметку TabItem вручную в редакторе XAML.

Добавим элемент FlowDocumentReader в StackPanel, переименуем его в myDocumentReader и растянем на всю поверхность StackPanel. Поместим в этот новый компонент пустой элемент <FlowDocument>:

```
<FlowDocumentReader x:Name="myDocumentReader" Height="269.4">
  <FlowDocument/>
</FlowDocumentReader>
```

Далее можно добавить к элементу <FlowDocument> классы документа (например, List, Paragraph, Section, Table, LineBreak, Figure, Floater и Span). Вот один из возможных способов конфигурирования FlowDocument:

```
<FlowDocumentReader x:Name="myDocumentReader" Height="269.4">
  <FlowDocument>
    <Section Foreground = "Yellow" Background = "Black">
      <Paragraph FontSize = "20">
        Here are some fun facts about the WPF Documents API!
      </Paragraph>
    </Section>
    <List/>
    <Paragraph/>
  </FlowDocument>
</FlowDocumentReader>
```

Если теперь запустить программу (нажав <F5>), то должна появиться возможность изменения масштаба отображения документа (с применением ползунка в нижнем правом углу), поиска ключевых слов (с помощью редактора поиска, расположенного внизу слева) и вывода данных в одном из трех режимов (с использованием кнопок компоновки).

Прежде чем перейти к следующему шагу, можно отредактировать разметку XAML, чтобы вместо FlowDocumentReader применялся другой контейнер FlowDocument, такой как FlowDocumentScrollViewer или RichTextBox. После этого нужно снова запустить приложение и обратить внимание на отличия в обработке данных документа. По завершении эксперимента необходимо возвратиться к использованию типа FlowDocumentReader.

Наполнение FlowDocument с помощью кода

Теперь давайте построим блок List и оставшийся блок Paragraph в коде. Это важно знать, т.к. элемент FlowDocument часто требуется наполнять на основе пользовательского ввода, внешних файлов, информации из базы данных или другого источника. Первым делом в редакторе XAML назначим элементам List и Paragraph подходящие имена, чтобы можно было получать доступ к ним в коде:

```
<List x:Name="listOfFunFacts"/>
<Paragraph x:Name="paraBodyText"/>
```

В файле кода определим новый закрытый метод по имени PopulateDocument(). Он будет добавлять к List набор элементов ListItem, каждый из которых имеет элемент Paragraph с единственным элементом Run. Кроме того, этот вспомогательный метод динамически строит сформатированный абзац с применением трех отдельных объектов Run, как показано ниже:

```
private void PopulateDocument()
{
  // Добавить некоторые данные в элемент List.
  this.listOfFunFacts.FontSize = 14;
  this.listOfFunFacts.MarkerStyle = TextMarkerStyle.Circle;
  this.listOfFunFacts.ListItems.Add(new ListItem( new
    Paragraph(new Run("Fixed documents are for WYSIWYG print ready docs!"))));
  this.listOfFunFacts.ListItems.Add(new ListItem(
    new Paragraph(new Run("The API supports tables and embedded figures!"))));
  this.listOfFunFacts.ListItems.Add(new ListItem(
    new Paragraph(new Run("Flow documents are read only!"))));
  this.listOfFunFacts.ListItems.Add(new ListItem(new Paragraph(new Run
    ("BlockUIContainer allows you to embed WPF controls in the document!")
    )));

  // Добавить некоторые данные в элемент Paragraph.
  // Первая часть абзаца.
  Run prefix = new Run("This paragraph was generated ");

  // Середина абзаца.
  Bold b = new Bold();
  Run infix = new Run("dynamically");
  infix.Foreground = Brushes.Red;
  infix.FontSize = 30;
  b.Inlines.Add(infix);

  // Последняя часть абзаца.
  Run suffix = new Run(" at runtime!");

  // Добавить все части в коллекцию встроенных элементов Paragraph.
  this.paraBodyText.Inlines.Add(prefix);
  this.paraBodyText.Inlines.Add(infix);
  this.paraBodyText.Inlines.Add(suffix);
}
```

Вызовем этот метод в конструкторе окна. После этого можно запустить приложение и увидеть новое динамически сгенерированное содержимое документа (рис. 27.35).

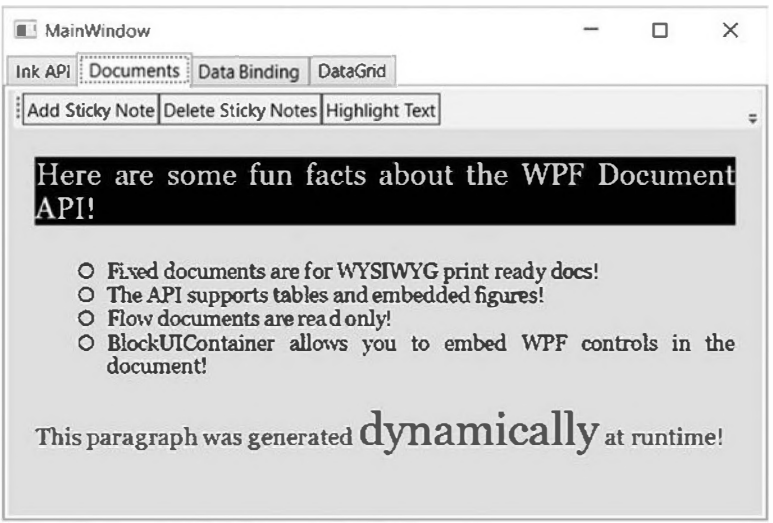


Рис. 27.35. Элемент управления `FlowDocumentReader`

Включение аннотаций и "клейких" заметок

Теперь можно строить документ с интересующими данными, используя разметку XAML и код C#; тем не менее, нужно еще кое-что сделать с тремя кнопками в панели инструментов для вкладки Documents. В составе WPF доступен набор команд, которые специально предназначены для применения с интерфейсом Documents API. С их помощью можно позволить пользователю выделять часть документа, а также добавлять аннотации в виде "клейких" заметок. Лучше всего то, что все это делается путем написания всего нескольких строк кода (и небольшого объема разметки).

Объекты команд для Documents API находятся в пространстве имен System.Windows.Annotations сборки PresentationFramework.dll. Таким образом, для использования таких объектов в разметке XAML понадобится определить специальное пространство имен XML в открывающем элементе <Window> (обратите внимание, что префиксом дескриптора является a):

```
<Window
  ...
  xmlns:a=
    "clr-namespace:System.Windows.Annotations;assembly=PresentationFramework"
  x:Class="WpfControlsAndAPIs.MainWindow"
  x:Name="Window"
  Title="MainWindow"
  Width="856" Height="383" mc:Ignorable="d"
    WindowStartupLocation="CenterScreen" >
  ...
</Window>
```

Модифицируем три определения <Button>, установив свойство Command каждого из них в соответствующую команду аннотаций:

```
<ToolBar>
  <Button BorderBrush="Green" Content="Add Sticky Note"
          Command="a:AnnotationService.CreateTextStickyNoteCommand"/>
  <Button BorderBrush="Green" Content="Delete Sticky Notes"
          Command="a:AnnotationService.DeleteStickyNotesCommand"/>
  <Button BorderBrush="Green" Content="Highlight Text"
          Command="a:AnnotationService.CreateHighlightCommand"/>
</ToolBar>
```

Последнее, что потребуется сделать — включить службы аннотации для объекта FlowDocumentReader, который был назван myDocumentReader. Добавим в класс еще один закрытый метод по имени EnableAnnotations(), который будет вызываться в конструкторе окна. Затем импортируем следующие пространства имен:

```
using System.Windows.Annotations;
using System.Windows.Annotations.Storage;
```

Теперь реализуем этот метод:

```
private void EnableAnnotations()
{
  // Создать объект AnnotationService, работающий с FlowDocumentReader.
  AnnotationService anoService = new AnnotationService(myDocumentReader);

  // Создать объект MemoryStream, который будет содержать аннотации.
  MemoryStream anoStream = new MemoryStream();

  // Создать XML-хранилище на основе MemoryStream. Этот объект можно
  // использовать для программного добавления, удаления или поиска аннотаций.
  AnnotationStore store = new XmlStreamStore(anoStream);
```

```
    // Включить службы аннотаций.
    anoService.Enable(store);
}
```

Класс `AnnotationService` позволяет заданному диспетчеру компоновки документа получить поддержку аннотаций. Перед тем, как вызывать метод `Enable()` этого объекта, необходимо предоставить местоположение объекта для хранения аннотированных данных, которые в настоящем примере являются областью памяти, представленной объектом `MemoryStream`. Обратите внимание, что объект `AnnotationService` соединяется с объектом `Stream` посредством `AnnotationStore`.

Запустим приложение. После выделения какого-нибудь текста можно щелкнуть на кнопке Add Sticky Note (Добавить "клейкую" заметку) и ввести некоторую информацию. Выделенный текст также можно подсвечивать (по умолчанию желтым цветом). Наконец, можно удалять заметки, выбирая их и щелкая на кнопке Delete Stricky Note (Удалить "клейкую" заметку). Результат тестового запуска показан на рис. 27.36.

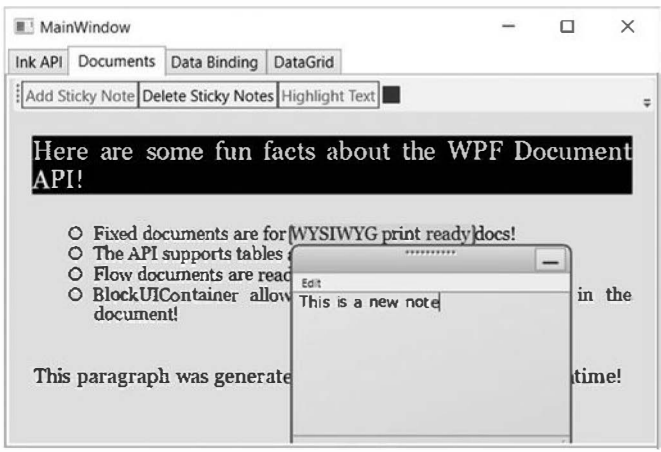


Рис. 27.36. Использование "клейких" заметок

Сохранение и загрузка потокового документа

Давайте завершим краткое введение в интерфейс Documents API рассмотрением простых процессов сохранения документа в файл и чтения документа из файла. Вспомните, что если объект `FlowDocument` не упакован в `RichTextBox`, то конечный пользователь не может редактировать документ; однако когда часть документа создается динамически во время выполнения, его может понадобиться сохранить для последующей работы. Возможность загрузки документа в стиле XPS также полезна во многих приложениях WPF, поскольку она позволяет определить пустой документ и загружать его на лету.

В следующем фрагменте разметки предполагается, что в панель инструментов вкладки `Documents` были добавлены два новых элемента `Button`, объявленные следующим образом (обратите внимание, что в разметке никакие события не обрабатываются):

```
<Button x:Name="btnSaveDoc" HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch" Width="75" Content="Save Doc"/>
<Button x:Name="btnLoadDoc" HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch" Width="75" Content="Load Doc"/>
```

Поместим в конструктор окна приведенные ниже лямбда-выражения для сохранения и загрузки данных `FlowDocument` (для получения доступа к классам `XamlReader` и `XamlWriter` потребуется импортировать пространство имен `System.Windows.Markup`):

```
public MainWindow()
{
  ...
  // Построить обработчики событий Click для сохранения
  // и загрузки документа нефиксированного формата.
  btnSaveDoc.Click += (o, s) =>
  {
    using(FileStream fStream = File.Open(
      "documentData.xaml", FileMode.Create))
    {
      XamlWriter.Save(this.myDocumentReader.Document, fStream);
    }
  };
  btnLoadDoc.Click += (o, s) =>
  {
    using(FileStream fStream = File.Open("documentData.xaml", FileMode.Open))
    {
      try
      {
        FlowDocument doc = XamlReader.Load(fStream) as FlowDocument;
        this.myDocumentReader.Document = doc;
      }
      catch(Exception ex) {MessageBox.Show(ex.Message, "Error Loading Doc!");}
    }
  };
}
```

Вот и все, что нужно сделать для сохранения документа (аннотации здесь не сохраняются; тем не менее, их можно сохранить с применением служб аннотаций). После щелчка на кнопке Save Doc (Сохранить документ) в папке `\bin\Debug` появится новый файл `*.xaml`, который содержит данные текущего документа.

На этом исследование интерфейса Documents API завершено. На самом деле в нем есть еще много аспектов, которые в главе не рассматривались, но вы получили достаточное представление об основах. В конце главы мы затронем нескольких тем, касающихся привязки данных, и завершим текущее приложение.

Введение в модель привязки данных WPF

Элементы управления часто служат целью для разнообразных операций привязки данных. Выражаясь просто, *привязка данных* представляет собой действие по подключению свойств элемента управления к значениям данных, которые могут изменяться на протяжении жизненного цикла приложения. Это позволяет элементу пользовательского интерфейса отображать состояние переменной в коде. Например, привязку данных можно использовать для решения следующих задач:

- отмечать флажок элемента управления `CheckBox` на основе булевского свойства заданного объекта;
- отображать в элементах `TextBox` информацию, извлеченную из реляционной базы данных;
- подключать элемент `Label` к целому числу, представляющему количество файлов в папке.

При работе со встроенным механизмом привязки данных WPF важно помнить о разнице между *источником* и *местом назначения* операции привязки. Как и можно было ожидать, источником операции привязки данных являются сами данные (булевское свойство, реляционные данные и т.д.), а местом назначения (или целью) — свойство элемента управления пользовательского интерфейса, в котором задействуется содержимое данных (вроде свойства элемента управления `CheckBox` или `TextBox`).

По правде говоря, применять инфраструктуру привязки данных WPF не обязательно. При желании самостоятельно реализовать логику привязки данных подключение между источником и местом назначения обычно потребует обработки различных событий и написания процедурного кода для соединения источника с целью. Например, если в окне имеется элемент `ScrollBar`, который должен отображать свое значение внутри `Label`, то можно обработать событие `ValueChanged` элемента `ScrollBar` и соответствующим образом обновить содержимое `Label`.

Однако привязку данных WPF можно использовать для подключения источника к цели непосредственно в разметке XAML (или в файле кода C#) без необходимости в обработке разнообразных событий или жесткого кодирования соединений между источником и целью. Вдобавок с помощью логики привязки данных можно обеспечить синхронизацию источника и цели в случае изменения значений данных.

Построение вкладки `Data Binding`

В окне Document Outline заменим `Grid` в третьей вкладке контейнером `StackPanel`. С применением панели инструментов и окна Properties среды Visual Studio создадим следующую начальную компоновку:

```
<TabItem x:Name="tabDataBinding" Header="Data Binding">
  <StackPanel Width="250">
    <Label Content="Move the scroll bar to see the current value"/>
  <!--Значение линейки прокрутки является источником этой привязки данных-->
    <ScrollBar x:Name="mySB" Orientation="Horizontal" Height="30"
           Minimum = "1" Maximum = "100" LargeChange="1" SmallChange="1"/>
    <!-- Содержимое метки будет привязано к
         линейке прокрутки -->
    <Label x:Name="labelSBThumb" Height="30"
           BorderBrush="Blue"
           BorderThickness="2" Content = "0"/>
  </StackPanel>
</TabItem>
```

Обратите внимание, что объект `<ScrollBar>` (названный здесь `mySB`) сконфигурирован с диапазоном от `1` до `100`. Цель заключается в том, чтобы при изменении положения ползунка линейки прокрутки (либо по щелчку на стрелке влево или вправо) элемент `Label` автоматически обновлялся текущим значением. В настоящий момент значение свойства `Content` элемента управления `Label` установлено в `"0"`; тем не менее, мы изменим его посредством операции привязки данных.

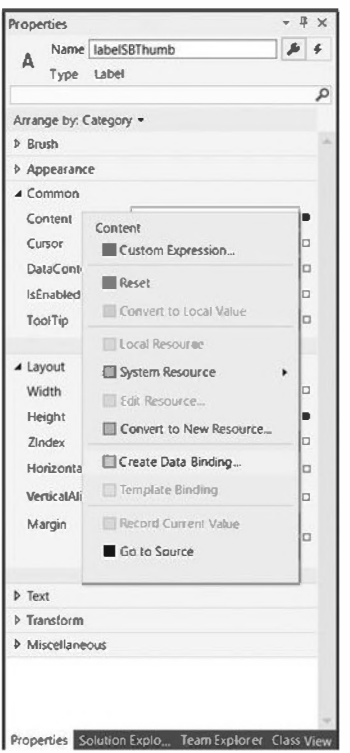


Рис. 27.37. Конфигурирование операции привязки данных

Установка привязки данных с использованием Visual Studio

Механизмом, обеспечивающим определение привязки в разметке XAML, является расширение разметки `{Binding}`. Установка привязки между элементами управления в Visual Studio производится легко. В рассматриваемом примере отыщем свойство `Content` объекта `Label` по имени `labelSBThumb` (в области Common окна Properties) и щелкнем на маленьком квадрате рядом со

свойством для открытия контекстного меню. Выберем в нем пункт Create Data Binding (Создать привязку данных), как показано на рис. 27.37.

Далее в раскрывающемся списке Binding type (Тип привязки) выберем вариант ElementName, что приведет к выдаче списка всех элементов в файле XAML, которые могут быть выбраны в качестве источника операции привязки данных. В дереве Element name (Имя элемента) найдем объект `ScrollBar` (по имени `mySB`), а в дереве Path (Путь) — свойство `Value` (рис. 27.38). Щелкнем на кнопке OK.

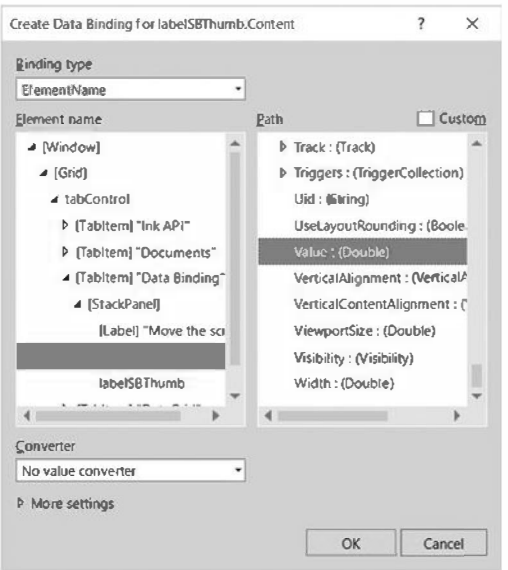


Рис. 27.38. Выбор объекта-источника и его свойства

После запуска программы обнаружится, что содержимое метки обновляется значением линейки прокрутки при перемещении ползунка. Взгляните на разметку XAML, сгенерированную инструментом привязки данных:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
       BorderThickness="2" Content = "{Binding Value, ElementName=mySB}"/>
```

Обратите внимание на значение, присвоенное свойству `Content` элемента `Label`. Значение `ElementName` здесь представляет источник операции привязки данных (объект `ScrollBar`), а первый элемент после ключевого слова `Binding` (т.е. `Value`) представляет (в этом случае) свойство элемента, который необходимо получить.

Если в прошлом вам приходилось работать с привязкой данных WPF, то вы можете ожидать увидеть применение лексемы `Path` для установки наблюдаемого свойства объекта. Например, следующая разметка также будет корректно обновлять `Label`:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
  BorderThickness="2" Content = "{Binding Path=Value, ElementName=mySB }"/>
```

По умолчанию аспект `Path=` операции привязки данных опускается, если только свойство не является подсвойством другого объекта (например, `myObject.MyProperty.Object2.Property2`).

Свойство `DataContext`

Для определения операции привязки данных в XAML может использоваться альтернативный формат, при котором допускается разбивать значения, указанные расшире-

нием разметки {Binding}, за счет явного указания в свойстве DataContext источника операции привязки:

```
<!-- Разбиение объекта и значения посредством DataContext -->
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
       BorderThickness="2"
       DataContext = "{Binding ElementName=mySB}"
       Content = "{Binding Path=Value}" />
```

Вывод будет идентичным. С учетом этого вас наверняка интересует, в каких случаях необходимо устанавливать свойство DataContext явно. Поступать так может быть удобно из-за того, что подэлементы способны наследовать свои значения в дереве разметки.

Подобным образом можно легко устанавливать один и тот же источник данных для семейства элементов управления, не повторяя избыточные XAML-фрагменты "{Binding ElementName=X, Path=Y}" во множестве элементов управления. Например, пусть в контейнер <StackPanel> текущей вкладки добавлен новый элемент Button (вскоре вы увидите, почему он имеет настолько большой размер):

```
<Button Content="Click" Height="140"/>
```

Для генерации привязок данных для множества элементов управления можно было бы применить Visual Studio, но вместо этого введем модифицированную разметку в редакторе XAML:

```
<!-- Обратите внимание, что StackPanel устанавливает свойство DataContext -->
<StackPanel Width="250" DataContext = "{Binding ElementName=mySB}">
  <Label Content="Move the scroll bar to see the current value"/>
  <ScrollBar Orientation="Horizontal" Height="30" Name="mySB"
             Maximum = "100" LargeChange="1" SmallChange="1"/>
  <!-- Теперь оба элемента пользовательского интерфейса работают
       со значением линейки прокрутки уникальными путями -->
  <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
         BorderThickness="2" Content = "{Binding Path=Value}"/>
  <Button Content="Click" Height="200"
          FontSize = "{Binding Path=Value}"/>
</StackPanel>
```

Здесь свойство DataContext в <StackPanel> устанавливается напрямую. Следовательно, при перемещении ползунка не только отображается текущее значение в элементе Label, но также увеличивается размер шрифта элемента Button в соответствие с тем же значением (на рис. 27.39 показан возможный вывод).

Преобразование данных с использованием IValueConverter

Вместо ожидаемого целого числа для представления положения ползунка в типе ScrollBar применяется значение double. По этой причине по мере перемещения ползунка внутри элемента Label будут отображаться разнообразные значения с плавающей точкой (вроде 61.0576923076923), которые выглядят не слишком интуитивно понятными для конечного пользователя, ожидающего увидеть целые числа (такие как 61, 62, 63 и т.д.).

Когда значение из операции привязки данных необходимо преобразовать в альтернативный формат, можно создать специальный класс, реализующий интерфейс IValueCVonverter, который доступен в пространстве имен System.Windows.Data. В этом интерфейсе определены два члена, позволяющие выполнять преобразование между источником и целью (в случае двунаправленной привязки). После определения такой класс можно использовать для дальнейшего уточнения процесса привязки данных.

Рис. 27.39. Привязка значения `ScrollBar` к элементам `Label` и `Button`

Предполагая, что в элементе управления `Label` должны отображаться целые числа, можно построить показанный ниже специальный класс преобразования. Выберем пункт меню Project⇨Add Class (Проект⇨Добавить класс) и добавим класс по имени `MyDoubleConverter`. Поместим в него следующий код:

```
class MyDoubleConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
                        System.Globalization.CultureInfo culture)
  {
    // Преобразовать значение double в int.
    double v = (double)value;
    return (int)v;
  }
  public object ConvertBack(object value, Type targetType, object parameter,
                            System.Globalization.CultureInfo culture)
  {
    // Поскольку заботиться о "двунаправленной" привязке
    // не нужно, просто возвратить значение value.
    return value;
  }
}
```

Метод `Convert()` вызывается при передаче значения от источника (`ScrollBar`) к цели (свойство `Content` элемента `Label`). Хотя он принимает много входных аргументов, для этого преобразования понадобится манипулировать только входным аргументом типа `object`, который представляет текущее значение `double`. Данный тип можно применять для приведения к целому и возврата нового числа.

Метод `ConvertBack()` будет вызываться, когда значение передается от цели к источнику (если включен двунаправленный режим привязки). Здесь мы просто возвращаем значение `value`. Это позволяет вводить в `TextBox` значение с плавающей точкой (например, `99.9`) и автоматически преобразовывать его в целочисленное значение (`99`), когда пользователь перемещает фокус из элемента управления. Такое "бесплатное" преобразование происходит из-за того, что метод `Convert()` будет вызываться еще раз после вызова `ConvertBack()`. Если просто возвратить `null` из `ConvertBack()`, то синхронизация привязки будет выглядеть нарушенной, т.к. элемент `TextBox` по-прежнему будет отображать число с плавающей точкой.

Установка привязок данных в коде

Располагая классом MyDoubleConverter, специальный преобразователь можно регистрировать с любым элементом управления, в котором желательно его использовать. Это допускается делать полностью в разметке XAML, но тогда потребуется определить ряд специальных объектных ресурсов, которые будут рассматриваться в следующей главе. Пока что мы можем зарегистрировать класс преобразователя данных в коде. Начнем с очистки текущего определения элемента управления <Label> внутри вкладки Data Binding, чтобы расширение разметки {Binding} больше не применялось:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
       BorderThickness="2" Content = "0"/>
```

В конструкторе окна вызовем новый закрытый вспомогательный метод по имени SetBindings(), код которого показан ниже:

```
private void SetBindings()
{
  // Создать объект Binding.
  Binding b = new Binding();
  // Зарегистрировать преобразователь, источник и путь.
  b.Converter = new MyDoubleConverter();
  b.Source = this.mySB;
  b.Path = new PropertyPath("Value");
  // Вызвать метод SetBinding объекта Label.
  this.labelSBThumb.SetBinding(Label.ContentProperty, b);
}
```

Единственной частью этого метода, которая может выглядеть несколько необычной, является вызов SetBinding(). Обратите внимание, что первый параметр обращается к статическому, доступному только для чтения полю ContentProperty класса Label. Как вы узнаете далее в главе, такая конструкция называется *свойством зависимости*. Пока просто имейте в виду, что при установке привязки в коде первый аргумент почти всегда требует указания имени класса, нуждающегося в привязке (Label в рассматриваемом случае), за которым следует обращение к внутреннему свойству с добавлением к его имени суффикса Property. Запустив приложение, можно удостовериться в том, что элемент Label отображает только целые числа.

Построение вкладки DataGrid

В предыдущем примере привязки данных иллюстрировался способ конфигурирования двух (или большего количества) элементов управления для участия в операции привязки данных. Наряду с тем, что это удобно, возможно также привязывать данные из файлов XML, базы данных и объектов в памяти. Чтобы завершить текущий пример, спроектируем финальную вкладку DataGrid, которая будет отображать информацию, извлеченную из таблицы Inventory базы данных AutoLot.

Как и с другими вкладками, начнем с замены текущего контейнера Grid контейнером StackPanel. Мы сделаем это путем прямого обновления разметки XAML в Visual Studio. Затем внутри нового элемента StackPanel определим элемент управления DataGrid по имени gridInventory:

```
<TabItem x:Name="tabDataGrid" Header="DataGrid">
  <StackPanel>
    <DataGrid x:Name="gridInventory" Height="288"/>
  </StackPanel>
</TabItem>
```

С помощью диспетчера пакетов NuGet добавим в проект инфраструктуру Entity Framework. Затем добавим ссылку на сборку `AutoLotDAL.dll`, созданную в главе 23 (с использованием Entity Framework). Содержимое файла `App.config` будет обновлено для учета Entity Framework, но придется еще вручную добавить строку подключения. Вот пример строки подключения:

```
<connectionStrings>
  <add name="AutoLotConnection"
  connectionString="data source=.\SQLEXPRESS2014;initialcatalog=AutoLot;
  integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"
  providerName="System.Data.SqlClient" />
</connectionStrings>
```

Откроем файл кода окна, добавим последний вспомогательный метод по имени `ConfigureGrid()` и вызовем его в конструкторе. Предполагая, что пространство имен `AutoLotDAL` было импортировано, останется лишь добавить несколько строк кода:

```
private void ConfigureGrid()
{
  using (var repo = new InventoryRepo())
  {
// Построить запрос LINQ, который извлекает некоторые данные из таблицы Inventory
   gridInventory.ItemsSource =
      repo.GetAll().Select(x=>new { x.CarId,x.Make,x.Color,x.PetName});
  }
}
```

Обратите внимание, что мы не привязываем `context.Inventories` к коллекции `ItemsSource` сетки напрямую; взамен строится запрос LINQ, который запрашивает те же самые данные в сущностях. Причина такого подхода в том, что объект `Inventory` *также* содержит дополнительные свойства Entity Framework, которые будут появляться в сетке, но не отображаются на физическую базу данных.

Если запустить проект в том виде, как есть, то можно увидеть исключительно простую сетку. Чтобы немного улучшить ее, отредактируем в окне Properties среды Visual Studio категорию `Rows` элемента управления `DataGrid`. Как минимум понадобится установить свойство `AlternationCount` в 2 и выбрать специальные цвета в свойствах `AlternatingRowBackground` и `RowBackground` с помощью интегрированного редактора. Внешний вид вкладки показан на рис. 27.40.

На этом текущий пример завершен. В последующих главах вы увидите в действии другие элементы управления, но к настоящему моменту вы должны чувствовать себя увереннее при построении пользовательских интерфейсов в Visual Studio и работе с разметкой XAML и кодом C#.

Исходный код. Проект `WpfControlsAndAPIs` доступен в подкаталоге `Chapter_27`.

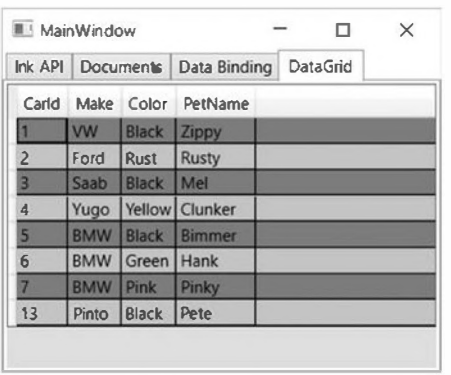


Рис. 27.40. Финальная вкладка проекта

Роль свойств зависимости

Подобно любому API-интерфейсу .NET внутри реализации инфраструктуры WPF применяется каждый член системы типов .NET (классы, структуры, интерфейсы, делегаты, перечисления) и каждый член типа (свойства, методы, события, константные данные, поля только для чтения и т.д.).

Однако WPF также поддерживает уникальную программную концепцию под названием *свойство зависимости*.

Как и "нормальное" свойство .NET (которое в литературе, посвященной WPF, часто называют *свойством CLR*), свойство зависимости можно устанавливать декларативно с использованием разметки XAML или программно в файле кода. Кроме того, свойства зависимости (подобно свойствам CLR) в конечном итоге предназначены для инкапсуляции полей данных класса и могут быть сконфигурированы как доступные только для чтения, только для записи или для чтения и записи.

Более интересно то, что практически всегда вы даже не будете знать о том, что в действительности устанавливаете (или читаете) свойство зависимости, а не свойство CLR! Например, свойства `Height` и `Width`, которые элементы управления WPF наследуют от класса `FrameworkElement`, а также член `Content`, унаследованный от класса `ControlContent`, на самом деле являются свойствами зависимости:

```
<!-- Установить три свойства зависимости -->
<Button x:Name = "btnMyButton" Height = "50" Width = "100" Content = "OK"/>
```

С учетом всех указанных сходств возникает вопрос: зачем нужно было определять в WPF новый термин для такой знакомой концепции? Ответ кроется в способе реализации свойства зависимости внутри класса. Пример кодирования будет показан ниже, а на высоком уровне все свойства зависимости создаются в следующей манере.

- Прежде всего, класс, который определяет свойство зависимости, должен иметь в своей цепочке наследования класс `DependencyObject`.
- Одиночное свойство зависимости представляется как открытое, статическое, допускающее только чтение поле типа `DependencyProperty`. По соглашению это поле именуется путем снабжения имени оболочки CLR (см. последний пункт этого списка) суффиксом в виде слова `Property`.
- Переменная типа `DependencyProperty` регистрируется посредством вызова статического метода `DependencyProperty.Register()`, который обычно происходит в статическом конструкторе или встраивается в объявление переменной.
- Наконец, в классе будет определено дружественное к XAML свойство CLR, которое вызывает методы, предоставляемые классом `DependencyObject`, для получения и установки значения.

После реализации свойства зависимости предлагают несколько мощных средств, которые применяются разнообразными технологиями WPF, в том числе привязкой данных, службами анимации, стилями, шаблонами и т.д. Мотивацией создания свойств зависимости было желание предоставить способ вычисления значений свойств на основе значений из других источников. Ниже приведен список основных преимуществ, которые далеко выходят за рамки простой инкапсуляции данных, обеспечиваемой свойствами CLR.

- Свойства зависимости могут наследовать свои значения от определения XAML родительского элемента. Например, если в открывающем дескрипторе `<Window>` определено значение для атрибута `FontSize`, то все элементы управления внутри этого `Window` по умолчанию будут иметь тот же самый размер шрифта.
- Свойства зависимости поддерживают возможность получать значения, которые установлены элементами внутри их области определения XAML, например, в случае установки элементом `Button` свойства `Dock` родительского контейнера `DockPanel`. (Вспомните из главы 26, что присоединяемые свойства делают именно это, поскольку являются разновидностью свойств зависимости.)

- Свойства зависимости позволяют инфраструктуре WPF вычислять значение на основе множества внешних значений, что может быть очень важно для служб анимации и привязки данных.
- Свойства зависимости предоставляют поддержку инфраструктуры для триггеров WPF (также довольно часто используемых при работе с анимацией и привязкой данных).

Имейте в виду, что во многих случаях вы будете взаимодействовать с существующим свойством зависимости в манере, идентичной работе с обычным свойством CLR (благодаря оболочке XAML). В предыдущем разделе, посвященном привязке данных, вы узнали, что если необходимо установить привязку данных в коде, то должен быть вызван метод `SetBinding()` на целевом объекте операции и указано *свойство зависимости*, с которым будет работать привязка:

```
private void SetBindings()
{
  Binding b = new Binding();
  b.Converter = new MyDoubleConverter();
  b.Source = this.mySB;
  b.Path = new PropertyPath("Value");
  // Указать свойство зависимости.
  this.labelSBThumb.SetBinding(Label.ContentProperty, b);
}
```

Вы увидите похожий код в главе 29 во время исследования запуска анимации в коде:

```
// Указать свойство зависимости.
rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
```

Потребность в построении специального свойства зависимости возникает только во время разработки собственного элемента управления WPF. Например, когда создается класс `UserControl` с четырьмя специальными свойствами, которые должны тесно интегрироваться с API-интерфейсом WPF, они должны быть реализованы с применением логики свойств зависимости.

В частности, если нужно, чтобы свойство было целью операции привязки данных или анимации, если оно должно уведомлять о своем изменении, если свойство должно быть в состоянии работать в качестве установщика в стиле WPF или получать свои значения от родительского элемента, то возможностей обычного свойства CLR для этого *не* достаточно. В случае использования обычного свойства другие программисты действительно могут получать и устанавливать его значение, но если они попытаются применить такое свойство внутри контекста службы WPF, то оно не будет работать ожидаемым образом. Поскольку никогда заранее не известно, как другие пожелают взаимодействовать со свойствами специальных классов `UserControl`, нужно выработать в себе привычку *всегда* определять свойства зависимости при построении специальных элементов управления.

Исследование существующего свойства зависимости

Прежде чем вы узнаете, каким образом создавать специальное свойство зависимости, давайте рассмотрим внутреннюю реализацию свойства `Height` класса `FrameworkElement`. Ниже приведен соответствующий код (с комментариями):

```
// FrameworkElement "является" DependencyObject.
public class FrameworkElement : UIElement, IFrameworkInputElement,
  IInputElement, ISupportInitialize, IHaveResources, IQueryAmbient
```

```
{
  ...
  // Статическое поле только для чтения типа DependencyProperty.
  public static readonly DependencyProperty HeightProperty;

  // Поле DependencyProperty часто регистрируется
  // в статическом конструкторе класса.
  static FrameworkElement()
  {
    ...
    HeightProperty = DependencyProperty.Register(
      "Height",
      typeof(double),
      typeof(FrameworkElement),
      new FrameworkPropertyMetadata((double) 1.0 / (double) 0.0,
        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),
      new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));
  }

  // Оболочка CLR, реализованная с использованием
  // унаследованных методов GetValue()/SetValue().
  public double Height
  {
    get { return (double) base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
  }
}
```

Как видите, по сравнению с обычными свойствами CLR свойства зависимости требуют немалого объема дополнительного кода. На самом деле зависимость может оказаться даже еще более сложной, чем здесь показано (к счастью, многие реализации проще свойства `Height`).

В первую очередь вспомните, что если в классе необходимо определить свойство зависимости, то он должен иметь в своей цепочке наследования `DependencyObject`, т.к. именно этот класс определяет методы `GetValue()` и `SetValue()`, применяемые в оболочке CLR. Из-за того, что класс `FrameworkElement` "является" `DependencyObject`, указанное требование удовлетворено. Далее вспомните, что сущность, где действительно хранится значение свойства (значение `double` в случае `Height`), представляется как открытое, статическое, допускающее только чтение поле типа `DependencyProperty`. По соглашению имя этого свойства должно всегда формироваться из имени связанной оболочки CLR с добавлением суффикса `Property`:

```
public static readonly DependencyProperty HeightProperty;
```

Учитывая, что свойства зависимости объявляются как статические поля, они обычно создаются (и регистрируются) внутри статического конструктора класса. Объект `DependencyProperty` создается посредством вызова статического метода `DependencyProperty.Register()`. Данный метод имеет множество перегруженных версий, но в случае свойства `Height` он вызывается следующим образом:

```
HeightProperty = DependencyProperty.Register(
  "Height",
  typeof(double),
  typeof(FrameworkElement),
  new FrameworkPropertyMetadata((double)0.0,
    FrameworkPropertyMetadataOptions.AffectsMeasure,
    new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),
  new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));
```

Первым аргументом, передаваемым методу DependencyProperty.Register(), является имя обычного свойства CLR класса (в этом случае Height), а второй аргумент содержит информацию о типе данных, который его инкапсулирует (double). Третий аргумент указывает информацию о типе класса, которому принадлежит свойство (FrameworkElement). Хотя такие сведения могут показаться избыточными (в конце концов, поле HeightProperty уже определено внутри класса FrameworkElement), это очень продуманный аспект WPF, поскольку он позволяет одному классу регистрировать свойства в другом классе (даже если его определение было запечатано).

Четвертый аргумент, передаваемый методу DependencyProperty.Register() в рассмотренном примере — это то, что действительно делает свойства зависимости уникальными. Здесь передается объект FrameworkPropertyMetadata, который описывает разнообразные детали относительно того, как инфраструктура WPF должна обрабатывать это свойство в плане уведомлений с помощью обратных вызовов (если свойству необходимо извещать других, когда его значение изменяется). Кроме того, объект FrameworkPropertyMetadata указывает различные параметры (представляемые перечислением FrameworkPropertyMetadataOptions), которые управляют тем, на что свойство воздействует (работает ли оно с привязкой данных, может ли наследоваться и т.д.). В данном случае аргументы конструктора FrameworkPropertyMetadata можно описать так:

```
new FrameworkPropertyMetadata(
  // Стандартное значение свойства.
  (double)0.0,

  // Параметры метаданных.
  FrameworkPropertyMetadataOptions.AffectsMeasure,

  // Делегат, который указывает на метод, вызываемый при изменении свойства.
  new PropertyChangedCallback(FrameworkElement.OnTransformDirty)
)
```

Поскольку последний аргумент конструктора FrameworkPropertyMetadata является делегатом, обратите внимание, что он указывает на статический метод OnTransformDirty() класса FrameworkElement. Код этого метода здесь не приводится, но имейте в виду, что при создании специального свойства зависимости всегда можно указывать делегат PropertyChangeCallback, нацеленный на метод, который будет вызываться в случае изменения значения свойства.

Возвратимся к финальному параметру метода DependencyProperty.Register() — второму делегату типа ValidateVakueCallback, указывающему на метод класса FrameworkElement, который вызывается для проверки достоверности значения, присваиваемого свойству:

```
new ValidateValueCallback(FrameworkElement.IsWidthHeightValid)
```

Этот метод содержит логику, которую обычно ожидают найти в блоке установки значения свойства (подробнее об этом речь пойдет в следующем разделе):

```
private static bool IsWidthHeightValid(object value)
{
  double num = (double) value;
  return ((!DoubleUtil.IsNaN(num) && (num >= 0.0))
    && !double.IsPositiveInfinity(num));
}
```

После того, как объект DependencyProperty зарегистрирован, остается упаковать поле в обычное свойство CLR (Height в рассматриваемом случае). Тем не менее, обратите внимание, что блоки get и set не просто возвращают или устанавливают значение

double переменной-члена уровня класса, а делают это косвенно с использованием методов GetValue() и SetValue() базового класса System.Windows.DependencyObject:

```
public double Height
{
  get { return (double) base.GetValue(HeightProperty); }
  set { base.SetValue(HeightProperty, value); }
}
```

Важные замечания относительно оболочек свойств CLR

Чтобы подвести итоги сказанному до сих пор, отметим, что свойства зависимости выглядят как обычные свойства, когда вы извлекаете или устанавливаете их значения в разметке XAML либо в коде, но "за кулисами" они реализованы с помощью намного более замысловатых приемов кодирования. Вспомните, что основным назначением этого процесса является построение специального элемента управления, имеющего специальные свойства, которые должны быть интегрированы со службами WPF, требующими взаимодействия через свойства зависимости (например, анимацией, привязкой данных и стилями).

Несмотря на то что часть реализации свойства зависимости предусматривает определение оболочки CLR, *вы никогда не должны помещать логику проверки достоверности в блок set*. Если уж на то пошло, оболочка CLR свойства зависимости *не должна делать ничего кроме вызовов GetValue() или SetValue()*.

Дело в том, что исполняющая среда WPF сконструирована таким образом, что если написать разметку XAML, которая выглядит как установка свойства, например:

```
<Button x:Name="myButton" Height="100" .../>
```

то исполняющая среда вообще обойдет блок установки свойства Height и *напрямую* вызовет метод SetValue()! Причина такого необычного поведения связана с простым приемом оптимизации. Если бы исполняющая среда WPF обращалась к блоку установки свойства Height, то ей пришлось бы во время выполнения посредством рефлексии выяснять, где находится поле DependencyProperty (указанное в первом аргументе SetValue()), ссылаться на него в памяти и т.д. То же самое остается справедливым и при написании разметки XAML, которая извлекает значение свойства Height — метод GetValue() будет вызываться напрямую. Но раз так, то зачем вообще строить оболочку CLR? Дело в том, что XAML в WPF не позволяет вызывать функции в разметке, поэтому следующий фрагмент приведет к ошибке:

```
<!-- Ошибка! Вызывать методы в XAML-разметке WPF нельзя! -->
<Button x:Name="myButton" this.SetValue("100") .../>
```

На самом деле установку или получение значения в разметке с применением оболочки CLR следует считать способом сообщения исполняющей среде WPF о необходимости вызова методов GetValue()/SetValue(), т.к. напрямую делать это в разметке невозможно. А что, если обратиться к оболочке CLR в коде, как показано ниже?

```
Button b = new Button();
b.Height = 10;
```

В этом случае, если блок установки свойства Height содержит какой-то код помимо вызова SetValue(), то он *должен* выполниться, потому что оптимизация синтаксического анализатора XAML в WPF не задействуется.

Запомните основное правило: при регистрации свойства зависимости используйте делегат ValidateValueCallback для указания на метод, который выполняет проверку достоверности данных. Это гарантирует корректное поведение независимо от того, что применяется для получения/установки свойства зависимости — разметка XAML или код.

Построение специального свойства зависимости

Если вы уже слегка запутались в данном месте главы, то такая реакция совершенно нормальна. Создание свойств зависимости может требовать некоторого времени на привыкание. Хорошо это или плохо, но подобным образом выглядит часть процесса построения многих специальных элементов управления WPF, так что давайте посмотрим, как создается свойство зависимости.

Начнем с создания нового проекта WPF Application по имени `CustomDepPropApp`. Выберем в меню Project (Проект) пункт Add New Item (Добавить новый элемент) и, указав User Control (WPF) (Пользовательский элемент управления (WPF)) для типа элемента, создадим элемент с именем `ShowNumberControl.xaml` (рис. 27.41).

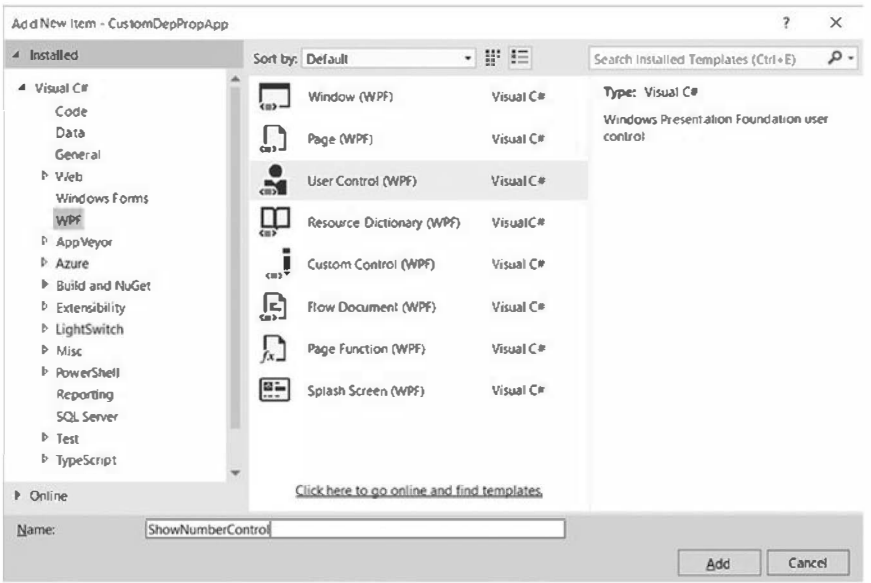


Рис. 27.41. Вставка нового специального элемента управления `UserControl`

На заметку! Более подробные сведения о классе `UserControl` в WPF будут даны в главе 29, а пока просто следуйте указаниям при проработке примера.

Подобно окну типы `UserControl` в WPF имеют файл XAML и связанный файл кода. Модифицируем разметку XAML пользовательского элемента управления, чтобы определить простой элемент `Label` внутри `Grid`:

```
<UserControl x:Class="CustomDepPropApp.ShowNumberControl"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d=http://schemas.microsoft.com/expression/blend/2008
  xmlns:local="clr-namespace:CustomDepPropApp"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="300">
  <Grid>
  <Label x:Name="numberDisplay" Height="50" Width="200" Background="LightBlue"/>
  </Grid>
</UserControl>
```

В файле кода для этого элемента создадим обычное свойство .NET, которое упаковывает поле типа `int` и устанавливает новое значение для свойства `Content` элемента `Label`:

```
public partial class ShowNumberControl : UserControl
{
  public ShowNumberControl()
  {
    InitializeComponent();
  }
  // Обычное свойство .NET.
  private int _currNumber = 0;
  public int CurrentNumber
  {
    get { return _currNumber; }
    set
    {
      _currNumber = value;
      numberDisplay.Content = CurrentNumber.ToString();
    }
  }
}
```

Теперь добавим в определение XAML окна объявление экземпляра специального элемента управления внутри диспетчера компоновки `StackPanel`. Поскольку специальный элемент управления не входит в состав основных сборок WPF, понадобится определить специальное пространство имен XML, которое отображается на этот элемент. Вот требуемая разметка:

```
<Window x:Class="CustomDepPropApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:myCtrls="clr-namespace:CustomDepPropApp"
    xmlns:local="clr-namespace:CustomDepPropApp"
    mc:Ignorable="d"
    Title="Simple Dependency Property App" Height="150" Width="250"
    WindowStartupLocation="CenterScreen">
  <StackPanel>
    <myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100"/>
  </StackPanel>
</Window>
```

Как видите, визуальный конструктор Visual Studio, похоже, корректно отображает значение, установленное в свойстве `CurrentNumber` (рис. 27.42).

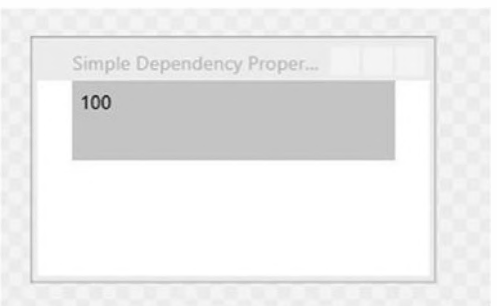


Рис. 27.42. Свойство выглядит работающим так, как ожидалось

Однако что, если к свойству CurrentNumber необходимо применить объект анимации, который обеспечит изменение значения свойства от 100 до 200 в течение 10 секунд? Если это желательно сделать в разметке, то область <myCtrls:ShowNumberControl> можно изменить следующим образом:

```
<myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100">
  <myCtrls:ShowNumberControl.Triggers>
    <EventTrigger RoutedEvent = "myCtrls:ShowNumberControl.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "CurrentNumber">
            <Int32Animation From = "100" To = "200" Duration = "0:0:10"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </myCtrls:ShowNumberControl.Triggers>
</myCtrls:ShowNumberControl>
```

После запуска приложения объект анимации не сможет найти подходящую цель и потому будет проигнорирован. Причина в том, что свойство CurrentNumber не было зарегистрировано как свойство зависимости! Чтобы устранить проблему, возвратимся в файл кода для специального элемента управления и полностью закомментируем текущую логику свойства (включая закрытое поддерживающее поле). Теперь поместим курсор внутрь области определения класса и введем фрагмент кода propdp. После ввода propdp два раза нажмем клавишу <Tab>. Фрагмент кода развернется в базовый шаблон свойства зависимости:

```
public int MyProperty
{
  get { return (int)GetValue(MyPropertyProperty); }
  set { SetValue(MyPropertyProperty, value); }
}

// Использовать DependencyProperty в качестве поддерживающего хранилища
// для MyProperty.
// Это сделает возможной анимацию, применение стилей, привязку и т.д.
public static readonly DependencyProperty MyPropertyProperty =
  DependencyProperty.Register("MyProperty", typeof(int),
typeof(ownerclass), new PropertyMetadata(0));
```

Модифицируем шаблон, как показано ниже:

```
public partial class ShowNumberControl : UserControl
{
  public int CurrentNumber
  {
    get { return (int)GetValue(CurrentNumberProperty); }
    set { SetValue(CurrentNumberProperty, value); }
  }
  public static readonly DependencyProperty CurrentNumberProperty =
    DependencyProperty.Register("CurrentNumber",
    typeof(int),
    typeof(ShowNumberControl),
    new UIPropertyMetadata(0));
  ...
}
```

Это похоже на то, что делалось в реализации свойства Height; тем не менее, фрагмент кода propdp регистрирует свойство непосредственно в теле, а не в статическом конструкторе (что хорошо). Также обратите внимание, что объект UIPropertyMetadata используется для определения стандартного целочисленного значения (0) вместо более сложного объекта FrameworkPropertyMetadata. В итоге получается простейшая версия CurrentNumber как свойства зависимости.

Добавление процедуры проверки достоверности данных

Несмотря на наличие свойства зависимости по имени CurrentNumber, анимация все еще не наблюдается. Следующей корректировкой будет указание функции, вызываемой для выполнения проверки достоверности данных. В этом примере мы предположим, что нужно обеспечить нахождение значения свойства CurrentNumber в диапазоне между 0 и 500.

Добавим в метод DependencyProperty.Register() финальный аргумент типа ValidateValueCallback, указывающий на метод по имени ValidateCurrentNumber.

Здесь ValidateValueCallback является делегатом, который может указывать только на методы, возвращающие тип bool и принимающие единственный аргумент типа object. Экземпляр object представляет присваиваемое новое значение. Реализация ValidateCurrentNumber должна возвращать true, если входное значение находится в ожидаемом диапазоне, и false — в противном случае:

```
public static readonly DependencyProperty CurrentNumberProperty =
  DependencyProperty.Register("CurrentNumber", typeof(int),
    typeof(ShowNumberControl), new UIPropertyMetadata(100),
   new ValidateValueCallback(ValidateCurrentNumber));
public static bool ValidateCurrentNumber(object value)
{
  // Очень простое бизнес-правило: значение должно находиться
  // в диапазоне между 0 и 500.
  if (Convert.ToInt32(value) >= 0 && Convert.ToInt32(value) <= 500)
    return true;
  else
    return false;
}
```

Реагирование на изменение свойства

Итак, допустимое число уже есть, но анимация по-прежнему отсутствует. Последнее изменение, которое потребуется внести — передать во втором аргументе конструктора UIPropertyMrtadata объект PropertyChangedCallback. Данный делегат может указывать на любой метод, принимающий DependencyObject в первом параметре и Dependency PropertyChangeEventArgs — во втором. Модифицируем код следующим образом:

```
// Обратите внимание на второй параметр конструктора UIPropertyMetadata.
public static readonly DependencyProperty CurrentNumberProperty =
  DependencyProperty.Register("CurrentNumber", typeof(int),
   typeof(ShowNumberControl), new UIPropertyMetadata(100,
         new PropertyChangedCallback(CurrentNumberChanged)),
         new ValidateValueCallback(ValidateCurrentNumber));
```

Конечной целью внутри метода CurrentNumberChamged() будет изменение свойства Content объекта Label на новое значение, присвоенное свойству CurrentNumber. Однако здесь мы сталкиваемся с серьезной проблемой: метод CurrentNumberChanged() является статическим, т.к. он должен работать со статическим объектом DependencyProperty. Каким же образом тогда получить доступ к объекту Label для текущего экземпляра ShowNumberControl? Нужная ссылка содержится в первом параметре DependencyObject.

Новое значение можно найти с применением входных аргументов события. Ниже показан необходимый код, который будет изменять свойство Content объекта Label:

```
private static void CurrentNumberChanged(DependencyObject depObj,
   DependencyPropertyChangedEventArgs args)
{
  // Привести DependencyObject к ShowNumberControl.
  ShowNumberControl c = (ShowNumberControl)depObj;
  // Получить элемент управления Label в ShowNumberControl.
  Label theLabel = c.numberDisplay;
  // Установить для Label новое значение.
  theLabel.Content = args.NewValue.ToString();
}
```

Видите, насколько долгий путь пришлось пройти, чтобы всего лишь изменить содержимое метки! Преимущество заключается в том, что теперь свойство зависимости CurrentNumber может быть целью для стиля WPF, объекта анимации, операции привязки данных и т.д. Снова запустив приложение, легко заметить, что значение изменяется во время выполнения.

На этом обзор свойств зависимости WPF завершен. Хотя теперь вы должны гораздо лучше понимать, что они позволяют делать, и как создавать собственные такие свойства, имейте в виду, что многие детали здесь не были раскрыты. Если вам однажды понадобится создавать множество собственных элементов управления, поддерживающих специальные свойства, то загляните в подраздел "Properties" ("Свойства") узла "WPF Fundamentals" ("Основы WPF") в документации .NET Framework 4.6 SDK. Там вы найдете намного больше примеров построения свойств зависимости, присоединяемых свойств, разнообразных способов конфигурирования метаданных и массу других подробных сведений.

Исходный код. Проект CustomDepPropApp доступен в подкаталоге Chapter_27.

Резюме

В этой главе рассматривались некоторые аспекты, связанные с элементами управления WPF, начиная с обзора набора инструментов для элементов управления и роли диспетчеров компоновки (панелей). Первый пример был посвящен построению простого приложения текстового процессора. В нем демонстрировалось использование интегрированной в WPF функциональности проверки правописания, а также создание главного окна с системой меню, строкой состояния и панелью инструментов.

Более важно то, что вы узнали, каким образом строить команды WPF. Эти независимые от элемента управления события можно присоединять к элементу пользовательского интерфейса или входному жесту для автоматического наследования готовой функциональности (например, операций с буфером обмена).

Кроме того, вы получили дополнительные сведения о применении встроенных визуальных конструкторов Visual Studio для построения пользовательских интерфейсов. В частности, с их помощью был создан сложный пользовательский интерфейс, а попутно вы ознакомились с интерфейсами Ink API и Documents API, доступными в WPF. Вы также получили представление об операциях привязки данных WPF, включая использование класса DataGrid из WPF для отображения информации из специальной базы данных AutoLot.

Наконец, вы выяснили, что инфраструктура WPF добавляет уникальный аспект к традиционным программным примитивам .NET, особенно к свойствам и событиям. Как было показано, механизм свойств зависимости позволяет строить свойство, которое может интегрироваться с набором служб WPF (анимации, привязки данных, стилей и т.д.). В качестве связанного замечания: механизм маршрутизируемых событий предоставляет событию способ распространяться вверх или вниз по дереву разметки.

ГЛАВА 28

Службы графической визуализации WPF

В этой главе мы рассмотрим возможности графической визуализации WPF. Вы увидите, что инфраструктура WPF предоставляет три отдельных способа визуализации графических данных: фигуры, рисунки и визуальные объекты. Разобравшись в преимуществах и недостатках каждого подхода, мы приступим к исследованию мира интерактивной двухмерной графики с использованием классов из пространства имен `System.Windows.Shapes`. После этого будет показано, как с помощью рисунков и геометрии визуализировать двухмерные данные в легковесной манере. И, наконец, вы узнаете, каким образом добиться от визуального уровня максимальной функциональности и производительности.

Попутно будут затронуты многие связанные темы, такие как создание специальных кистей и перьев, применение графических трансформаций к визуализации и выполнение операций проверки попадания. В частности, вы увидите, как можно упростить решение задач кодирования графики с помощью интегрированных инструментов Visual Studio и дополнительного средства под названием Inkscape.

На заметку! Графика является ключевым аспектом разработки WPF. Даже если вы не строите приложение с интенсивной графикой (вроде видеоигры или мультимедийного приложения), то темы этой главы критически важны при работе с такими службами, как шаблоны элементов управления, анимация и настройка привязки данных.

Службы графической визуализации WPF

Инфраструктура WPF использует особую разновидность графической визуализации, которая известна под названием *графика режима сохранения* (retained mode). Попросту говоря, это означает, что после применения разметки XAML или процедурного кода для генерирования графической визуализации инфраструктура WPF несет ответственность за сохранение таких визуальных элементов и обеспечение их корректной перерисовки и обновления в оптимальной манере. Таким образом, визуализируемые графические данные присутствуют постоянно, даже когда конечный пользователь скрывает изображение, изменяя размер окна или сворачивая его, перекрывая одно окно другим и т.д.

По разительному контрасту предшествующие версии API-интерфейсов графической визуализации от Microsoft (включая GDI+ в Windows Forms) были графическими системами *прямого режима* (immediate mode). В этой модели ответственность за корректное "запоминание" и обновление визуализируемых элементов на протяжении времени жизни приложения возлагалась на программиста. Например, в приложении Windows Forms

визуализация фигуры, подобной прямоугольнику, предусматривала обработку события `Paint` (или переопределение виртуального метода `OnPaint()`), получение объекта `Graphics` для рисования прямоугольника и, что важнее всего, добавление инфраструктуры, обеспечивающей сохранение изображения в ситуации, когда пользователь изменил размеры окна (например, за счет создания переменных-членов для представления позиции прямоугольника и вызова метода `Invalidate()` во многих местах кода).

Переход от графики прямого режима к графике режима сохранения — действительно хорошее решение, т.к. программистам приходится писать и сопровождать гораздо меньший объем рутинного кода для поддержки графики. Однако это не говорит о том, что API-интерфейс графики WPF *полностью* отличается от более ранних инструментальных наборов визуализации. Например, как и GDI+, инфраструктура WPF поддерживает разнообразные типы объектов кистей и перьев, приемы проверки попадания, области отсечения, графические трансформации и т.д. Поэтому если у вас есть опыт работы с GDI+ (или GDI на языке C/C++), то вы уже имеете неплохое представление о том, как выполнять базовую визуализацию в WPF.

Варианты графической визуализации WPF

Как и с другими аспектами разработки приложений WPF, существует выбор из нескольких способов выполнения графической визуализации после того, как принято решение делать это в разметке XAML или в процедурном коде C# (либо возможно с помощью их комбинации). В частности, инфраструктура WPF предлагает следующие три индивидуальных подхода к визуализации графических данных.

- *Фигуры.* Инфраструктура WPF предоставляет пространство имен `System.Windows.Shapes`, в котором определено небольшое количество классов для визуализации двухмерных геометрических объектов (прямоугольников, эллипсов, многоугольников и т.п.). Хотя эти типы очень просты в использовании и очень мощные, в случае непродуманного применения они могут привести к значительным накладным расходам памяти.
- *Рисунки и геометрии.* Второй способ визуализации графических данных в WPF предполагает работу с классами, производными от абстрактного класса `System.Windows.Media.Drawing`. Используя такие классы, как `GeometryDrawing` или `ImageDrawing` (в дополнение к различным *геометрическим объектам*), можно визуализировать графические данные в более легковесной (но менее функциональной) манере.
- *Визуальные объекты.* Самый быстрый и легковесный способ визуализации графических данных в WPF предусматривает работу с визуальным уровнем, который доступен только через код C#. С применением классов, производных от `System.Windows.Media.Visual` можно взаимодействовать непосредственно с графической подсистемой WPF.

Причина предоставления разных способов решения той же самой задачи (т.е. визуализации графических данных) связана с расходом памяти и в конечном итоге с производительностью приложения. Поскольку WPF является системой, очень интенсивно использующей графику, нет ничего необычного в том, что приложению требуется визуализировать сотни или даже тысячи различных изображений на поверхности окна, и выбор реализации (фигуры, рисунки или визуальные объекты) может оказать огромное влияние.

Важно понимать, что при построении приложения WPF высока вероятность, что придется использовать все три подхода. В качестве эмпирического правила запомните: если нужен умеренный объем *интерактивных* графических данных, которыми

может манипулировать пользователь (принимающих ввод от мыши, отображающих всплывающие подсказки и т.д.), то следует применять члены из пространства имен `System.Windows.Shapes`.

В отличие от этого рисунки и геометрии лучше подходят, когда необходимо моделировать сложные, большей частью не интерактивные векторные графические данные с использованием разметки XAML или кода C#. Хотя рисунки и геометрии способны реагировать на события мыши, а также поддерживают проверку попадания и операции перетаскивания, для выполнения таких действий обычно приходится писать больше кода.

Наконец, если требуется самый быстрый способ визуализации значительных объемов графических данных, то должен быть выбран визуальный уровень. Например, предположим, что инфраструктура WPF применяется для построения научного приложения, которое должно отображать тысячи точек на графике данных. За счет использования визуального уровня эти точки графика можно визуализировать наиболее оптимальным способом. Как будет показано далее в главе, визуальный уровень доступен только из кода C#, но не из разметки XAML.

Независимо от выбранного подхода (фигуры, рисунки и геометрии или визуальные объекты), всегда будут применяться общепринятые графические примитивы, такие как кисти (для заполнения ограниченных областей), перья (для рисования контуров) и объекты трансформации (которые видоизменяют данные). Начнем изучение с классов из пространства имен `System.Windows.Shapes`.

На заметку! Инфраструктура WPF поставляется также с полнофункциональным API-интерфейсом, который можно использовать для визуализации и манипулирования трехмерной графикой, но в книге он не рассматривается. В случае заинтересованности во внедрении трехмерной графики в свои приложения вам следует обратиться в документацию .NET Framework 4.6 SDK.

Визуализация графических данных с использованием фигур

Члены пространства имен `System.Windows.Shapes` предлагают наиболее прямолинейный, интерактивный и самый затратный в отношении памяти способ визуализации двухмерного изображения. Это довольно небольшое пространство имен (расположенное в сборке `PresentationFramework.dll`) состоит всего из шести запечатанных классов, которые расширяют абстрактный базовый класс `Shape`: `Ellipse`, `Rectangle`, `Line`, `Polygon`, `Polyline` и `Path`.

Создадим новый проект WPF Application (Приложение WPF) по имени `RenderingWithShapes` и изменим заголовок главного окна в `MainWindow.xaml` на `"Fun with Shapes!"`. Отыскав в браузере объектов Visual Studio абстрактный класс `Shape` (рис. 28.1) и раскрыв все его узлы, можно заметить, что каждый производный от `Shape` класс получает значительную часть функциональности по цепочке наследования.

Некоторые из родительских классов должны быть знакомы из материала предыдущих двух глав. Вспомните, например, что в `UIElement` определены многочисленные методы для получения ввода от мыши и обработки событий перетаскивания, а в `FrameworkElement` определены члены для работы с изменением размеров, всплывающими подсказками, курсорами мыши и т.п. С учетом этой цепочки наследования имейте в виду, что при визуализации графических данных с применением классов, производных от `Shape`, объекты получаются почти такими же функциональными (с точки зрения взаимодействия с пользователем), как элементы управления WPF!

Рис. 28.1. Базовый класс `Shape` получает много функциональности от своих родительских классов

Скажем, для выяснения, щелкнул ли пользователь на визуализированном изображении, достаточно лишь обработать событие `MouseDown`. Например, если написать следующую разметку XAML для объекта `Rectangle` внутри элемента управления `Grid` начального окна `Window`:

```
<Rectangle x:Name="myRect" Height="30" Width="30"
           Fill="Green" MouseDown="myRect_MouseDown"/>
```

то можно реализовать обработчик события `MouseDown`, который изменяет цвет фона прямоугольника в результате щелчка на нем:

```
private void myRect_MouseDown(object sender, MouseButtonEventArgs e)
{
  // Изменить цвет прямоугольника в результате щелчка на нем.
  myRect.Fill = Brushes.Pink;
}
```

В отличие от других инструментальных наборов, предназначенных для работы с графикой, вам *не* придется писать громоздкий код инфраструктуры, в котором вручную сопоставляются координаты мыши с геометрией, выясняется попадание курсора внутрь границ, выполняется визуализация в неотображаемый буфер и т.д. Члены пространства имен `System.Windows.Shapes` просто реагируют на зарегистрированные вами события подобно типичному элементу управления WPF (`Button` и т.д.).

Недостаток всей этой готовой функциональности связан с тем, что фигуры потребляют довольно много памяти. При построении научного приложения, которое рисует тысячи точек на экране, использование фигур будет неудачным выбором (по существу таким же расточительным по памяти, как визуализация тысяч объектов `Button`). Тем не менее, когда нужно сгенерировать интерактивное двухмерное векторное изображение, фигуры оказываются прекрасным выбором.

Помимо функциональности, унаследованной от родительских классов `UIElement` и `FrameworkElement`, в классе `Shape` определено множество собственных членов, наиболее полезные из которых кратко описаны в табл. 28.1.

Таблица 28.1. Ключевые свойства базового класса `Shape`

Свойства	Описание
`DefiningGeometry`	Возвращает объект `Geometry`, который представляет общие размеры текущей фигуры. Этот объект содержит *только* точки, применяемые для визуализации данных, и не имеет никаких следов функциональности из класса `UIElement` или `FrameworkElement`
`Fill`	Позволяет указать "объект кисти" для визуализации внутренней области фигуры
`GeometryTransform`	Позволяет применять трансформацию к фигуре *до* ее визуализации на экране. Унаследованное (от `UIElement`) свойство `RenderTransform` применяет трансформацию *после* визуализации фигуры на экране
`Stretch`	Описывает, каким образом расположить фигуру в выделенном ей пространстве, например, по ее позиции внутри диспетчера компоновки. Это управляется с использованием соответствующего перечисления `System.Windows.Media.Stretch`
`Stroke`	Определяет объект кисти или в ряде случаев объект пера (который на самом деле является замаскированной кистью), применяемый для рисования границы фигуры
`StrokeDashArray`, `StrokeEndLineCap`, `StrokeStartLineCap`, `StrokeThickness`	Эти (и другие) свойства, связанные со штрихами, управляют тем, как сконфигурированы линии при рисовании границ фигуры. В большинстве случаев данные свойства будут конфигурировать кисть, используемую для рисования границы или линии

На заметку! Если вы забудете установить свойства `Fill` и `Stroke`, то WPF предоставит "невидимые" кисти, вследствие чего фигура не будет видна на экране!

Добавление прямоугольников, эллипсов и линий на поверхность `Canvas`

Позже в главе вы научитесь применять инструмент Expression Design для генерации описаний XAML графических данных. А пока мы построим приложение WPF, которое может визуализировать фигуры, используя XAML и C#, и одновременно ознакомимся с процессом проверки попадания. Для начала удалим текущее описание `Rectangle` и логику обработчика событий C#. Затем модифицируем первоначальную разметку XAML для `<Window>`, определив элемент `<DockPanel>`, который содержит (пока пустые) элементы `<ToolBar>` и `<Canvas>`. Обратите внимание, что каждому содержащемуся элементу посредством свойства `Name` назначается подходящее имя:

```
<DockPanel LastChildFill="True">
  <ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
  </ToolBar>
  <Canvas Background="LightBlue" Name="canvasDrawingArea"/>
</DockPanel>
```

Теперь наполним элемент `<ToolBar>` набором объектов `<RadioButton>`, каждый из которых содержит объект специфического класса, производного от `Shape`. Легко заме-

тить, что каждому элементу <RadioButton> назначается то же самое групповое имя GroupName (чтобы обеспечить взаимное исключение) и также подходящее индивидуальное имя:

```
<ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
  <RadioButton Name="circleOption" GroupName="shapeSelection">
    <Ellipse Fill="Green" Height="35" Width="35" />
  </RadioButton>
  <RadioButton Name="rectOption" GroupName="shapeSelection">
    <Rectangle Fill="Red" Height="35"
               Width="35" RadiusY="10" RadiusX="10" />
  </RadioButton>
  <RadioButton Name="lineOption" GroupName="shapeSelection">
    <Line Height="35" Width="35"
          StrokeThickness="10" Stroke="Blue"
          X1="10" Y1="10" Y2="25" X2="25"
          StrokeStartLineCap="Triangle" StrokeEndLineCap="Round" />
  </RadioButton>
</ToolBar>
```

Как видите, объявление объектов Rectangle, Ellipse и Line в XAML довольно прямолинейно и требует лишь минимальных комментариев. Вспомните, что свойство Fill позволяет указать *кисть* для рисования внутренностей фигуры. Когда нужна кисть сплошного цвета, можно просто задать жестко закодированную строку известных значений, а соответствующий преобразователь типа сгенерирует корректный объект. Интересная характеристика типа Rectangle связана с тем, что в нем определены свойства RadiusX и RadiusY, позволяющие визуализировать скругленные углы.

Объект Line представлен своими начальной и конечной точками с применением свойств X1, X2, Y1 и Y2 (учитывая, что *высота* и *ширина* при описании линии имеют мало смысла). Здесь устанавливается несколько дополнительных свойств, которые управляют тем, как визуализируются начальная и конечная точки объекта Line, а также настраивают параметры штриха. На рис. 28.2 показана визуализированная панель инструментов в визуальном конструкторе WPF среды Visual Studio.

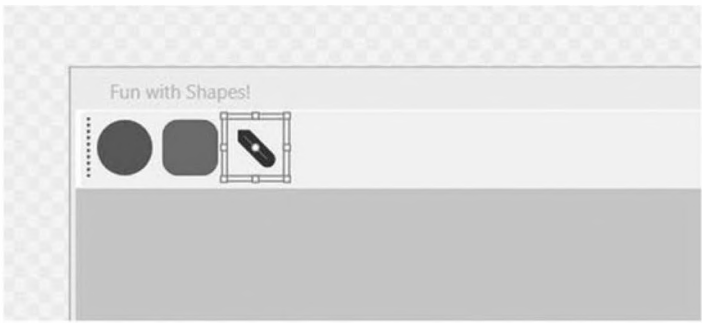


Рис. 28.2. Использование объектов Shape в качестве содержимого для набора элементов RadioButton

С применением окна Properties (Свойства) среды Visual Studio создадим обработчик события MouseLeftButtonDown для Canvas и обработчик события Click для каждого элемента RadioButton. Цель заключается в том, чтобы в коде C# визуализировать выбранную фигуру (круг, квадрат или линию), когда пользователь щелкает внутри Canvas. Первым делом определим следующее вложенное перечисление (и соответствующую переменную-член) внутри класса, производного от Window:

```
public partial class MainWindow : Window
{
  private enum SelectedShape
  { Circle, Rectangle, Line }
  private SelectedShape _currentShape;
  ...
}
```

В каждом обработчике `Click` **установим переменную-член** `currentShape` **в корректное значение** `SelectedShape`. **Например, ниже показан код обработчика события** `Click` **элемента** `RadioButton` **по имени** `circleOption`. **Остальные два обработчика** `Click` **реализованы в аналогичной манере.**

```
private void circleOption_Click(object sender, RoutedEventArgs e)
{
  _currentShape = SelectedShape.Circle;
}
private void rectOption_Click(object sender, RoutedEventArgs e)
{
  _currentShape = SelectedShape.Rectangle;
}
private void lineOption_Click(object sender, RoutedEventArgs e)
{
  _currentShape = SelectedShape.Line;
}
```

С помощью обработчика события `MouseLeftButtonDown` **элемента** `Canvas` **будет визуализироваться корректная фигура (предопределенного размера) в начальной точке, соответствующей позиции X,Y курсора мыши. Ниже приведена полная реализация с последующим анализом:**

```
private void canvasDrawingArea_MouseLeftButtonDown(object sender,
                                                   MouseButtonEventArgs e)
{
  Shape shapeToRender = null;
  // Сконфигурировать корректную фигуру для рисования.
  switch (_currentShape)
  {
    case SelectedShape.Circle:
     shapeToRender = new Ellipse() { Fill = Brushes.Green, Height = 35, Width = 35 };
     break;
    case SelectedShape.Rectangle:
      shapeToRender = new Rectangle()
       { Fill = Brushes.Red, Height = 35, Width = 35, RadiusX = 10, RadiusY = 10 };
      break;
    case SelectedShape.Line:
      shapeToRender = new Line()
      {
        Stroke = Brushes.Blue,
        StrokeThickness = 10,
        X1 = 0, X2 = 50, Y1 = 0, Y2 = 50,
        StrokeStartLineCap= PenLineCap.Triangle,
        StrokeEndLineCap = PenLineCap.Round
      };
      break;
    default:
      return;
  }
```

```
    // Установить верхний левый угол для рисования на холсте.
    Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
    Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);

    // Нарисовать фигуру.
    canvasDrawingArea.Children.Add(shapeToRender);
}
```

На заметку! Вы могли заметить, что объекты `Ellipse`, `Rectangle` и `Line`, создаваемые в этом методе, имеют те же настройки свойств, что и соответствующие определения XAML. Код вполне ожидаемо можно упростить, но это требует понимания объектных ресурсов WPF, которые будут рассматриваться в главе 29.

Как видите, в коде производится проверка переменной-члена `currentShape` для создания корректного объекта, производного от `Shape`. Затем устанавливаются координаты левого верхнего угла внутри `Canvas` с использованием входного объекта `MouseButtonEventArgs`. И, наконец, в коллекцию объектов `UIElement`, поддерживаемую `Canvas`, добавляется новый производный от `Shape` объект. Если запустить программу прямо сейчас, то она должна позволить щелкать левой кнопкой мыши где угодно на холсте и визуализировать в позиции щелчка выбранную фигуру.

Удаление прямоугольников, эллипсов и линий с поверхности `Canvas`

Теперь, имея элемент `Canvas` с коллекцией объектов, может возникнуть вопрос: как динамически удалить элемент, скажем, в ответ на щелчок пользователя правой кнопкой мыши на фигуре? Это делается с помощью класса `VisualTreeHelper` из пространства имен `System.Windows.Media`. Роль "визуальных деревьев" и "логических деревьев" более подробно объясняется в главе 29, а пока обработаем событие `MouseRightButtonDown` объекта `Canvas` и реализуем соответствующий обработчик:

```
private void canvasDrawingArea_MouseRightButtonDown(object sender,
                                                    MouseButtonEventArgs e)
{
  // Сначала получить координаты X,Y позиции, где пользователь выполнил щелчок.
  Point pt = e.GetPosition((Canvas)sender);

  // Использовать метод HitTest() класса VisualTreeHelper, чтобы
  // выяснить, щелкнул ли пользователь на элементе внутри Canvas.
  HitTestResult result = VisualTreeHelper.HitTest(canvasDrawingArea, pt);

  // Если переменная result не равна null, то щелчок произведен на фигуре.
  if (result != null)
  {
    // Получить фигуру, на которой совершен щелчок, и удалить ее из Canvas.
    canvasDrawingArea.Children.Remove(result.VisualHit as Shape);
  }
}
```

Код обработчика начинается с получения точных координат X,Y позиции, где пользователь щелкнул внутри `Canvas`, и проверки попадания посредством статического метода `VisualTreeHelper.HitTest()`. Возвращаемым значением является объект `HitTestResult`, который будет установлен в `null`, если пользователь выполнил щелчок не на `UIElement` внутри `Canvas`. Если значение `HitTestResult` *не* равно `null`, то с помощью свойства `VisualHit` можно получить элемент `UIElement`, на котором был совершен щелчок, и привести его к типу, производному от `Shape` (вспомните, что `Canvas` может

содержать любой UIElement, а не только фигуры). Детали, связанные с "визуальным деревом", будут изложены в следующей главе.

На заметку! По умолчанию метод VisualTreeHelper.HitTest() возвращает объект UIElement самого верхнего уровня, на котором совершен щелчок, и не предоставляет информацию о других объектах, расположенных под ним (т.е. перекрытых в Z-порядке).

В результате внесенных модификаций должна появиться возможность добавления фигуры на Canvas щелчком левой кнопки мыши и ее удаления щелчком правой кнопки мыши. На рис. 28.3 демонстрируется функциональность текущего примера.

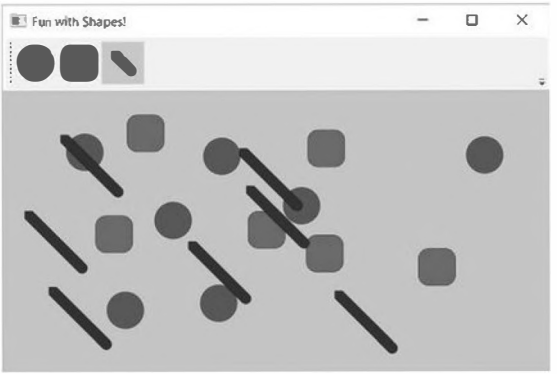


Рис. 28.3. Работа с фигурами

К настоящему моменту мы применяли объекты типов, производных от Shape, для визуализации содержимого элементов RadioButton с использованием разметки XAML и заполняли Canvas в коде C#. Во время исследования роли кистей и графических трансформаций к этому примеру будет добавлена дополнительная функциональность. К слову, в другом примере главы будут иллюстрироваться приемы перетаскивания на объектах UIElement. А пока давайте рассмотрим оставшиеся члены пространства имен System.Windows.Shapes.

Работа с элементами Polyline и Polygon

В текущем примере используются только три класса, производных от Shape. Остальные дочерние классы (Polyline, Polygon и Path) чрезвычайно утомительно корректно визуализировать без инструментальной поддержки (такой как Expression Blend или другие инструменты, которые могут создавать векторную графику) — просто потому, что они требуют определения большого количества точек для своего выходного представления. Вскоре вы узнаете о роли инструмента Expression Design, а пока займемся кратким обзором оставшихся типов Shapes.

Тип Polyline позволяет определить коллекцию координат X,Y (через свойство Points) для рисования последовательности линейных сегментов, не требующих замыкания. Тип Polygon похож, но запрограммирован так, что всегда замыкает контур, соединяя начальную точку с конечной, и заполняет внутреннюю область с помощью указанной кисти. Предположим, что в редакторе Kaxaml или, еще лучше, в специальном редакторе XAML, построенном в главе 26, создан следующий элемент <StackPanel>:

```
<!-- Элемент Polyline не замыкает автоматически конечные точки -->
<Polyline Stroke ="Red" StrokeThickness ="20" StrokeLineJoin ="Round"
          Points ="10,10 40,40 10,90 300,50"/>
```

```
<!-- Элемент Polygon всегда замыкает конечные точки -->
<Polygon Fill ="AliceBlue" StrokeThickness ="5" Stroke ="Green"
         Points ="40,10 70,80 10,50" />
```

На рис. 28.4 показывает визуализированный вывод в приложении MyXamlPad.

Работа с элементом Path

Применяя только типы Rectangle, Ellipse, Polygon, Polyline и Line, нарисовать детализированное двухмерное векторное изображение было бы исключительно трудно, т.к. эти примитивы не позволяют легко фиксировать графические данные, подобные кривым, объединениям перекрывающихся данных и т.д. Последний производный от Shape класс, Path, предоставляет возможность определения сложных двухмерных графических данных в виде коллекции независимых *геометрий*. После того, как коллекция таких геометрий определена, ее можно присвоить свойству Data класса Path, где эта информация будет использоваться для визуализации сложного двухмерного изображения.

Свойство Data получает объект производного от System.Windows.Media.Geometry класса, который содержит ключевые члены, кратко описанные в табл. 28.2.

Таблица 28.2. Избранные члены класса System.Windows.Media.Geometry

Член	Описание
Bounds	Устанавливает текущий ограничивающий прямоугольник, который содержит геометрию
FillContains()	Определяет, находится ли заданный объект Point (или другой объект Geometry) внутри границ отдельного класса, производного от Geometry. Это полезно при вычислениях для проверки попадания
GetArea()	Возвращает общую область, занятую объектом производного от Geometry типа
GetRenderBounds()	Возвращает объект Rect, содержащий наименьший возможный прямоугольник, который может быть применен для визуализации объекта класса, производного от Geometry
Transform	Назначает геометрии объект Transform для изменения визуализации

Классы, которые расширяют класс Geometry (табл. 28.3), выглядят очень похожими на свои аналоги, производные от Shape. Например, класс EllipseGeometry имеет члены, подобные членам класса Ellipse. Крупное отличие связано с тем, что производным от Geometry классам *не известно*, каким образом визуализировать себя напрямую, поскольку они не являются UIElement. Взамен классы, производные от Geometry, представляют всего лишь коллекцию данных о точках, которая указывает объекту Path, как их визуализировать.

Таблица 28.3. Классы, производные от Geometry

Класс	Описание
LineGeometry	Представляет прямую линию
RectangleGeometry	Представляет прямоугольник
EllipseGeometry	Представляет эллипс
GeometryGroup	Позволяет группировать вместе несколько объектов Geometry
CombinedGeometry	Позволяет объединять два разных объекта Geometry в единую фигуру
PathGeometry	Представляет фигуру, образованную из линий и кривых

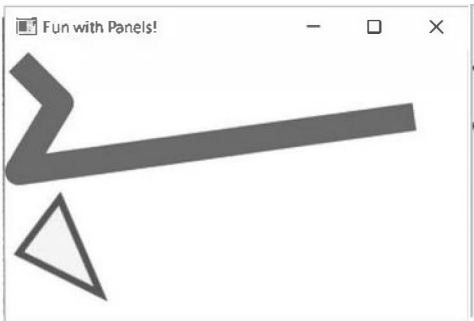


Рис. 28.4. Элементы `Polyline` и `Polygon`

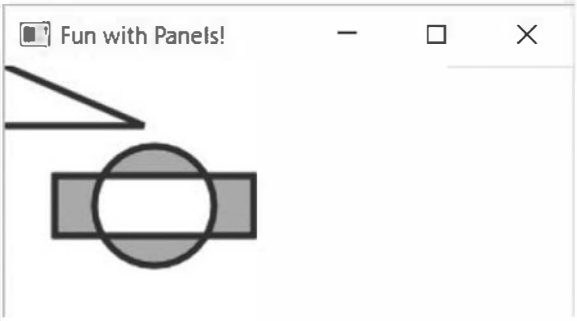


Рис. 28.5. Элемент `Path`, содержащий разнообразные объекты `Geometry`

На заметку! В инфраструктуре WPF класс `Path` не является единственным классом, который может использовать коллекцию геометрий. Например, `DoubleAnimationUsingPath`, `DrawingGroup`, `GeometryDrawing` и даже `UIElement` могут применять геометрии для визуализации через свойства `PathGeometry`, `ClipGeometry`, `Geometry` и `Clip` соответственно.

В показанной далее разметке для элемента `Path` используется несколько производных от `Geometry` типов. Обратите внимание, что свойство `Data` объекта `Path` устанавливается в объект `GeometryGroup`, который содержит объекты других производных от `Geometry` классов, таких как `EllipseGeometry`, `RectangleGeometry` и `LineGeometry`. Результат представлен на рис. 28.5.

```
<!-- Элемент Path содержит набор объектов
     Geometry, установленный в свойстве Data -->
<Path Fill = "Orange" Stroke = "Blue" StrokeThickness = "3">
  <Path.Data>
    <GeometryGroup>
      <EllipseGeometry Center = "75,70"
                       RadiusX = "30" RadiusY = "30" />
    <RectangleGeometry Rect = "25,55 100 30" />
    <LineGeometry StartPoint="0,0" EndPoint="70,30" />
    <LineGeometry StartPoint="70,30" EndPoint="0,30" />
  </GeometryGroup>
  </Path.Data>
</Path>
```

Изображение на рис. 28.5 может быть визуализировано с применением показанных ранее классов `Line`, `Ellipse` и `Rectangle`. Однако это потребовало бы помещения различных объектов `UIElement` в память. Когда для моделирования точек того, что нужно нарисовать, используются геометрии, а затем коллекция геометрий помещается в контейнер, который способен визуализировать данные (в этом случае `Path`), то тем самым сокращается расход памяти.

Теперь вспомните, что класс `Path` имеет ту же цепочку наследования, что и любой член пространства имен `System.Windows.Shapes`, поэтому он обладает возможностью отправлять такие же уведомления о событиях, как и другие элементы `UIElement`. Следовательно, если определить тот же самый элемент `<Path>` в проекте Visual Studio, то выяснить, что пользователь щелкнул в любом месте линии, можно будет за счет обработки события мыши (не забывайте, что редактор Kaxaml не разрешает обрабатывать события для написанной разметки).

"Мини-язык" моделирования путей

Из всех классов, перечисленных в табл. 28.2, класс PathGeometry наиболее сложен для конфигурирования в терминах XAML и кода. Это связано с тем фактом, что каждый *сегмент* PathGeometry состоит из объектов, содержащих разнообразные сегменты и фигуры (к примеру, ArcSegment, BezierSegment, LineSegment, PolyBezierSegment, PolyLineSegment, PolyQuadraticBezierSegment и т.д.). Вот пример объекта Path, свойство Data которого было установлено в элемент <PathGeometry>, состоящий из различных фигур и сегментов:

```
<Path Stroke="Black" StrokeThickness="1">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="10,50">
          <PathFigure.Segments>
          <BezierSegment
            Point1="100,0"
            Point2="200,200"
            Point3="300,100"/>
          <LineSegment Point="400,100" />
          <ArcSegment
            Size="50,50" RotationAngle="45"
            IsLargeArc="True" SweepDirection="Clockwise"
            Point="200,100"/>
          </PathFigure.Segments>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

По правде говоря, лишь очень немногим программистам придется когда-либо вручную строить сложные двухмерные изображения, напрямую описывая объекты классов, производных от Geometry или PathSegment. Позже в этой главе вы узнаете, как преобразовывать векторную графику в операторы "мини-языка" моделирования путей, которые можно применять в разметке XAML.

Даже с учетом содействия со стороны упомянутых ранее инструментов объем разметки XAML, требуемой для определения сложных объектов Path, может быть устрашающе большим, т.к. данные состоят из полных описаний различных объектов классов, производных от Geometry или PathSegment. Для того чтобы создавать более лаконичную разметку, в классе Path поддерживается специализированный "мини-язык".

Например, вместо установки свойства Data объекта Path в коллекцию объектов производных от Geometry и PathSegment классов его можно установить в одиночный строковый литерал, содержащий набор известных символов и различных значений, которые определяют фигуру, подлежащую визуализации. Ниже приведен простой пример, а его результирующий вывод показан на рис. 28.6:

```
<Path Stroke="Black" StrokeThickness="3"
      Data="M 10,75 C 70,15 250,270 300,175 H 240" />
```

Команда M (от *move* — переместить) принимает координаты X,Y позиции, которая представляет начальную точку рисования. Команда C (от *curve* — кривая) принимает последовательность точек для визуализации кривой (точнее кубической кривой Безье), а команда H (от *horizontal* — горизонтальная) рисует горизонтальную линию.

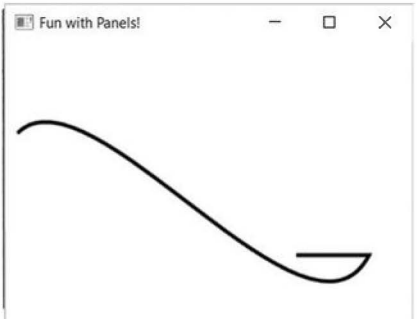


Рис. 28.6. "Мини-язык" моделирования путей позволяет компактно описывать объектную модель `Geometry/PathSegment`

И снова следует отметить, что вам придется очень редко вручную строить или анализировать строковый литерал, содержащий инструкции мини-языка моделирования путей. Тем не менее, хотя бы не будет казаться непонятной разметка XAML, генерируемая специализированными инструментами. Если вас интересуют детали этой конкретной грамматики, то обращайтесь в раздел "Path Markup Syntax" ("Синтаксис разметки путей") документации .NET Framework 4.6 SDK.

Кисти и перья WPF

Каждый способ графической визуализации (фигуры, рисование и геометрии, а также визуальные объекты) интенсивно использует *кисти*, которые позволяют управлять заполнением внутренней области двухмерной фигуры. В WPF предоставляются шесть разных типов кистей, и все они расширяют класс `System.Windows.Media.Brush`. Несмотря на то что `Brush` является абстрактным классом, его потомки, описанные в табл. 28.4, могут применяться для заполнения области содержимым почти любого мыслимого вида.

Таблица 28.4. Классы, производные от `Brush`

Класс	Описание
`DrawingBrush`	Заполняет область с помощью объекта производного от `Drawing` класса (`GeometryDrawing`, `ImageDrawing` или `VideoDrawing`)
`OmageBrush`	Заполняет область изображением (представленным посредством объекта `ImageSource`)
`LinearGradientBrush`	Заполняет область линейным градиентом
`RadialGradientBrush`	Заполняет область радиальным градиентом
`SolidColorBrush`	Заполняет область сплошным цветом, указанным в свойстве `Color`
`VisualBrush`	Заполняет область с помощью объекта производного от `Visual` класса (`DrawingVisual`, `Viewport3DVisual` и `ContentVisual`)

Классы `DrawingBrush` и `VisualBrush` позволяют строить кисть на основе существующего класса, производного от `Drawing` или `Visual`. Эти классы кистей используются при работе с двумя другими способами визуализации графики WPF (рисунками или визуальными объектами) и будут объясняться далее в главе.

Класс `ImageBrush` позволяет строить кисть, отображающую данные изображения из внешнего файла или встроенного ресурса приложения, который указан в его свойстве `ImageSource`. Оставшиеся типы кистей (`LinearGradientBrush` и `RadialGradientBrush`) довольно просты в применении, хотя требуемая разметка XAML может оказаться многословной. К счастью, в среде Visual Studio поддерживаются интегрированные редакторы кистей, которые облегчают задачу генерации стилизованных кистей.

Конфигурирование кистей с использованием Visual Studio

Давайте обновим приложение WPF для рисования, `RenderingShapes`, чтобы использовать в нем более интересные кисти.

В трех фигурах, которые были задействованы до сих пор при визуализации данных в панели инструментов, применяются простые сплошные цвета, так что их значения можно зафиксировать с помощью простых строковых литералов. Чтобы сделать задачу чуть более интересной, теперь мы будем использовать интегрированный редактор кистей. Удостоверившись, что редактор XAML начального окна открыт в IDE-среде, выберем элемент `Ellipse`. В окне Properties отыщем категорию Brush (Кисть) и щелкнем на свойстве `Fill` (рис. 28.7).

В верхней части редактора кистей находится набор свойств, которые являются "совместимыми с кистью" для выбранного элемента (т.е. `Fill`, `Stroke` и `OpacityMask`). Под ними расположен набор вкладок, которые позволяют конфигурировать разные типы кистей, включая текущую кисть со сплошным цветом. Для управления цветом текущей кисти можно применять инструмент выбора цвета, а также ползунки ARGB (alpha, red, green, blue — прозрачность, красный, зеленый, синий). С помощью этих ползунков и связанной с ними области выбора цвета можно создать сплошной цвет любого вида. Используем указанные инструменты для изменения цвета в свойстве `Fill` элемента `Ellipse` и просмотрим результирующую разметку XAML. Как видите, цвет сохранен в виде шестнадцатеричного значения:

```
<Ellipse Fill="#FF47CE47" Height="35" Width="35" />
```

Что более интересно, тот же самый редактор позволяет конфигурировать и градиентные кисти, которые применяются для определения последовательностей цветов и точек перехода цветов. Вспомните, что редактор кистей предлагает набор вкладок, первая из которых позволяет установить *null-кисть* для отсутствующего визуализированного вывода. Остальные четыре дают возможность установить кисть сплошного цвета (как только что было показано), градиентную кисть, мозаичную кисть и кисть с изображением.

Щелкнем на вкладке градиентной кисти; редактор отобразит несколько новых опций (рис. 28.8). Три кнопки в нижнем левом углу позволяют выбрать линейный градиент, радиальный градиент или обратить градиентные переходы. Полоса внизу покажет текущий цвет каждого градиентного перехода, который будет представлен специальным ползунком. По мере перетаскивания ползунка по полосе градиента можно управлять смещением градиента. Более того, щелкнув на конкретном ползунке, можно изменить цвет определенного градиентного перехода с помощью селектора цвета. Наконец, щелчок прямо на полосе градиента позволяет добавлять дополнительные градиентные переходы.

Потратьте некоторое время на освоение этого редактора. Мы построим радиальную градиентную кисть, содержащую три градиентных перехода, и установим их цвета. На рис. 28.8 показан пример кисти, использующей три разных оттенка зеленого цвета.

В результате IDE-среда обновит разметку XAML, добавив набор специальных кистей и установив их в совместимых с кистями свойствах (в рассматриваемом примере это свойство `Fill` элемента `Ellipse`) с применением синтаксиса "свойство-элемент":

```
<Ellipse Height="35" Width="35">
  <Ellipse.Fill>
    <RadialGradientBrush>
      <GradientStop Color="#FF87E71B" Offset="0.589" />
      <GradientStop Color="#FF2BA92B" Offset="0.013" />
      <GradientStop Color="#FF34B71B" Offset="1" />
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

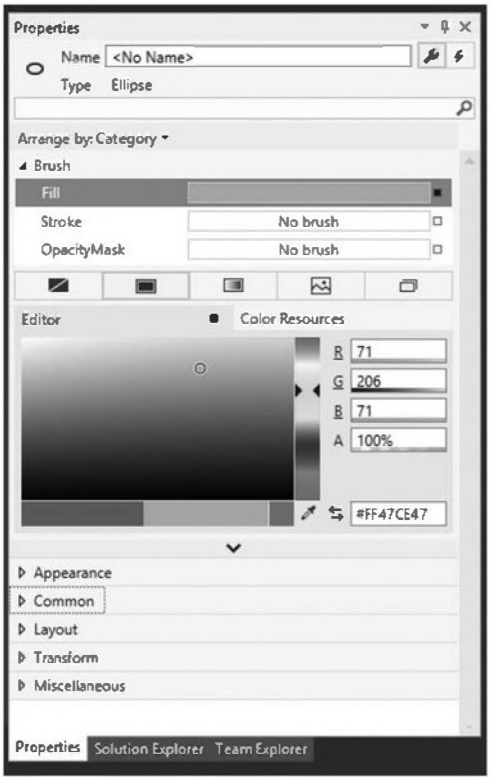


Рис. 28.7. Любое свойство, которое требует кисти, может быть сконфигурировано с помощью интегрированного редактора кистей

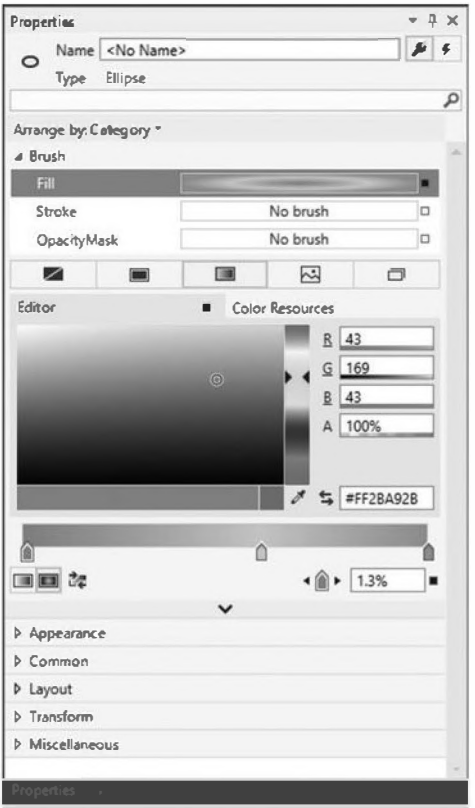


Рис. 28.8. Редактор кистей Visual Studio позволяет строить базовые градиентные кисти

Конфигурирование кистей в коде

Теперь, когда мы построили специальную кисть для определения XAML элемента `Ellipse`, соответствующий код C# устарел в том, что он по-прежнему будет визуализировать круг со сплошным зеленым цветом. Для восстановления синхронизации модифицируем нужный оператор `case`, чтобы использовать только что созданную кисть. Ниже показано необходимое обновление, которое выглядит более сложным, чем можно было ожидать, т.к. шестнадцатеричное значение преобразуется в подходящий объект `Color` посредством класса `System.Windows.Media.ColorConverter` (результат изменения представлен на рис. 28.9):

```
case SelectedShape.Circle:
  shapeToRender = new Ellipse() { Height = 35, Width = 35 };
  // Создать кисть RadialGradientBrush в коде.
  RadialGradientBrush brush = new RadialGradientBrush();
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF87E71B"), 0.589));
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF2BA92B"), 0.013));
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF34B71B"), 1));
  shapeToRender.Fill = brush;
  break;
```

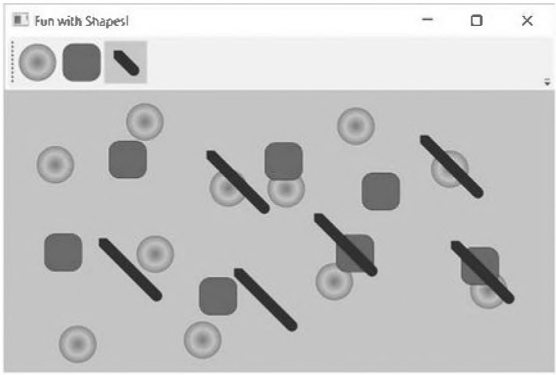


Рис. 28.9. Рисование более интересных кругов

Кстати, объекты `GradientStop` можно строить, указывая простой цвет в качестве первого параметра конструктора с применением перечисления `Colors`, которое дает сконфигурированный объект `Color`:

```
GradientStop g = new GradientStop(Colors.Aquamarine, 1);
```

Если требуется более тонкий контроль, то можно передавать объект `Color`, сконфигурированный в коде, например:

```
Color myColor = new Color() { R = 200, G = 100, B = 20, A = 40 };
GradientStop g = new GradientStop(myColor, 34);
```

Разумеется, использование перечисления `Colors` и класса `Color` не ограничивается градиентными кистями. Их можно применять всякий раз, когда необходимо представить значение цвета в коде.

Конфигурирование перьев

В сравнении с кистями *перо* представляет собой объект для рисования границ геометрий или в случае класса `Line` либо `PolyLine` — самой линейной геометрии. В частности, класс `Pen` позволяет рисовать линию указанной толщины, представленную значением типа `double`. Вдобавок объект `Pen` может быть сконфигурирован с помощью того же самого вида свойств, что и в классе `Shape`, таких как начальный и конечный концы пера, шаблоны точек-тире и т.д. Например, для определения атрибутов пера к определению фигуры можно добавить следующую разметку:

```
<Pen Thickness="10" LineJoin="Round" EndLineCap="Triangle" StartLineCap="Round" />
```

Во многих случаях создавать объект `Pen` непосредственно не придется, потому что это делается косвенно, когда присваиваются значения таким свойствам, как `StrokeThickness` производного от `Shape` типа (а также других типов `UIElement`). Однако построение специального объекта `Pen` очень удобно при работе с типами, производными от `Drawing` (которые рассматриваются позже в главе). Среда Visual Studio не располагает редактором перьев *как таковым*, но позволяет конфигурировать все свойства, связанные со штрихами, выбранного элемента с использованием окна Properties.

Применение графических трансформаций

В завершение обсуждения фигур рассмотрим тему *трансформаций*. Инфраструктура WPF поставляется с многочисленными классами, которые расширяют абстрактный базовый класс `System.Winodws.Media.Transform`. В табл. 28.5 кратко описаны основные классы, производные от `Transform`.

Таблица 28.5. Основные классы, производные от `System.Windows.Media.Transform`

Класс	Описание
`MatrixTransform`	Создает произвольную матричную трансформацию, которая используется для манипулирования объектами или координатными системами на двухмерной плоскости
`RotateTransform`	Поворачивает объект по часовой стрелке вокруг указанной точки в двухмерной системе координат (*x*, *y*)
`ScaleTransform`	Масштабирует объект в двухмерной системе координат (*x*, *y*)
`SkewTransform`	Перекашивает объект в двухмерной системе координат (*x*, *y*)
`TranslateTransform`	Преобразует (перемещает) объект в двухмерной системе координат (*x*, *y*)
`TransformGroup`	Представляет комбинированный объект `Transform`, состоящий из других объектов `Transform`

Трансформации могут применяться к любым объектам `UIElement` (например, к объектам производных от `Shape` классов, а также к элементам управления, таким как `Button`, `TextBox` и т.п.). Используя эти классы трансформаций, можно визуализировать графические данные под заданным углом, скашивать изображение на поверхности и растягивать, сжимать либо поворачивать целевой элемент разными способами.

На заметку! Хотя объекты трансформаций могут применяться повсеместно, вы сочтете их наиболее удобными при работе с анимацией WPF и специальными шаблонами элементов управления. Как будет показано далее в главе, анимацию WPF можно использовать для включения в специальный элемент управления визуальных подсказок, предназначенных конечному пользователю.

Назначать целевому объекту (`Button`, `Path` и т.д.) трансформации (либо их целый набор) можно с помощью двух общих свойств, `LayoutTransform` и `RenderTransform`.

Свойство `LayoutTransform` удобно тем, что трансформация происходит *перед* визуализацией элементов в диспетчере компоновки и потому не влияет на операции Z-упорядочивания (т.е. трансформируемые данные изображений не перекрываются).

С другой стороны, трансформация из свойства `RenderTransform` инициируется после того, как элементы попали в свои контейнеры, поэтому вполне возможно, что элементы будут трансформированы с перекрытием друг друга в зависимости от того, как они организованы в контейнере.

Первый взгляд на трансформации

Вскоре мы добавим к проекту `RenderingWithShapes` некоторую трансформирующую логику. Чтобы увидеть объект трансформации в действии, откроем редактор Kaxaml (или созданный ранее специальный редактор XML), определим простой элемент `<StackPanel>` в корневом элементе `<Page>` или `<Window>` и установим свойство `Orientation` в `Horizontal`. Далее добавим следующий элемент `<Rectangle>`, который будет нарисован под углом в 45 градусов с применением объекта `RotateTransform`:

```
<!-- Элемент Rectangle с трансформацией поворотом -->
<Rectangle Height ="100" Width ="40" Fill ="Red">
  <Rectangle.LayoutTransform>
    <RotateTransform Angle ="45"/>
  </Rectangle.LayoutTransform>
</Rectangle>
```

Здесь элемент <Button> скашивается на поверхности на 20 градусов посредством трансформации <SkewTransform>:

```
<!-- Элемент Button с трансформацией скашиванием -->
<Button Content ="Click Me!" Width="95" Height="40">
  <Button.LayoutTransform>
   <SkewTransform AngleX ="20" AngleY ="20"/>
  </Button.LayoutTransform>
</Button>
```

Для полноты картины ниже приведен элемент <Ellipse>, масштабированный на 20% с помощью трансформации ScaleTransform (обратите внимание на значения, установленные в свойствах Height и Width), а также элемент <TextBox>, к которому применена группа объектов трансформации:

```
<!-- Элемент Ellipse, масштабированный на 20% -->
<Ellipse Fill ="Blue" Width="5" Height="5">
  <Ellipse.LayoutTransform>
    <ScaleTransform ScaleX ="20" ScaleY ="20"/>
  </Ellipse.LayoutTransform>
</Ellipse>

<!-- Элемент TextBox, повернутый и скошенный -->
<TextBox Text ="Me Too!" Width="50" Height="40">
  <TextBox.LayoutTransform>
    <TransformGroup>
      <RotateTransform Angle ="45"/>
      <SkewTransform AngleX ="5" AngleY ="20"/>
    </TransformGroup>
  </TextBox.LayoutTransform>
</TextBox>
```

Следует отметить, что в случае применения трансформации выполнять какие-либо ручные вычисления для реагирования на проверку попадания, перемещение фокуса ввода и аналогичные действия не придется. Графический механизм WPF самостоятельно решает такие задачи. Например, на рис. 28.10 можно видеть, что элемент TextBox по-прежнему реагирует на клавиатурный ввод.

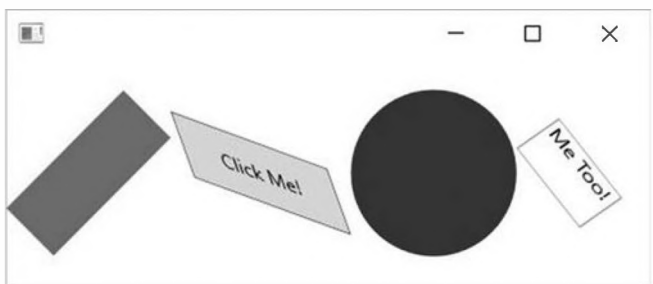


Рис. 28.10. Результат применения объектов графических трансформаций

Трансформация данных Canvas

Теперь давайте внедрим логику трансформации в пример RenderingWithShapes. В дополнение к применению объектов трансформации к одиночному элементу (Rectangle, TextBox и т.д.), их можно также применять к диспетчеру компоновки, чтобы трансформировать все внутренние данные. Например, можно было бы визуализировать всю панель <DockPanel> главного окна под углом:

```
<DockPanel LastChildFill="True">
  <DockPanel.LayoutTransform>
    <RotateTransform Angle="45"/>
  </DockPanel.LayoutTransform>
  ...
</DockPanel>
```

В рассматриваемом примере это несколько чрезмерно, так что давайте добавим последнюю (менее энергичную) возможность, которая позволит пользователю зеркально отобразить целый контейнер `Canvas` и всю содержащуюся в нем графику. Начнем с добавления в `<ToolBar>` финального элемента `<ToggleButton>` со следующим определением:

```
<ToggleButton Name="flipCanvas" Click="flipCanvas_Click" Content="Flip Canvas!"/>
```

Внутри обработчика события `Click` для нового элемента `ToggleButton` создадим объект `RotateTransform` и подключим его к объекту `Canvas` через свойство `LayoutTransform`, если элемент `ToggleButton` отмечен. Если же элемент `ToggleButton` не отмечен, то удалим трансформацию, установив свойство `LayoutTransform` в `null`.

```
private void flipCanvas_Click(object sender, RoutedEventArgs e)
{
  if (flipCanvas.IsChecked == true)
  {
    RotateTransform rotate = new RotateTransform(-180);
    canvasDrawingArea.LayoutTransform = rotate;
  }
  else
  {
    canvasDrawingArea.LayoutTransform = null;
  }
}
```

Запустив приложение, добавим несколько графических фигур в область `Canvas`. После щелчка на новой кнопке обнаружится, что фигуры выходят за границы `Canvas` (рис. 28.11). Причина в том, что не был определен прямоугольник отсечения.

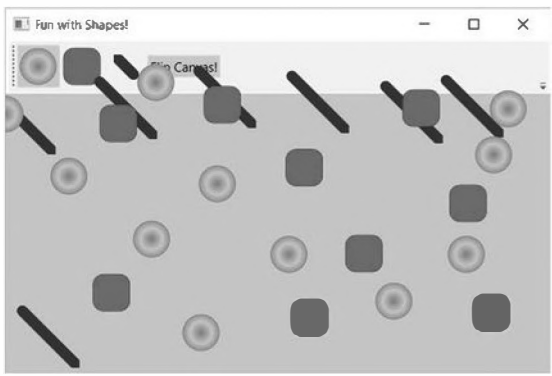


Рис. 28.11. После трансформации фигуры выходят за границы `Canvas`

Исправить проблему легко. Вместо того чтобы вручную писать сложную логику отсечения, просто установим свойство `ClipToBounds` элемента `<Canvas>` в `True`, предотвратив визуализацию дочерних элементов вне границ родительского элемента. После запуска приложения можно заметить, что графические данные больше не покидают границ отведенной области.

```
<Canvas ClipToBounds = "True" ... >
```

Последняя небольшая модификация, которую понадобится внести, связана с тем фактом, что когда пользователь зеркально отображает холст, щелкая на кнопке переключения, а затем щелкает на нем для рисования новой фигуры, то точка, где был произведен щелчок, *не* является той позицией, куда попадут графические данные. Взамен они появятся в месте нахождения курсора мыши.

Чтобы устранить эту проблему, обратимся к коду примера. Решение заключается в добавлении еще одной переменной-члена булевского типа (_isFlipped), с помощью которой обеспечивается применение того же самого объекта трансформации к рисуемой фигуре перед выполнением визуализации (через RenderTransform). Ниже показан основной фрагмент кода:

```
private bool _isFlipped = false;
private void canvasDrawingArea_MouseLeftButtonDown(object sender,
                                                   MouseButtonEventArgs e)
{
  Shape shapeToRender = null;
  ...
  // _isFlipped – закрытое булевское поле. Его значение
  // переключается при щелчке на кнопке переключения.
  if (_isFlipped)
  {
    RotateTransform rotate = new RotateTransform(-180);
    shapeToRender.RenderTransform = rotate;
  }

  // Установить верхнюю левую точку для рисования на холсте.
  Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
  Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);
  // Нарисовать фигуру.
  canvasDrawingArea.Children.Add(shapeToRender);
}

private void flipCanvas_Click(object sender, RoutedEventArgs e)
{
  if (flipCanvas.IsChecked == true)
  {
    RotateTransform rotate = new RotateTransform(-180);
    canvasDrawingArea.LayoutTransform = rotate;
  }
  else
  {
    canvasDrawingArea.LayoutTransform = null;
  }
}
```

На этом исследование пространства имен System.Windows.Shapes, кистей и трансформаций завершено. Прежде чем перейти к анализу роли визуализации графики с использованием рисунков и геометрий, давайте посмотрим, как IDE-среда Visual Studio может упростить работу с примитивными графическими элементами.

Исходный код. Проект RenderingWithShapes доступен в подкаталоге Chapter_28.

Работа с редактором трансформаций Visual Studio

В предыдущем примере разнообразные трансформации применялись за счет ручного ввода разметки и написания кода C#. Наряду с тем, что это вполне удобно, последняя версия Visual Studio поставляется со встроенным редактором трансформаций. Он не настолько мощный как инструмент Expression Blend, но позволяет легко генерировать необходимую разметку трансформаций с использованием интегрированных средств. Вспомните, что получателем служб трансформаций может быть любой элемент пользовательского интерфейса, в том числе диспетчер компоновки, содержащий различные элементы управления. Чтобы продемонстрировать работу с редактором трансформаций Visual Studio, мы создадим новый проект WPF Application по имени `FunWithTransforms`.

Построение начальной компоновки

Первым делом разделим первоначальный элемент `Grid` на две колонки с применением встроенного редактора сетки (точные размеры роли не играют). Теперь отыщем в панели инструментов элемент управления `StackPanel` и добавим его так, чтобы он занимал все пространство первой колонки `Grid`:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <StakPanelGrid.Row="0" Grid.Columne="0"></StackPanel>
</Grid>
```

Выберем новый элемент `StackPanel` в окне Document Outline (Схема документа) и поместим в него три элемента управления `Button` (рис. 28.12).

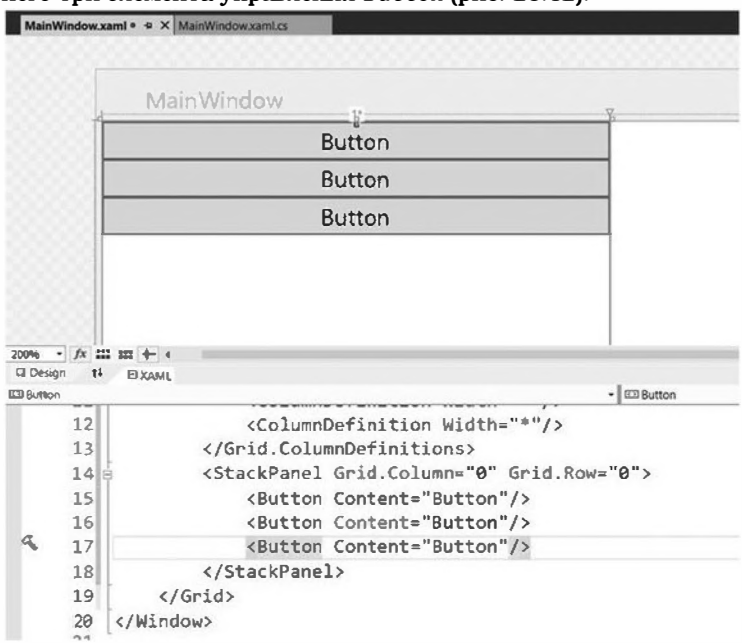


Рис. 28.12. Контейнер `StackPanel` с элементами управления `Button`

Теперь по очереди выберем каждый элемент `Button` и укажем в свойстве `Content` (в области Common (Общие) окна Properties (Свойства)) значения `Skew`, `Rotate` и `Flip`. Кроме того, в области Name (Имя) окна Properties назначим каждой кнопке подходящее имя, такое как `btnSkew`, `btnRotate` и `btnFile`, а на вкладке Events (События) обработаем событие `Click` для каждого элемента управления `Button`. Вскоре мы реализуем эти обработчики.

Для завершения пользовательского интерфейса создадим графику (используя любой прием, представленный ранее в главе) во второй колонке `Grid`. Окончательная компоновка показана на рис. 28.13. Здесь есть два элемента управления `Ellipse`, сгруппированные внутри элемента управления `Canvas` по имени `myCanvas`.

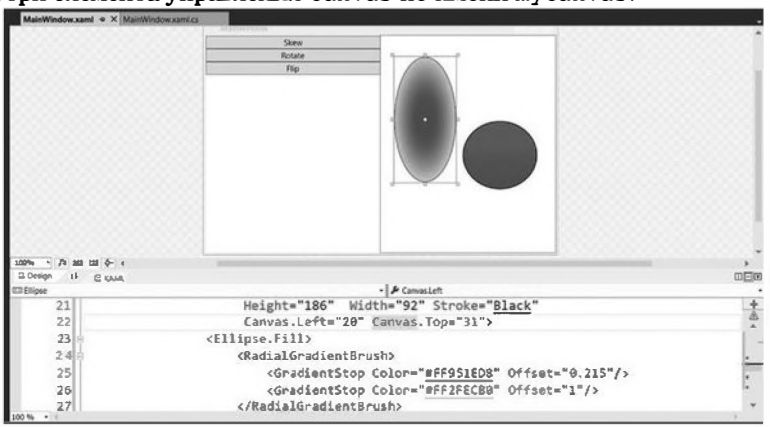


Рис. 28.13. Компоновка для примера с трансформациями

Вот разметка, применяемая в данном примере:

```
<Canvas x:Name="myCanvas" Grid.Column="1" Grid.Row="0">
  <Ellipse HorizontalAlignment="Left" VerticalAlignment="Top"
      Height="186"  Width="92" Stroke="Black"
      Canvas.Left="20" Canvas.Top="31">
    <Ellipse.Fill>
      <RadialGradientBrush>
        <GradientStop Color="#FF951ED8" Offset="0.215"/>
        <GradientStop Color="#FF2FECB0" Offset="1"/>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
  <Ellipse HorizontalAlignment="Left" VerticalAlignment="Top"
      Height="101" Width="110" Stroke="Black"
      Canvas.Left="122" Canvas.Top="126">
    <Ellipse.Fill>
      <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
        <GradientStop Color="#FFB91DDC" Offset="0.355"/>
        <GradientStop Color="#FFB0381D" Offset="1"/>
      </LinearGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
</Canvas>
```

Применение трансформаций на этапе проектирования

Как упоминалось ранее, IDE-среда Visual Studio предоставляет встроенный редактор трансформаций, который можно найти в окне Properties. Раскроем раздел Transform

Рис. 28.14. Редактор трансформаций

(Трансформация), чтобы отобразить области RenderTransform и LayoutTransform этого редактора (рис. 28.14).

Подобно разделу Brush раздел Transform предлагает несколько вкладок, предназначенных для конфигурирования разнообразных типов графической трансформации текущего выбранного элемента. В табл. 28.6 описаны варианты трансформации, доступные на этих вкладках (в порядке слева направо).

Опробуйте каждую из описанных трансформаций, используя в качестве цели специальную фигуру (для отмены выполненной операции просто нажимайте <Ctrl+Z>). Как и многие другие аспекты раздела Transform окна Properties, каждая трансформация имеет уникальный набор параметров конфигурации, которые должны стать вполне понятными, как только вы просмотрите их. Например, редактор трансформации Skew позволяет устанавливать значения скоса X и Y, а редактор трансформации Flip дает возможность зеркально отображать относительно оси X или Y и т.д.

Таблица 28.6. Варианты трансформации

Трансформация	Описание
Translate (Трансляция)	Позволяет сдвинуть местоположение элемента в позицию X,Y
Rotate (Поворот)	Позволяет повернуть элемент на угол до 360 градусов
Scale (Масштабирование)	Позволяет увеличить или уменьшить элемент на масштабный коэффициент в направлениях X и Y
Skew (Перекашивание)	Позволяет скосить ограничивающий прямоугольник, содержащий выбранный элемент, на масштабный коэффициент в направлениях X и Y
Center Point (Центральная точка)	При повороте или зеркальном отображении объекта элемент перемещается относительно фиксированной точки, которая называется центральной точкой объекта. По умолчанию центральная точка объекта расположена в центре объекта; тем не менее, эта трансформация позволяет изменять центральную точку объекта для выполнения поворота или зеркального отображения относительно другой точки
Flip (Зеркальное отображение)	Позволяет зеркально отобразить выбранный элемент на основе координаты X или Y центральной точки

Трансформация холста в коде

Реализации всех обработчиков событий `Click` будут более или менее похожими. Мы сконфигурируем объект трансформации и присвоим его объекту `myCanvas`. Затем после запуска приложения можно будет щелкать на кнопке, чтобы просматривать результат применения трансформации. Ниже приведен полный код всех обработчиков событий (обратите внимание на установку свойства `LayoutTransform`, что позволяет данным фигуры позиционироваться относительно родительского контейнера):

```
private void btnFlip_Click(object sender, System.Windows.RoutedEventArgs e)
{
  myCanvas.LayoutTransform = new ScaleTransform(-1, 1);
}
private void btnRotate_Click(object sender, System.Windows.RoutedEventArgs e)
{
  myCanvas.LayoutTransform = new RotateTransform(180);
}
private void btnSkew_Click(object sender, System.Windows.RoutedEventArgs e)
{
  myCanvas.LayoutTransform = new SkewTransform(40, -20);
}
```

Исходный код. Проект `FunWithTransformations` доступен в подкаталоге `Chapter_28`.

Визуализация графических данных с использованием рисунков и геометрий

Несмотря на то что типы `Shape` позволяют генерировать интерактивную двухмерную поверхность любого вида, из-за насыщенной цепочки наследования они потребляют довольно много памяти. И хотя класс `Path` может помочь снизить накладные расходы за счет применения включенной геометрии (вместо крупной коллекции других фигур), инфраструктура WPF предоставляет развитый API-интерфейс рисования и геометрии, который визуализирует еще более легковесные двухмерные векторные изображения.

Входной точкой в этот API-интерфейс является абстрактный класс `System.Windows.Media.Drawing` (из сборки `PresentationCore.dll`), который сам по себе всего лишь определяет ограничивающий прямоугольник для хранения визуализаций. На рис. 28.15 видно, что цепочка наследования класса `Drawing` значительно меньше, чем у класса `Shape`, поскольку `UIElement` и `FrameworkElement` в ней отсутствуют.

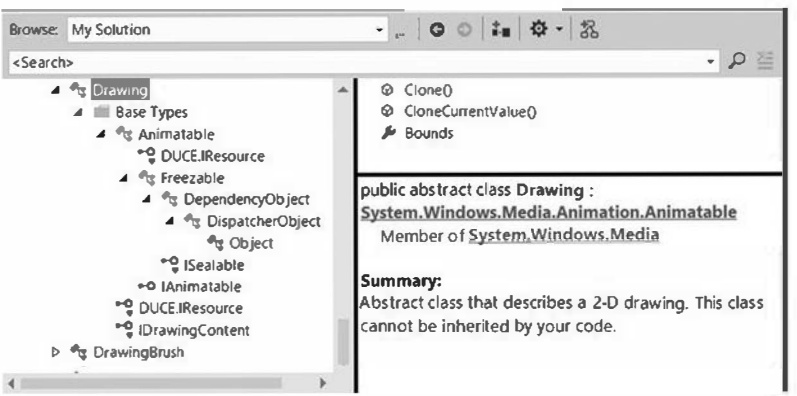


Рис. 28.15. Класс `Drawing` является более легковесным, чем `Shape`

Инфраструктура WPF предлагает разнообразные классы, расширяющие `Drawing`, каждый из которых представляет отдельный способ рисования содержимого (табл. 28.7).

Будучи более легковесными, производные от `Drawing` типы не обладают встроенной возможностью обработки событий, т.к. они не являются `UIElement` или `FrameworkElement` (хотя допускают программную реализацию логики проверки попадания).

Таблица 28.7. Классы, производные от `Drawing`

Класс	Описание
`DrawingGrdoup`	Используется для комбинирования коллекции отдельных объектов, производных от `Drawing`, в единую составную визуализацию
`GeometryDrawing`	Применяется для визуализации двухмерных фигур в очень легковесной манере
`GlyphRunDrawing`	Используется для визуализации текстовых данных с применением служб графической визуализации WPF
`ImageDrawing`	Используется для визуализации файла изображения, или набора геометрий, внутри ограничивающего прямоугольника
`VideoDrawing`	Применяется для воспроизведения аудио- или видеофайла. Этот тип может полноценно использоваться только в процедурном коде. Для воспроизведения видео в разметке XAML лучше подойдет тип `MediaPlayer`

Другое ключевое отличие между типами, производными от `Drawing`, и типами, производными от `Shape`, состоит в том, что производные от `Drawing` типы не умеют визуализировать себя, поскольку не унаследованы от `UIElement`! Для отображения содержимого производные типы должны помещаться в какой-то контейнерный объект (в частности — `DrawingImage`, `DrawingBrush` или `DrawingVisual`).

Класс `DrawingImage` позволяет помещать рисунки и геометрии внутрь элемента управления `Image` из WPF, который обычно применяется для отображения данных из внешнего файла. Класс `DrawingBrush` дает возможность строить кисть на основе рисунков и их геометрий, чтобы установить свойство, требующее кисть. Наконец, класс `DrawingVisual` используется только на "визуальном" уровне графической визуализации, полностью управляемом из кода C#.

Хотя работать с рисунками немного сложнее, чем с простыми фигурами, отделение графической композиции от графической визуализации делает типы, производные от `Drawing`, намного более легковесными, чем производные от `Shape` типы, одновременно сохраняя их ключевые службы.

Построение кисти `DrawingBrush` с использованием геометрий

Ранее в главе элемент `Path` заполнялся группой геометрий примерно так:

```
<Path Fill = "Orange" Stroke = "Blue" StrokeThickness = "3">
  <Path.Data>
    <GeometryGroup>
      <EllipseGeometry Center = "75,70"
        RadiusX = "30" RadiusY = "30" />
      <RectangleGeometry Rect = "25,55 100 30" />
      <LineGeometry StartPoint="0,0" EndPoint="70,30" />
      <LineGeometry StartPoint="70,30" EndPoint="0,30" />
    </GeometryGroup>
   </Path.Data>
</Path>
```

Поступая подобным образом, мы достигаем интерактивности `Path` при чрезвычайной легковесности, присущей геометриям. Однако если необходимо визуализировать аналогичный вывод и отсутствует потребность в любой (готовой) интерактивности, то тот же самый элемент `<GeometryGroup>` можно поместить внутрь `DrawingBrush`:

```
<DrawingBrush>
  <DrawingBrush.Drawing>
    <GeometryDrawing>
      <GeometryDrawing.Geometry>
        <GeometryGroup>
          <EllipseGeometry Center = "75,70"
                           RadiusX = "30" RadiusY = "30" />
          <RectangleGeometry Rect = "25,55 100 30" />
          <LineGeometry StartPoint="0,0" EndPoint="70,30" />
          <LineGeometry StartPoint="70,30" EndPoint="0,30" />
        </GeometryGroup>
      </GeometryDrawing.Geometry>
      <!-- Специальное перо для рисования границ -->
      <GeometryDrawing.Pen>
        <Pen Brush="Blue" Thickness="3"/>
      </GeometryDrawing.Pen>
      <!-- Специальная кисть для заполнения внутренней области -->
      <GeometryDrawing.Brush>
        <SolidColorBrush Color="Orange"/>
      </GeometryDrawing.Brush>
    </GeometryDrawing>
  </DrawingBrush.Drawing>
</DrawingBrush>
```

При помещении группы геометрий внутрь `DrawingBrush` также понадобится установить объект `Pen`, применяемый для рисования границ, потому что свойство `Stroke` больше не наследуется от базового класса `Shape`. Здесь был создан элемент `<Pen>` с теми же настройками, которые использовались в значениях `Stroke` и `StrokeThickness` из предыдущего примера `Path`.

Более того, поскольку свойство `Fill` больше не наследуется от класса `Shape`, нужно также применять синтаксис "элемент-свойство" для определения объекта кисти, предназначенного элементу `<DrawingGeometry>`, со сплошным оранжевым цветом, как в предыдущих настройках `Path`.

Рисование с помощью DrawingBrush

Теперь объект `DrawingBrush` можно использовать для установки значения любого свойства, требующего объекта кисти. Например, после подготовки следующей разметки в редакторе Kaxaml посредством синтаксиса "элемент-свойство" можно рисовать изображение по всей поверхности `Page`:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Page.Background>
    <!-- Тот же самый объект DrawingBrush, что и ранее -->
    <DrawingBrush>
      ...
    </DrawingBrush>
  </Page.Background>
</Page>
```

Или же `<DrawingBrush>` можно применять для установки другого совместимого с кистью свойства, такого как свойство `Background` элемента `Button`:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Button Height="100" Width="100">
    <Button.Background>
      <!-- Тот же самый объект DrawingBrush, что и ранее -->
      <DrawingBrush>
        ...
      </DrawingBrush>
    </Button.Background>
  </Button>
</Page>
```

Независимо от того, какое совместимое с кистью свойство устанавливается с использованием специального объекта <DrawingBrush>, визуализация двухмерного графического изображения в итоге получается с намного меньшими накладными расходами, чем в случае визуализации того же изображения посредством фигур.

Включение типов Drawing в DrawingImage

Тип DrawingImage позволяет подключать рисованную геометрию к элементу управления <Image> из WPF. Взгляните на следующую разметку:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Image Height="100" Width="100">
    <Image.Source>
      <DrawingImage>
        <DrawingImage.Drawing>
          <GeometryDrawing>
            <GeometryDrawing.Geometry>
              <GeometryGroup>
                <EllipseGeometry Center = "75,70"
                                 RadiusX = "30" RadiusY = "30" />
                <RectangleGeometry Rect = "25,55 100 30" />
                <LineGeometry StartPoint="0,0" EndPoint="70,30" />
                <LineGeometry StartPoint="70,30" EndPoint="0,30" />
              </GeometryGroup>
            </GeometryDrawing.Geometry>
            <!-- Специальное перо для рисования границ -->
            <GeometryDrawing.Pen>
              <Pen Brush="Blue" Thickness="3"/>
            </GeometryDrawing.Pen>
            <!-- Специальная кисть для заполнения внутренней области -->
            <GeometryDrawing.Brush>
              <SolidColorBrush Color="Orange"/>
            </GeometryDrawing.Brush>
          </GeometryDrawing>
        </DrawingImage.Drawing>
      </DrawingImage>
    </Image.Source>
  </Image>
</Page>
```

В этом случае элемент <GeometryDrawing> был помещен внутрь <DrawingImage>, а не в <DrawingBrush>. С применением <DrawingImage> можно установить свойство Source элемента управления Image.

Работа с векторными изображениями

Вы наверняка согласитесь с тем, что художнику будет весьма затруднительно создавать сложное векторное изображение с использованием инструментов и приемов, предоставляемых средой Visual Studio. В распоряжении художников имеются собственные наборы инструментов, которые позволяют производить замечательную векторную графику. Изобразительными возможностями подобного рода не обладает ни среда Visual Studio, ни сопровождающее ее средство Expression Blend. Перед тем, как векторные изображения можно будет импортировать в приложение WPF, они должны быть преобразованы в выражения путей. После этого можно программировать с применением сгенерированной объектной модели, используя Visual Studio.

На заметку! В предшествующих изданиях книги демонстрировался программный пакет под названием Expression Design. Он был одним из продуктов в рамках Expression Studio, но, к сожалению, данный полный комплект больше не обновляется. В то время как это программное обеспечение по-прежнему доступно при наличии подписки MSDN, в настоящем издании книги применяется программное обеспечение с открытым кодом и трюк с печатью для преобразования векторной графики в требуемую информацию путей для визуализации XAML. Используемое изображение (laser_sign.svg), а также экспортированные данные путей (laser_sign.xaml) включены в подкаталог Chapter_28 загружаемого кода примеров. Изображение взято из статьи Википедии по адресу https://ru.wikipedia.org/wiki/Символы_опасности.

Преобразование файла с векторной графикой в XAML

Прежде чем можно будет импортировать сложные графические данные (такие как векторная графика) в приложение WPF, графику понадобится преобразовать в данные путей. Чтобы проиллюстрировать, как это делается, возьмем пример файла изображения .svg с упомянутым выше знаком опасности лазерного излучения. Затем загрузим и установим инструмент с открытым кодом под названием Inkscape (находящийся по адресу www.inkscape.org). С помощью Inkscape откроем файл laser_sign.svg из подкаталога Chapter_28. Изображение должно быть похожим на то, что показано на рис. 28.16.

На заметку! Очень хорошим бесплатным инструментом для обработки изображений также является ImageMagick (www.ImageMagick.org). К сожалению, трюк, о котором вы вскоре узнаете, не работает в ImageMagick на машине Windows 10.

Следующие шаги поначалу покажутся несколько странными, но на самом деле они представляют собой простой способ преобразования векторных изображений в разметку XAML. Когда изображение приобрело желаемый вид, необходимо выбрать пункт меню File⇨Print (Файл⇨Печать). Далее потребуется указать Microsoft XPS Document Writer в качестве целевого принтера и щелкнуть на кнопке Print (Печать), как показано на рис. 28.17. В открывшемся окне следует ввести имя файла и выбрать место, где он должен быть сохранен, после чего щелкнуть на кнопке Save (Сохранить). В результате получается завершенный файл *.xps (или *.oxps).

На заметку! В зависимости от нескольких переменных в конфигурации системы сгенерированный файл будет иметь либо расширение .xps, либо расширение .oxps. В любом случае дальнейший процесс идентичен.

Файл `*.xps` или `*.oxps` в действительности является архивом ZIP. Переименовав расширение файла в `.zip`, его можно открыть в проводнике файлов (или в предпочитаемой утилите архивации). Файл содержит иерархию папок, приведенную на рис. 28.18.

Рис. 28.16. Пример графики со знаком опасности лазерного излучения в Inkscape

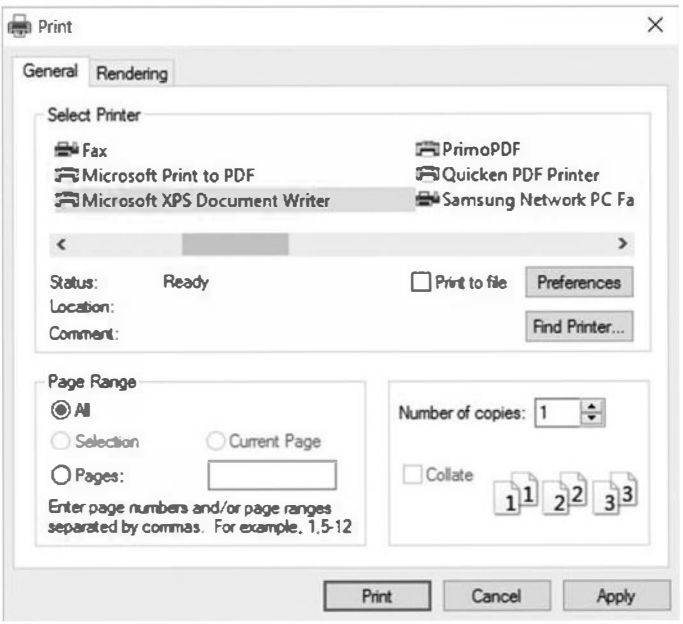


Рис. 28.17. Вывод графики на принтер Microsoft XPS Document Printer

Необходимый файл находится в папке `Pages` (`Documents/1/Pages`) и называется `1.fpage`. Откроем этот файл в текстовом редакторе и скопируем в буфер все данные кроме открывающего и закрывающего дескрипторов `<FixedPage>`. Данные путей затем могут быть помещены внутрь элемента `Canvas` главного окна в (написанном ранее) приложении `MyXamlPad`. После щелчка на кнопке View XAML (Просмотреть XAML) отобразится векторная графика, воспроизведенная в разметке XAML. На рис. 28.19 показано изображение, визуализированное с применением `MyXamlPad`.

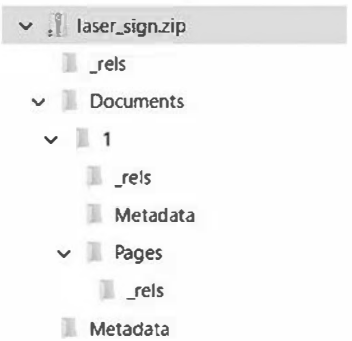


Рис. 28.18. Иерархия папок в файле `*.xps` или `*.oxps`

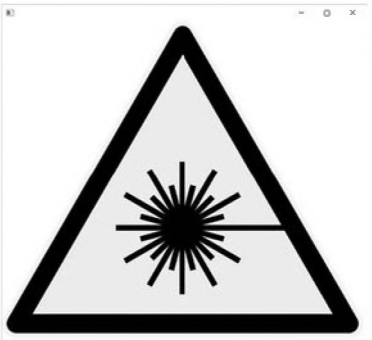


Рис. 28.19. Векторная графика, визуализированная в XAML

Импортирование графических данных в проект WPF

Создадим новый проект WPF Application по имени `InteractiveLaserSign`. Изменим значения свойств `Height` и `Width` первоначального элемента `Window` следующим образом, удалим исходный элемент управления `Grid` и заменим его `Canvas`:

```
<Window x:Class="InteractiveTeddyBear.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="625" Width="675">
  <Canvas>
  </Canvas>
</Window>
```

Скопируем полную разметку XAML из `MyXamlPad` (исключая внешний элемент `Canvas`) и вставим ее в элемент управления `Canvas` внутри `MainWindow`. Просмотрев окно в режиме проектирования, легко удостовериться в том, что знак опасности лазерного излучения успешно воспроизведен в приложении.

Если заглянуть в окно Document Outline, то можно заметить, что учтен каждый элемент XAML. Цель здесь в том, чтобы найти пару линий и назначить этим элементам имена. Хотя можно было бы вручную поискать корректные объекты в коде (что будет весьма утомительным), лучше щелкнуть на них в визуальном конструкторе. Это приведет к автоматическому выделению нужного узла в окне Document Outline. Для образования каждой линии данная графика использует два объекта, поэтому щелкнем на одной линии и посмотрим, где она представлена в окне Document Outline. Проверим объекты выше и ниже, чтобы выяснить, какая пара образовалась, после чего назначим им имена `Line1_1` и `Line1_2`. Повторим процесс в отношении другой линии, именуя ее объекты как `Line2_1` и `Line2_2`. Чтобы облегчить взаимодействие, изменим черный цвет кисти для `Line1_1` и `Line2_1` на другой, выбрав объект в окне Document Outline и изменив цвет кисти, как это делалось ранее.

Взаимодействие с объектами изображения

Теперь обработаем события для объектов. Выберем `Line1_1` и `Line2_1` в визуальном конструкторе, перейдем на вкладку Events (События) в окне Properties и введем требуемые имена обработчиков событий. В текущем примере для каждого объекта мы обработаем событие `MouseLeftButtonDown`, указывая каждый раз уникальное имя метода.

Ниже приведен простой код C#, который будет изменять внешний вид каждого объекта после щелчка на нем (если не хотите вводить весь показанный здесь код, то можете поместить в обработчики только вызовы `MessageBox.Show()`, отображающие подходящие сообщения):

```
private void Line1_1_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
  // Изменить цвет при щелчке.
  Line1_2.Fill = new SolidColorBrush(Colors.Red);
}
private void Line2_1_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
  // Размыть при щелчке.
  System.Windows.Media.Effects.BlurEffect blur =
    new System.Windows.Media.Effects.BlurEffect();
  blur.Radius = 10;
  Line2_1.Effect = blur;
}
```

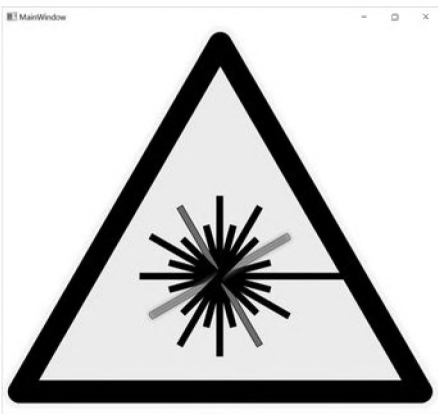


Рис. 28.20. Взаимодействие со сложными графическими данными

Запустим приложение и щелкнем на линиях, чтобы увидеть эффекты. Результаты должны быть похожими на представленные на рис. 28.20. Теперь вы понимаете процесс генерации данных путей для сложной графики и знаете, как взаимодействовать с графическими данными в коде. Вы наверняка согласитесь, что наличие у профессиональных художников возможности генерировать сложные графические данные и экспортировать их в виде разметки XAML исключительно важна. После того как графические данные сохранены в файле XAML, разработчики могут импортировать разметку и программировать в отношении объектной модели.

Исходный код. Проект `InteractiveLaserSign` доступен в подкаталоге `Chapter_28`.

Визуализация графических данных с использованием визуального уровня

Последний вариант визуализации графических данных с помощью WPF называется *визуальным уровнем*. Ранее уже упоминалось, что доступ к этому уровню возможен только из кода (он не дружественен по отношению к разметке XAML). Несмотря на то что подавляющее большинство приложений WPF будут хорошо работать с применением фигур, рисунков и геометрий, визуальный уровень обеспечивает самый быстрый способ визуализации крупных объемов графических данных. Как ни странно, он также может быть полезен, когда необходимо визуализировать единственное изображение на очень большой площади.

Например, если требуется заполнить фон окна простым статическим изображением, то визуальный уровень будет наиболее быстрым способом сделать это. Кроме того, он удобен, когда нужно очень быстро менять фон окна в зависимости от ввода пользователя или чего-нибудь еще.

Давайте построим небольшую программу, иллюстрирующую основы использования визуального уровня.

Базовый класс `Visual` и производные дочерние классы

Абстрактный класс `System.Windows.Media.Visual` предлагает минимальный набор служб (визуализацию, проверку попадания, трансформации) для визуализации графики, но не предоставляет поддержку дополнительных невизуальных служб, которые могут приводить к разбуханию кода (события ввода, службы компоновки, стили и привязка данных). Обратите внимание на простую цепочку наследования для типа `Visual`, показанную на рис. 28.21.

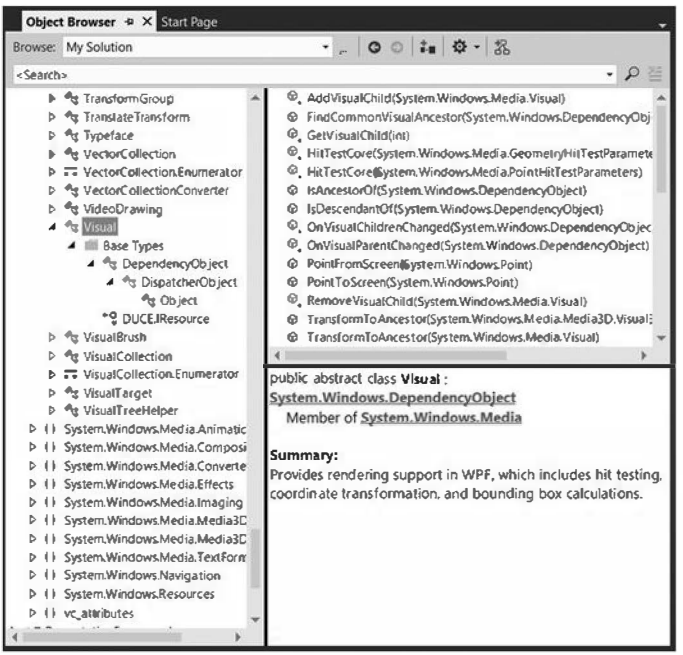


Рис. 28.21. Класс `Visual` предоставляет базовую проверку попадания, трансформацию координат и вычисление ограничивающих прямоугольников

Поскольку `Visual` — абстрактный базовый класс, для выполнения действительных операций визуализации должен применяться один из его производных классов. В WPF определено несколько подклассов `Visual`, в том числе `DrawingVisual`, `Viewport3DVisual` и `ContainerVisual`.

В рассматриваемом далее примере мы сосредоточимся только на `DrawingVisual`, легковесном классе рисования, который используется для визуализации фигур, изображений или текста.

Первый взгляд на класс `DrawingVisual`

Чтобы визуализировать данные на поверхности с применением класса `DrawingVisual`, понадобится выполнить следующие основные шаги:

- получить объект DrawingContext из DrawingVisual;
- использовать объект DrawingContext для визуализации графических данных.

Эти два шага представляют абсолютный минимум, необходимый для визуализации каких-то данных на поверхности. Тем не менее, когда нужно, чтобы визуализируемые графические данные реагировали на вычисления при проверке попадания (что важно для добавления взаимодействия с пользователем), потребуется также выполнить дополнительные шаги:

- обновить логическое и визуальное деревья, поддерживаемые контейнером, на котором производится визуализация;
- переопределить два виртуальных метода из класса FrameworkElement, позволив контейнеру получать созданные визуальные данные.

Давайте исследуем последние два шага более подробно. Чтобы продемонстрировать применение класса DrawingVisual для визуализации двухмерных данных, создадим в Visual Studio новый проект WPF Application по имени RenderingWithVisuals. Нашей первой целью будет использование класса DrawingVisual для динамического присваивания данных элементу управления Image из WPF. Начнем со следующего обновления разметки XAML окна:

```
<Window x:Class="RenderingWithVisuals.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title=" Fun with the Visual Layer" Height="350" Width="525"
        Loaded="Window_Loaded" WindowStartupLocation="CenterScreen">
    <StackPanel Background="AliceBlue" Name="myStackPanel">
      <Image Name="myImage" Height="80"/>
    </StackPanel>
</Window>
```

Обратите внимание, что элемент управления <Image> пока не имеет значения в свойстве Source, т.к. оно будет установлено во время выполнения. Кроме того, создан обработчик события Loaded окна, который выполняет работу по построению графических данных в памяти с применением объекта DrawingBrush. Вот реализация обработчика события Loaded:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  const int TextFontSize = 30;
  // Создать объект System.Windows.Media.FormattedText.
  FormattedText text = new FormattedText("Hello Visual Layer!",
    new System.Globalization.CultureInfo("en-us"),
    FlowDirection.LeftToRight,
    new Typeface(this.FontFamily, FontStyles.Italic,
                 FontWeights.DemiBold, FontStretches.UltraExpanded),
    TextFontSize,
    Brushes.Green);
  // Создать объект DrawingVisual и получить объект DrawingContext.
  DrawingVisual drawingVisual = new DrawingVisual();
  using(DrawingContext drawingContext = drawingVisual.RenderOpen())
  {
    // Вызвать любой из методов DrawingContext для визуализации данных.
    drawingContext.DrawRoundedRectangle(Brushes.Yellow, new Pen(Brushes.Black, 5),
      new Rect(5, 5, 450, 100), 20, 20);
    drawingContext.DrawText(text, new Point(20, 20));
  }
```

```
    // Динамически создать битовое изображение,
    // используя данные в объекте DrawingVisual.
    RenderTargetBitmap bmp =
         new RenderTargetBitmap(500, 100, 100, 90, PixelFormats.Pbgra32);
    bmp.Render(drawingVisual);
    // Установить источник для элемента управления Image.
    myImage.Source = bmp;
}
```

В этом коде задействовано несколько новых классов WPF, которые будут кратко описаны ниже (полные сведения можно получить в документации .NET Framework 4.6 SDK). Метод начинается с создания нового объекта `FormattedText`, который представляет текстовую часть конструируемого изображения в памяти. Как видите, конструктор позволяет указывать многочисленные атрибуты, в том числе размер шрифта, семейство шрифтов, цвет переднего плана и сам текст.

Затем посредством вызова метода `RenderOpen()` на экземпляре `DrawingVisual` получается необходимый объект `DrawingContext`. Здесь в `DrawingVisual` визуализируется цветной прямоугольник со скругленными углами, за которым следует форматированный текст. В обоих случаях графические данные помещаются в `DrawingVisual` с применением жестко закодированных значений, что не слишком хорошо в производственном приложении, но сгодится в этом простом тесте.

На заметку! Обязательно просмотрите описание класса `DrawingContext` в документации .NET Framework 4.6 SDK, чтобы ознакомиться со всеми членами, связанными с визуализацией. Если в прошлом вы работали с объектом `Graphics` из Windows Forms, то `DrawingContext` должен выглядеть очень похожим.

Несколько последних операторов отображают `DrawingVisual` на объект `RenderTargetBitmap`, который является членом пространства имен `System.Windows.Media.Imaging`. Этот класс принимает визуальный объект и трансформирует его в растровое изображение, находящееся в памяти. Затем устанавливается свойство `Source` элемента управления `Image` и получается вывод, показанный на рис. 28.22.

Рис. 28.22. Использование визуального уровня для визуализации находящегося в памяти растрового изображения

На заметку! Пространство имен `System.Windows.Media.Imaging` содержит дополнительные классы кодирования, которые позволяют сохранять находящийся в памяти объект `RenderTargetBitmap` в физический файл с разнообразными форматами. Детали ищите в описании `JpegBitmapEncoder` и связанных с ним классов.

Визуализация графических данных в специальном диспетчере компоновки

Хотя применение `DrawingVisual` для рисования на фоне элемента управления WPF представляет интерес, возможно чаще придется строить специальный диспетчер ком-

поновки (Grid, StackPanel, Canvas и т.д.), который внутренне использует визуальный уровень для визуализации своего содержимого. После создания такого специального диспетчера компоновки его можно подключить к обычному элементу Window (а также Page или UserControl) и позволить части пользовательского интерфейса использовать высоко оптимизированный агент визуализации, в то время как для визуализации некритичных графических данных будут применяться фигуры и рисунки.

Если дополнительная функциональность, предлагаемая специализированным диспетчером компоновки, не требуется, то можно просто расширить класс FrameworkElement, который имеет необходимую инфраструктуру, позволяющую содержать также и визуальные элементы. В целях иллюстрации вставим в проект новый класс по имени CustomVisualFrameworkElement. Унаследуем его от FrameworkElement и импортируем пространства имен System.Windows, System.Windows.Input и System.Windows.Media.

Класс CustomVisualFrameworkElement будет поддерживать переменную-член типа VisualCollection, которая содержит два фиксированных объекта DrawingVisual (конечно, в эту коллекцию можно было бы добавлять члены с помощью мыши, но мы решили сохранить пример простым). Модифицируем код класса следующим образом:

```
class CustomVisualFrameworkElement : FrameworkElement
{
  // Коллекция всех визуальных объектов.
  VisualCollection theVisuals;

  public CustomVisualFrameworkElement()
  {
    // Заполнить коллекцию VisualCollection несколькими
    // объектами DrawingVisual.
    // Аргумент конструктора представляет владельца визуальных объектов.
    theVisuals = new VisualCollection(this);
    theVisuals.Add(AddRect());
    theVisuals.Add(AddCircle());
  }
  private Visual AddCircle()
  {
    DrawingVisual drawingVisual = new DrawingVisual();

    // Получить объект DrawingContext для создания нового содержимого.
    using (DrawingContext drawingContext = drawingVisual.RenderOpen())
    {
      // Создать круг и нарисовать его в DrawingContext.
      Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
      drawingContext.DrawEllipse(Brushes.DarkBlue, null,
          new Point(70, 90), 40, 50);
    }
    return drawingVisual;
  }
  private Visual AddRect()
  {
    DrawingVisual drawingVisual = new DrawingVisual();
    using (DrawingContext drawingContext = drawingVisual.RenderOpen())
    {
      Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
      drawingContext.DrawRectangle(Brushes.Tomato, null, rect);
    }
    return drawingVisual;
  }
}
```

Прежде чем специальный элемент FrameworkElement можно будет использовать внутри Window, потребуется переопределить два упомянутых ранее ключевых виртуальных члена, которые вызываются внутренне инфраструктурой WPF во время процесса визуализации. Метод GetVisualChild() возвращает из коллекции дочерних элементов дочерний элемент по указанному индексу. Свойство VisualChildrenCount, допускающее только чтение, возвращает количество визуальных дочерних элементов внутри визуальной коллекции. Оба члена легко реализовать, т.к. всю реальную работу можно делегировать переменной-члену типа VisualCollection:

```
protected override int VisualChildrenCount
{
  get { return theVisuals.Count; }
}
protected override Visual GetVisualChild(int index)
{
  // Значение должно быть больше нуля, поэтому разумно это проверить.
  if (index < 0 || index >= theVisuals.Count)
  {
    throw new ArgumentOutOfRangeException();
  }
  return theVisuals[index];
}
```

Теперь мы располагаем достаточной функциональностью, чтобы протестировать наш специальный класс. Модифицируем описание XAML элемента Window, добавив в существующий контейнер StackPanel один объект CustomVisualFrameworkElement. Это потребует создания специального пространства имен XML, которое отображается на пространство имен .NET.

```
<Window x:Class="RenderingWithVisuals.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:custom="clr-namespace:RenderingWithVisuals"
  Title="Fun with the Visual Layer" Height="350" Width="525"
  Loaded="Window_Loaded" WindowStartupLocation="CenterScreen">
    <StackPanel Background="AliceBlue" Name="myStackPanel">
        <Image Name="myImage" Height="80"/>
        <custom:CustomVisualFrameworkElement/>
    </StackPanel>
</Window>
```

Результат выполнения программы показан на рис. 28.23.

Реагирование на операции проверки попадания

Поскольку класс DrawingVisual не поддерживает инфраструктуру UIElement или FrameworkElement, необходимо программно добавить возможность реагирования на операции проверки попадания. К счастью, на визуальном уровне сделать это очень просто благодаря концепции *логического* и *визуального* деревьев. Оказывается, что в результате написания блока XAML по существу строится логическое дерево элементов. Однако с каждым логическим деревом связано намного более развитое описание, известное как визуальное дерево, которое содержит низкоуровневые инструкции визуализации.

Данные деревья подробно рассматриваются в главе 29, а сейчас достаточно знать, что до тех пор, пока специальные визуальные объекты не будут зарегистрированы в этих структурах данных, выполнять операции проверки попадания невозможно. К счастью, контейнер VisualCollection обеспечивает регистрацию автоматически (вот почему в аргументе конструктора необходимо передавать ссылку на специальный элемент FrameworkElement).

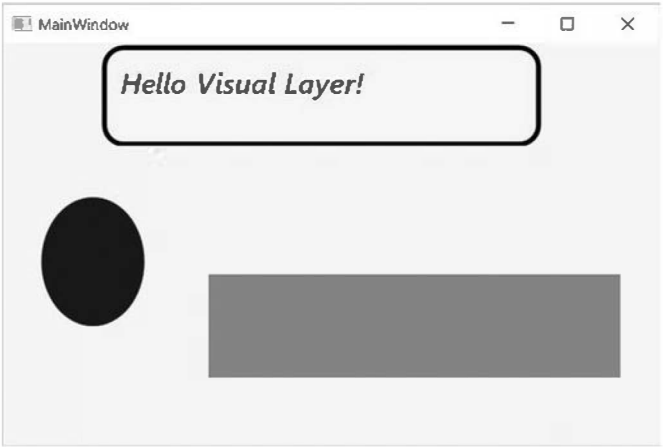


Рис. 28.23. Использование визуального уровня для визуализации данных в специальном элементе `FrameworkElement`

Изменим код класса `CustomVisualFrameworkElement` **для обработки события** `MouseDown` **в конструкторе класса с применением стандартного синтаксиса C#:**

```
this.MouseDown += MyVisualHost_MouseDown;
```

Реализация этого обработчика будет вызывать метод `VisualTreeHelper.HitTest()` **для выяснения, находится ли курсор мыши внутри границ одного из визуальных объектов. Чтобы сделать это, в одном из параметров метода** `HitTest()` **указывается делегат** `HitTestResultCallback`**, который будет выполнять вычисления. Добавим в класс** `CustomVisualFrameworkElement` **следующие методы:**

```
void MyVisualHost_MouseDown(object sender, MouseButtonEventArgs e)
{
  // Выяснить, где пользователь выполнил щелчок.
  Point pt = e.GetPosition((UIElement)sender);
  // Вызвать вспомогательную функцию через делегат, чтобы
  // посмотреть, был ли совершен щелчок на визуальном объекте.
  VisualTreeHelper.HitTest(this, null,
  new HitTestResultCallback(myCallback), new PointHitTestParameters(pt));
}
public HitTestResultBehavior myCallback(HitTestResult result)
{
  // Если щелчок был совершен на визуальном объекте, то
  // переключиться между скошенной и нормальной визуализацией.
  if (result.VisualHit.GetType() == typeof(DrawingVisual))
  {
    if (((DrawingVisual)result.VisualHit).Transform == null)
    {
      ((DrawingVisual)result.VisualHit).Transform = new SkewTransform(7, 7);
    }
    else
    {
      ((DrawingVisual)result.VisualHit).Transform = null;
    }
  }
  // Сообщить методу HitTest() о прекращении углубления в визуальное дерево.
  return HitTestResultBehavior.Stop;
}
```

Запустим программу снова. Теперь должна быть возможность щелкать на любом из отображенных визуальных объектов и наблюдать трансформацию в действии. Наряду с тем, что рассмотренный пример взаимодействия с визуальным уровнем WPF очень прост, не забывайте, что здесь можно использовать те же самые кисти, трансформации, перья и диспетчеры компоновки, которые обычно применяются в разметке XAML. Таким образом, вы уже знаете довольно много о работе с классами, производными от `Visual`.

Исходный код. Проект `RenderingWithVisuals` доступен в подкаталоге `Chapter_28`.

На этом исследование служб графической визуализации WPF завершено. Несмотря на раскрытие ряда интересных тем, на самом деле мы лишь слегка затронули обширную область графических возможностей инфраструктуры WPF. Дальнейшее изучение фигур, рисунков, кистей, трансформаций и визуальных объектов вы можете продолжить самостоятельно (в оставшихся главах, посвященных WPF, еще встретятся дополнительные детали).

Резюме

Поскольку Windows Presentation Foundation является чрезвычайно насыщенной графикой инфраструктурой для построения графических пользовательских интерфейсов, не удивительно, что существует несколько способов визуализации графического вывода. Глава начиналась с рассмотрения всех трех подходов к визуализации (фигуры, рисунки и визуальные объекты), а также разнообразных примитивов визуализации, таких как кисти, перья и трансформации.

Вспомните, что когда необходимо строить интерактивную двухмерную визуализацию, то фигуры делают этот процесс очень простым. С другой стороны, статические, не интерактивные изображения могут визуализироваться в более оптимальной манере с использованием рисунков и геометрии, а визуальный уровень (доступный только в коде) обеспечивает максимальный контроль и производительность.

ГЛАВА 29

Ресурсы, анимация, стили и шаблоны WPF

В этой главе будут представлены четыре важные (и взаимосвязанные) темы, которые позволят углубить понимание API-интерфейса Windows Presentation Foundation (WPF). Первым делом вы изучите роль *логических ресурсов*. Вы увидите, что система логических ресурсов (также называемых *объектными ресурсами*) — это способ ссылаться на часто используемые объекты внутри приложения WPF. Хотя логические ресурсы нередко пишутся на XAML, они могут быть определены и в процедурном коде.

Вы узнаете, как определять, выполнять и управлять последовательностью анимации. Вопреки тому, что можно было подумать, применение анимации WPF не ограничивается видеоиграми или мультимедиа-приложениями. В API-интерфейсе WPF анимация может использоваться, например, для подсветки кнопки, когда она получает фокус, или увеличения размера выбранной строки в `DataGrid`. Понимание анимации является ключевым аспектом построения специальных шаблонов элементов управления (как вы увидите позже в этой главе).

Затем будет объяснена роль стилей и шаблонов WPF. Подобно веб-странице, в которой применяются стили CSS или механизм тем ASP.NET, приложение WPF может определять общий вид и поведение для набора элементов управления. Такие стили можно определять в разметке и сохранять их в виде объектных ресурсов для последующего использования, а также динамически применять во время выполнения. В последнем примере вы научитесь строить специальные шаблоны элементов управления.

Система ресурсов WPF

Нашей первой задачей будет исследование темы встраивания и доступа к ресурсам приложения. Инфраструктура WPF поддерживает два вида ресурсов. Первый из них — *двоичные ресурсы*, и эта категория обычно включает элементы, которые большинство программистов считают ресурсами в традиционном смысле (встроенные файлы изображений или звуковых клипов, значки, используемые приложением, и т.д.).

Вторая категория, называемая *объектными ресурсами* или *логическими ресурсами*, представляет именованные объекты .NET, которые можно упаковывать и многократно применять повсюду в приложении. Несмотря на то что упаковывать в виде объектного ресурса разрешено любой объект .NET, логические ресурсы особенно удобны при работе с графическими данными любого рода, поскольку можно определить часто используемые графические примитивы (кисти, перья, анимации и т.д.) и ссылаться на них по мере необходимости.

Работа с двоичными ресурсами

Прежде чем перейти к теме объектных ресурсов, давайте кратко проанализируем, как упаковывать *двоичные ресурсы* вроде значков и файлов изображений (например, логотипов компании либо изображений для анимации) внутрь приложений. Создадим в Visual Studio новый проект WPF Application (Приложение WPF) по имени `BinaryResourcesApp`. Модифицируем разметку начального окна для применения `DockPanel` в качестве корня компоновки:

```
<Window x:Class="BinaryResourcesApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:BinaryResourcesApp"
    mc:Ignorable="d"
  Title="Fun with Binary Resources" Height="500" Width="649"
  Loaded="MainWindow_OnLoaded">
  <DockPanel LastChildFill="True">
  </DockPanel>
</Window>
```

Предположим, что приложение должно отображать внутри части окна один из трех файлов изображений, основываясь на пользовательском вводе. Элемент управления `Image` из WPF может использоваться не только для отображения типичного файла изображения (`*.bmp`, `*.gif`, `*.ico`, `*.jpg`, `*.png`, `*.wdp` или `*.tiff`), но также данных объекта `DrawingImage` (как было показано в главе 28). Построим пользовательский интерфейс окна, который поддерживает диспетчер компоновки `DockPanel`, содержащий простую панель инструментов с кнопками Next (Вперед) и Previous (Назад). Под панелью инструментов расположен элемент управления `Image`, свойство `Source` которого в текущий момент не установлено:

```
<Window x:Class="BinaryResourcesApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:BinaryResourcesApp"
    mc:Ignorable="d"
  Title="Fun with Binary Resources" Height="500" Width="649">
  Loaded="MainWindow_OnLoaded">
  <DockPanel LastChildFill="True">
    <ToolBar Height="60" Name="picturePickerToolbar" DockPanel.Dock="Top">
      <Button x:Name="btnPreviousImage" Height="40" Width="100" BorderBrush="Black"
              Margin="5" Content="Previous" Click="btnPreviousImage_Click"/>
      <Button x:Name="btnNextImage" Height="40" Width="100" BorderBrush="Black"
              Margin="5" Content="Next" Click="btnNextImage_Click"/>
    </ToolBar>
    <!-- Этот элемент Image будет заполняться в коде -->
    <Border BorderThickness="2" BorderBrush="Green">
      <Image x:Name="imageHolder" Stretch="Fill" />
    </Border>
  </DockPanel>
</Window>
```

Обратите внимание, что событие `Click` обрабатывается для каждого объекта `Button`. Предполагая, что для обработки событий применялась IDE-среда, в файле кода C# будут находиться три пустых метода. Каким образом реализовать обработчики событий `Click` для прохода в цикле по графическим данным? И что более важно: должны ли графические данные храниться на жестком диске пользователя или же быть встроенными в скомпилированную сборку? Давайте исследуем все возможные варианты.

Включение в проект несвязанных файлов ресурсов

Пусть файлы изображений необходимо поставлять как набор несвязанных файлов в каком-то подкаталоге пути установки приложения. В окне Solution Explorer среды Visual Studio щелкнем правой кнопкой мыши на узле проекта и выберем в контекстном меню пункт Add⇨New Folder (Добавить⇨Новая папка), чтобы создать папку для файлов изображений, которая должна быть названа `Images`.

Теперь щелкнем правой кнопкой мыши на папке `Images` и выберем в контекстном меню пункт Add⇨Existing Item (Добавить⇨Существующий элемент), чтобы скопировать в нее файлы изображений. В исходном коде для этого проекта доступны три файла изображений с именами `Deep.jpg`, `Dogs.jpg` и `Welcome.jpg`, которые можно включить в проект; при желании можно добавить три других файла изображений. На рис. 29.1 показан текущий вид окна Solution Explorer.

Конфигурирование несвязанных ресурсов

Если необходимо, чтобы IDE-среда Visual Studio копировала содержимое проекта в выходной каталог, то потребуется скорректировать несколько настроек в окне Properties (Свойства). Чтобы обеспечить копирование содержимого папки `Images` в папку `\bin\Debug`, начнем с выбора всех изображений в окне Solution Explorer. Когда все изображения выбраны, в окне Properties установим свойство Build Action (Действие сборки) в Content (Содержимое), а свойство Copy Output Directory (Копировать в выходной каталог) в Copy always (Копировать всегда), как показано на рис. 29.2.

Рис. 29.1. Новый подкаталог в проекте WPF, который содержит данные изображений

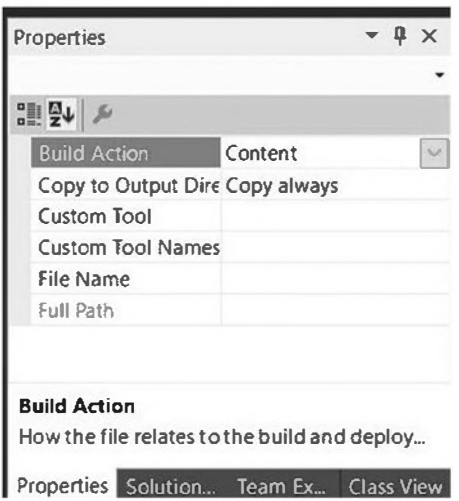


Рис. 29.2. Конфигурирование данных изображений для копирования в выходной каталог

На заметку! Для свойства Copy Output Directory можно было бы также выбрать вариант Copy if Newer (Копировать, если новее), что позволит сократить время копирования при построении крупных проектов с большим объемом содержимого. В рассматриваемом примере варианта Copy always вполне достаточно.

После перекомпиляции программы появится возможность щелкнуть на кнопке Show all Files (Показать все файлы) в окне Solution Explorer и просмотреть скопированную папку Images в каталоге \bin\Debug (может также потребоваться щелкнуть на кнопке Refresh (Обновить)). Результат показан на рис. 29.3.

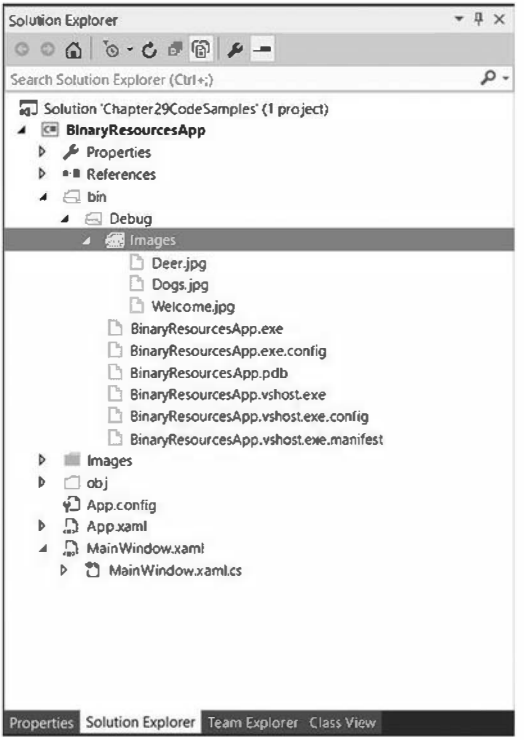


Рис. 29.3. Скопированные данные изображений

Программная загрузка изображения

Инфраструктура WPF предоставляет класс по имени BitmapImage, определенный в пространстве имен System.Windows.Media.Imaging. Он позволяет загружать данные из файла изображения, местоположение которого представлено объектом System.Uri. Наступило время обработать событие Loaded окна для заполнения списка List<T> элементов BitmapImage:

```
public partial class MainWindow : Window
{
  //Список файлов BitmapImage.
  List<BitmapImage> _images = new List<BitmapImage>();
  // Текущая позиция в списке.
  private int _currImage = 0;
  private const int MAX_IMAGES = 2;
  private void MainWindow_OnLoaded(object sender, RoutedEventArgs e)
  {
    try
    {
      string path = Environment.CurrentDirectory;
      // Загрузить эти изображения во время загрузки окна.
      _images.Add(new BitmapImage(new Uri($@"{path}\Images\Deer.jpg")));
      _images.Add(new BitmapImage(new Uri($@"{path}\Images\Dogs.jpg")));
      _images.Add(new BitmapImage(new Uri($@"{path}\Images\Welcome.jpg")));
      // Показать первое изображение в List<>.
      imageHolder.Source = _images[_currImage];
    }
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message);
    }
  }
  ...
}
```

В этом классе также определена переменная-член типа int (_currImage), которая позволит обработчикам событий Click проходить в цикле по каждому элементу в List<T> и отображать его в элементе управления Image, устанавливая свойство Source. (Здесь обработчик события Loaded устанавливает свойство Source в первое изображение из List<T>.) Вдобавок константа MAX_IMAGES даст возможность проверить верхнюю и нижнюю границы во время итерации по списку. Ниже показаны обработчики Click, которые делают именно это:

```
private void btnPreviousImage_Click(object sender, RoutedEventArgs e)
{
  if (--_currImage < 0)
    _currImage = MAX_IMAGES;
  imageHolder.Source = _images[_currImage];
}
private void btnNextImage_Click(object sender, RoutedEventArgs e)
{
  if (++_currImage > MAX_IMAGES)
    _currImage = 0;
  imageHolder.Source = _images[_currImage];
}
```

Теперь можно запустить программу и переключаться между всеми изображениями.

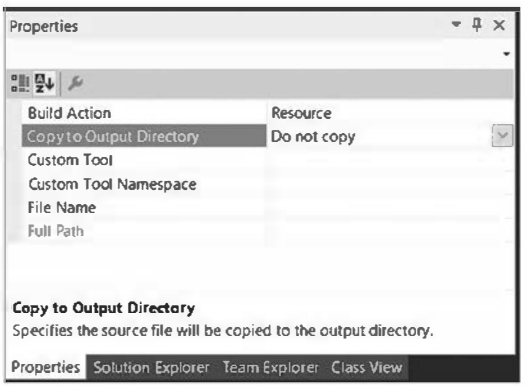


Рис. 29.4. Конфигурирование изображений как встроенных ресурсов

Встраивание ресурсов приложения

Если взамен файлы изображений нужно встроить прямо в сборку .NET как двоичные ресурсы, то понадобится выбрать файлы изображений в окне Solution Explorer (из папки \Images, а не \bin\Debug\Images) и установить свойство Build Action в Resource (Ресурс), а свойство Copy to Output Directory — в Do not copy (Не копировать), как показано на рис. 29.4.

В меню Build (Сборка) среды Visual Studio выберем пункт Clean Solution (Очистить решение), чтобы очистить текущее содержимое папки \bin\Debug\Images, и повторно скомпилируем проект. Обновим окно Solution Explorer и удостоверимся в том, что данные в каталоге \bin\Debug\Images отсутствуют. При текущих параметрах сборки графические данные больше не копируются в выходную папку, а встраиваются в саму сборку.

Теперь необходимо модифицировать код для загрузки изображений, чтобы извлекать их из скомпилированной сборки:

```
private void MainWindow_OnLoaded(object sender, RoutedEventArgs e)
{
 try
 {
   _images.Add(new BitmapImage(new Uri(@"/Images/Deer.jpg", UriKind.Relative)));
   _images.Add(new BitmapImage(new Uri(@"/Images/Dogs.jpg", UriKind.Relative)));
   _images.Add(new BitmapImage(new Uri(@"/Images/Welcome.jpg", UriKind.Relative)));
    imageHolder.Source = _images[_currImage];
 }
```

```
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message);
    }
  }
```

В этом случае больше не придется определять путь установки и можно просто задавать ресурсы по именам, которые учитывают название исходного подкаталога. Также обратите внимание, что при создании объектов `Uri` указывается значение `Relative` перечисления `UriKind`. В данный момент исполняемая программа представляет собой автономную сущность, которая может быть запущена из любого местоположения на машине, т.к. все скомпилированные данные находятся внутри сборки. На рис. 29.5 показано готовое приложение в работе.

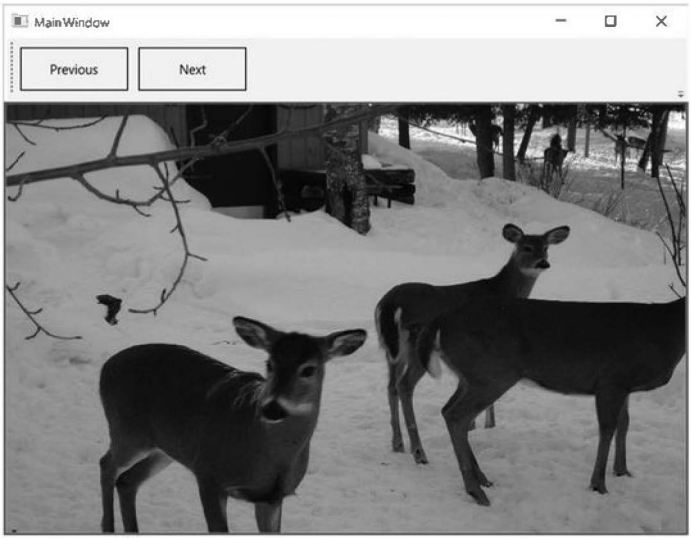


Рис. 29.5. Простое приложение для просмотра изображений

Исходный код. Проект `BinaryResourcesApp` доступен в подкаталоге `Chapter_29`.

Работа с объектными (логическими) ресурсами

При построении приложения WPF очень часто приходится определять большой объем разметки XAML для использования во многих местах окна или возможно в нескольких окнах или проектах. Например, предположим, что создана *безупречная* кисть с линейным градиентом, определение которой в разметке занимает 10 строк. Теперь эту кисть необходимо применить в качестве фонового цвета для каждого элемента `Button` в проекте, состоящем из 8 окон, т.е. всего получается 16 элементов `Button`.

Худшее, что можно было бы предпринять в такой ситуации — копировать и вставлять одну и ту же разметку XAML в каждый элемент управления `Button`. Очевидно, в итоге это могло бы стать настоящим кошмаром при сопровождении, т.к. всякий раз, когда нужно скорректировать внешний вид и поведение кисти, приходилось бы вносить изменения в многочисленные места.

К счастью, *объектные ресурсы* позволяют определить фрагмент разметки XAML, назначить ему имя и сохранить в подходящем словаре для использования в будущем. Подобно двоичным ресурсам объектные ресурсы часто компилируются в сборку, в которой они требуются. Однако при этом нет необходимости возиться со свойством Build Action. При условии, что разметка XAML помещена в корректное местоположение, компилятор позаботится обо всем остальном.

Взаимодействие с объектными ресурсами является крупной частью процесса разработки приложений WPF. Вы увидите, что объектные ресурсы могут быть намного сложнее, чем специальная кисть. Допускается определять анимацию на основе XAML, трехмерную визуализацию, специальный стиль элемента управления, шаблон данных, шаблон элемента управления и многое другое, упаковывая каждую сущность в многократно используемый ресурс.

Роль свойства `Resources`

Как уже упоминалось, для применения в приложении объектные ресурсы должны быть помещены в подходящий объект словаря. Каждый производный от `FrameworkElement` класс поддерживает свойство `Resources`, которое инкапсулирует объект `ResourceDictionary`, содержащий определенные объектные ресурсы. Объект `ResourceDictionary` может хранить элементы любого типа, потому что оперирует экземплярами `System.Object` и допускает манипуляции из разметки XAML или процедурного кода.

В WPF все элементы управления, элементы `Window`, `Page` (используемые при построении навигационных приложений или программ XBAP) и `UserControl` расширяют класс `FrameworkElement`, так что почти все виджеты предоставляют доступ к `ResourceDictionary`. Более того, класс `Application`, хотя и не расширяет `FrameworkElement`, но поддерживает свойство с идентичным именем `Resources`, которое предназначено для той же цели.

Определение ресурсов уровня окна

Чтобы приступить к исследованию роли объектных ресурсов, создадим в Visual Studio новый проект WPF Application по имени `ObjectResourcesApp` и заменим первоначальный элемент `Grid` горизонтально выровненным диспетчером компоновки `StackPanel`, внутри которого определим два элемента управления `Button`:

```
<Window x:Class="ObjectResourcesApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:ObjectResourcesApp"
    mc:Ignorable="d"
  Title="Fun with Object Resources" Height="350" Width="525">

  <StackPanel Orientation="Horizontal">
   <Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"/>
   <Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"/>
  </StackPanel>

</Window>
```

Выберем кнопку OK и установим в свойстве `Background` специальный тип кисти с применением интегрированного редактора кистей (который обсуждался в главе 28). Взгляните, как кисть была встроена внутрь области между дескрипторами `<Button>` и `</Button>`:

```
<Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20">
  <Button.Background>
    <RadialGradientBrush>
      <GradientStop Color="#FFC44EC4" Offset="0" />
      <GradientStop Color="#FF829CEB" Offset="1" />
      <GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>
  </Button.Background>
</Button>
```

Чтобы разрешить использовать эту кисть также и в кнопке Cancel (Отмена), область определения `<RadialGradientBrush>` должна быть расширена до словаря ресурсов родительского элемента. Например, если переместить `<RadialGradientBrush>` в `<StackPanel>`, то обе кнопки смогут применять одну и ту же кисть, т.к. они являются дочерними элементами того же самого диспетчера компоновки. Что еще лучше, кисть можно было бы упаковать в словарь ресурсов самого окна, в результате чего ее могли бы свободно использовать все аспекты содержимого окна (вложенные панели и т.д.).

Когда необходимо определить ресурс, для установки свойства `Resources` владельца применяется синтаксис "свойство-элемент". Кроме того, элементу ресурса назначается значение `x:Key`, которое будет использоваться другими частями окна для ссылки на объектный ресурс. Имейте в виду, что атрибуты `x:Key` и `x:Name` — не одно и то же! Атрибут `x:Name` позволяет получать доступ к объекту как к переменной-члену в файле кода, в то время как атрибут `x:Key` дает возможность ссылаться на элемент в словаре ресурсов.

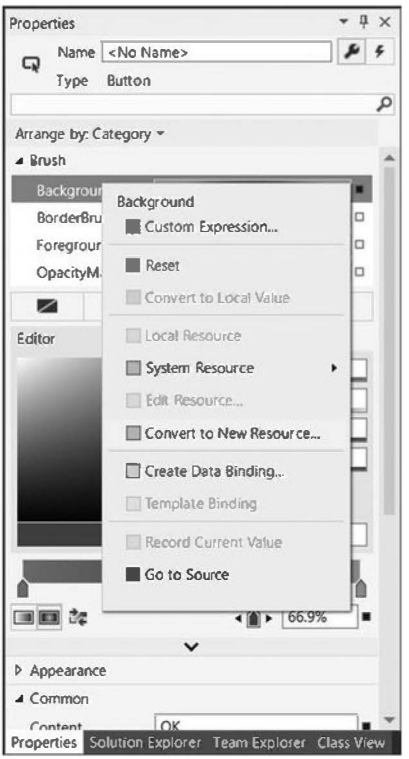


Рис. 29.6. Перемещение сложного объекта в контейнер ресурсов

Среда Visual Studio позволяет переместить ресурс на более высокий уровень с применением соответствующего окна Properties. Чтобы сделать это, сначала понадобится идентифицировать свойство, имеющее сложный объект, который необходимо упаковать в виде ресурса (в рассматриваемом примере это свойство `Background`). Рядом со свойством находится небольшой квадрат белого цвета, щелчок на котором приводит к открытию всплывающего меню. Выберем пункт Convert to New Resource (Преобразовать в новый ресурс), как продемонстрировано на рис. 29.6.

Будет запрошено имя ресурса (`myBrush`) и предложено указать, куда он должен быть помещен. Оставим отмеченным переключатель This document (Этот документ), который выбирается по умолчанию (рис. 29.7).

В результате разметка будет реструктурирована следующим образом:

```
<Window x:Class="ObjectResourcesApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

```
    xmlns:local="clr-namespace:ObjectResourcesApp"
    mc:Ignorable="d"
  Title="Fun with Object Resources" Height="350" Width="525">

  <Window.Resources>
    <RadialGradientBrush x:Key="myBrush">
      <GradientStop Color="#FFC44EC4" Offset="0" />
      <GradientStop Color="#FF829CEB" Offset="1" />
      <GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>
  </Window.Resources>

  <StackPanel Orientation="Horizontal">
    <Button Margin="25" Height="200" Width="200" Content="OK"
            FontSize="20" Background="{DynamicResource myBrush}"/>
    <Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"/>
  </StackPanel>
</Window>
```

Обратите внимание на новую область `<Window.Resources>`, **которая теперь содержит объект** `RadialGradientBrush`, **имеющий значение ключа** `myBrush`. **Новый ресурс создан как динамический (**`DynamicResource`**). Динамические ресурсы рассматриваются позже, а пока изменим тип ресурса на статический (**`StaticResource`**):**

```
<Button Margin="25" Height="200" Width="200" Content="OK"
        FontSize="20" Background="{StaticResource myBrush}"/>
<Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"/>
```

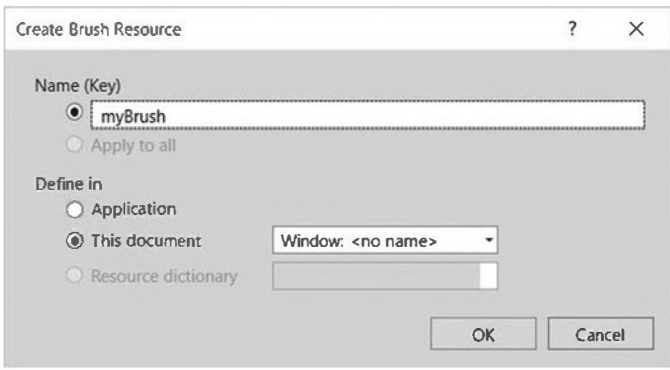


Рис. 29.7. Назначение имени объектному ресурсу

Расширение разметки {StaticResource}

Другое изменение, происходящее при извлечении объектного ресурса, связано с тем, что свойство, которое было целью извлечения (опять-таки, `Background`**), теперь использует расширение разметки** `{StaticResource}`. **Как видите, имя ключа указано в качестве аргумента. Теперь если в кнопке Cancel предпочтительно применять ту же самую кисть для рисования фона, то это делается легко. Или же если кнопка Cancel имеет какое-то сложное содержимое, то в любом ее внутреннем элементе можно также использовать ресурс уровня окна, как в свойстве** `Fill` **элемента** `Ellipse`:

```
<StackPanel Orientation="Horizontal">
  <Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"
          Background="{StaticResource myBrush}">
  </Button>
```

```
  <Button Margin="25" Height="200" Width="200" FontSize="20">
    <StackPanel>
      <Label HorizontalAlignment="Center" Content= "No Way!"/>
      <Ellipse Height="100" Width="100" Fill="{StaticResource myBrush}"/>
    </StackPanel>
  </Button>
</StackPanel>
```

Расширение разметки {DynamicResource}

При подключении к ресурсу, поддерживающему ключи, в свойстве также может применяться расширение разметки `{DynamicResource}`. Чтобы выяснить разницу, назначим кнопке OK имя `btnOK` и обработаем ее событие `Click`. Внутри обработчика события с помощью свойства `Resources` получим специальную кисть и внесем в нее изменение:

```
private void btnOK_Click(object sender, RoutedEventArgs e)
{
  // Получить кисть и внести в нее изменение.
  var b = (RadialGradientBrush)Resources["myBrush"];
  b.GradientStops[1] = new GradientStop(Colors.Black, 0.0);
}
```

На заметку! Здесь для поиска ресурса по имени используется индексатор `Resources`. Тем не менее, имейте в виду, что если ресурс найти не удастся, то будет сгенерировано исключение времени выполнения. Можно также применять метод `TryFindResource()`, который не приводит к генерации исключения, а просто возвращает `null`, когда указанный ресурс не найден.

После запуска приложения и щелчка на кнопке OK можно будет заметить, что изменение кисти учтено, а каждая кнопка обновляется для визуализации модифицированной кисти. Однако что если полностью изменить тип кисти, указанной ключом `myBrush`? Например:

```
private void btnOK_Click(object sender, RoutedEventArgs e)
{
  // Поместить в ячейку myBrush совершенно новую кисть.
  Resources["myBrush"] = new SolidColorBrush(Colors.Red);
}
```

На этот раз после щелчка на кнопке ожидаемых обновлений не происходит. Причина в том, что расширение разметки `{StaticResource}` применяет ресурс только однажды и остается "подключенным" к исходному объекту на протяжении времени жизни приложения. Тем не менее, если изменить в разметке все вхождения `{StaticResource}` на `{DynamicResource}`, то окажется, что специальная кисть будет заменена кистью со сплошным красным цветом.

По существу расширение разметки `{DynamicResource}` способно обнаруживать замену лежащего в основе объекта, указанного посредством ключа, новым объектом. Как и можно было предположить, это требует дополнительной инфраструктуры времени выполнения, так что обычно следует использовать `{StaticResource}`, если только не планируется заменять объектный ресурс другим объектом во время выполнения с уведомлением всех элементов, которые задействуют этот ресурс.

Ресурсы уровня приложения

Когда в словаре ресурсов окна имеются объектные ресурсы, их могут потреблять все элементы этого окна, но не другие окна приложения. Назначим кнопке Cancel имя

btnCancel и обработаем событие Click. Добавим в текущий проект новое окно (в файле TestWindow.xaml), содержащее единственную кнопку Button, щелчок на которой приводит к закрытию окна:

```
public partial class TestWindow : Window
{
  public TestWindow()
  {
    InitializeComponent();
  }
  private void btnClose_Click(object sender, RoutedEventArgs e)
  {
    Close();
  }
}
```

В обработчике Click кнопки Cancel первого окна загрузим и отобразим новое окно:

```
private void btnCancel_Click(object sender, RoutedEventArgs e)
{
  var w = new TestWindow();
  w.Owner = this;
  w.WindowStartupLocation =
     WindowStartupLocation.CenterOwner;
  w.ShowDialog();
}
```

В текущий момент в новом окне кисть myBrush не может быть применена, т.к. она не находится внутри корректной "области действия". Решение заключается в определении объектного ресурса на уровне приложения, а не на уровне конкретного окна. В Visual Studio не предусмотрено какого-либо способа автоматизировать такое действие, поэтому просто вырежем имеющееся определение объекта кисти из области <Windows.Resource> и поместим его в область <Application.Resources> файла App.xaml:

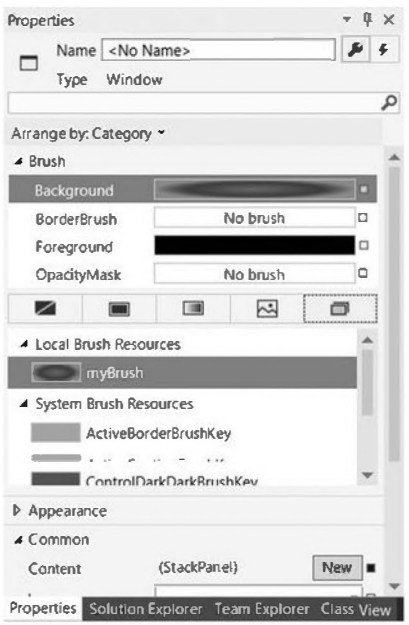


Рис. 29.8. Применение ресурсов уровня приложения

```
<Application x:Class="ObjectResourcesApp.App"
  xmlns="http://schemas.microsoft.com/
winfx/2006/xaml/presentation"
  xmlns:x=http://schemas.microsoft.com/
winfx/2006/xaml
 xmlns:local="clr-namespace:ObjectResourcesApp"
 StartupUri="MainWindow.xaml">
 <Application.Resources>
    <RadialGradientBrush x:Key="myBrush">
<GradientStop Color="#FFC44EC4" Offset="0" />
<GradientStop Color="#FF829CEB" Offset="1" />
<GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>
  </Application.Resources>
</Application>
```

Теперь в окне TestWindow для рисования фона может использоваться та же самая кисть. Чтобы найти свойство Background нового окна, понадобится щелкнуть на вкладке Brush resources (Ресурсы кисти) справа для просмотра ресурсов уровня приложения (рис. 29.8).

Определение объединенных словарей ресурсов

Ресурсы уровня приложения являются хорошей отправной точкой, но как быть, если необходимо определить набор сложных (или не очень) ресурсов, которые должны многократно использоваться во множестве проектов WPF? В таком случае понадобится определить то, что известно как *объединенный словарь ресурсов*. Он представляет собой всего лишь файл `.xaml`, который содержит коллекцию объектных ресурсов. Единственный проект может иметь любое требуемое количество таких файлов (один для кистей, один для анимации и т.д.), каждый из которых может быть добавлен в диалоговом окне Add New Item (Добавление нового элемента), доступном через меню Project (рис. 29.9).

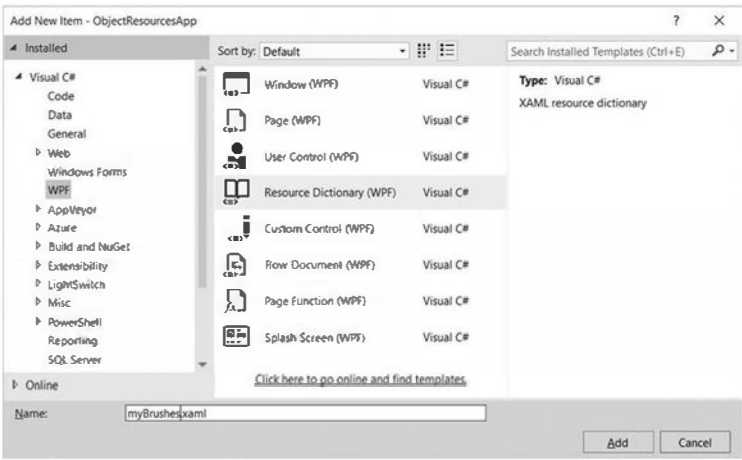


Рис. 29.9. Вставка нового объединенного словаря ресурсов

Вырежем текущие ресурсы из области определения `Application.Resources` в новом файле `MyBrushes.xaml` и перенесем их в словарь:

```
<ResourceDictionary
  xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
  xmlns:local="clr-namespace:ObjectResourcesApp"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <RadialGradientBrush x:Key="myBrush">
    <GradientStop Color="#FFC44EC4" Offset="0" />
    <GradientStop Color="#FF829CEB" Offset="1" />
    <GradientStop Color="#FF793879" Offset="0.669" />
  </RadialGradientBrush>

</ResourceDictionary>
```

Несмотря на то что данный словарь ресурсов является частью проекта, будут возникать ошибки времени выполнения. Дело в том, что все словари ресурсов должны быть объединены (обычно на уровне приложения) в единый словарь ресурсов. Для этого применяется следующий формат в файле `App.xaml` (обратите внимание, что множество словарей ресурсов объединяются за счет добавления элементов `<ResourceDictionary>` в область `<ResourceDictionary.MergedDictionaries>`):

```
<Application x:Class="ObjectResourcesApp.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
  xmlns:local="clr-namespace:ObjectResourcesApp"
  StartupUri="MainWindow.xaml">
```

```
    <!-- Взять логические ресурсы из файла MyBrushes.xaml -->
    <Application.Resources>
      <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
          <ResourceDictionary Source = "MyBrushes.xaml"/>
        </ResourceDictionary.MergedDictionaries>
      </ResourceDictionary>
    </Application.Resources>
</Application>
```

Определение сборки, включающей только ресурсы

И последнее по порядку, но не по важности: существует возможность создания библиотек классов, которые не содержат ничего кроме словарей объектных ресурсов. Это может быть полезно в ситуации, когда определен набор тем, которые должны применяться на уровне машины. Объектные ресурсы можно упаковать в выделенную сборку, после чего приложения, которые нуждаются в их использовании, будут загружать ресурсы в память.

Самый легкий способ построения сборки из одних ресурсов предусматривает создание проекта WPF User Control Library (Библиотека пользовательских элементов управления WPF). Добавим такой проект (по имени `MyBrushesLibrary`) к текущему решению, выбрав пункт меню Add⇨New Project (Добавить⇨Новый проект) в Visual Studio (рис. 29.10).

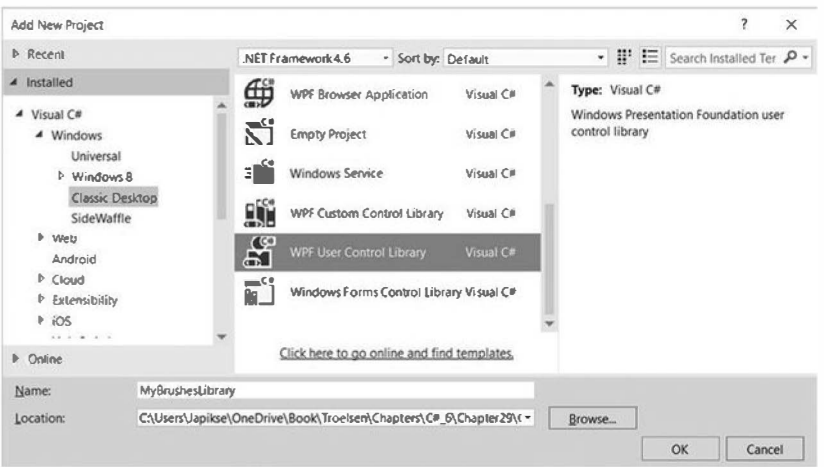


Рис. 29.10. Добавление проекта WPF User Control Library в качестве начальной точки для построения библиотеки из одних ресурсов

Теперь удалим файл `UserControl1.xaml` из проекта (единственные элементы, которые действительно необходимы — это ссылаемые сборки WPF). Перетащим файл `MyBrushes.xaml` в проект `MyBrushesLibrary` и удалим его из проекта `ObjectResourcesApp`. Наконец, откроем файл `MyBrushes.xaml` в проекте `MyBrushesLibrary` и изменим пространство имен `x:local` на `clr-namespace:MyBrushesLibrary`. Окно Solution Explorer должно теперь выглядеть так, как показано на рис. 29.11.

Скомпилируем проект WPF User Control Library. Затем установим ссылку на эту библиотеку из проекта `ObjectResourcesApp` с применением диалогового окна Add Reference (Добавление ссылки). Теперь объединим эти двоичные ресурсы со словарем ресурсов уровня приложения из проекта `ObjectResourcesApp`. Однако такое действие требует использования довольно забавного синтаксиса:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
<!-- Синтаксис выглядит как /ИмяСборки;Component/ИмяФайлаXamlВСборке.xaml -->
      <ResourceDictionary Source = "/MyBrushesLibrary;Component/MyBrushes.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

Имейте в виду, что данная строка восприимчива к пробелам. Если возле двоеточия или косых черт будут присутствовать лишние пробелы, то возникнут ошибки времени выполнения. Первая часть строки представляет собой дружественное имя внешней библиотеки (без файлового расширения). После двоеточия идет слово `Component`, а за ним имя скомпилированного двоичного ресурса, которое будет идентичным имени исходного словаря ресурсов XAML.

Итак, знакомство с системой управления ресурсами WPF завершено. Описанные здесь приемы придется часто применять в большинстве разрабатываемых приложений. Теперь давайте займемся исследованием встроенного API-интерфейса анимации WPF.

Исходный код. Проект `ObjectResourceApp` доступен в подкаталоге `Chapter_29`.

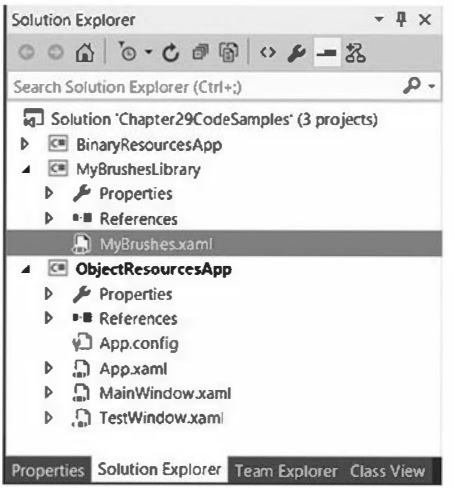


Рис. 29.11. Перемещение файла `MyBrushes.xaml` в новый проект библиотеки

Службы анимации WPF

В дополнение к службам графической визуализации, которые рассматривались в главе 28, инфраструктура WPF предлагает API-интерфейс для поддержки служб анимации. Встретив термин *анимация*, многим на ум приходит вращающийся логотип компании, последовательность сменяющих друг друга изображений (для создания иллюзии движения), подпрыгивающий текст на экране или программа специфического типа вроде видеоигры или мультимедиа-приложения.

Наряду с тем, что API-интерфейсы анимации WPF определенно могли бы использоваться для упомянутых выше целей, анимация может применяться всякий раз, когда приложению необходимо придать особый стиль. Например, можно было бы построить анимацию для кнопки на экране, чтобы она слегка увеличивалась, когда курсор мыши находится внутри ее границ (и возвращалась к прежним размерам, когда курсор покинул границы). Или же можно было бы предусмотреть анимацию для окна, обеспечив его закрытие с использованием определенного визуального эффекта, такого как постепенное исчезновение до полной прозрачности. В действительности поддержка анимации WPF может применяться в приложениях любого рода (бизнес-приложениях, мультимедиа-программах, видеоиграх и т.д.), когда у пользователя нужно создать более привлекательное впечатление.

Как и со многими другими аспектами WPF, с построением анимации не связано ничего нового. Единственной особенностью является то, что в отличие от других API-интерфейсов, которые вы могли использовать в прошлом (включая Windows Forms), разработчики не обязаны создавать необходимую инфраструктуру вручную.

В среде WPF отсутствует потребность в предварительном создании фоновых потоков или таймеров, применяемых для продвижения вперед анимационной последовательности, в определении специальных типов для представления анимации, в очистке и перерисовывании изображений либо в реализации утомительных математических вычислений. Подобно другим аспектам WPF анимацию можно строить целиком в разметке XAML, целиком в коде C# либо с использованием комбинации того и другого.

На заметку! В среде Visual Studio отсутствует поддержка создания анимации посредством каких-либо графических инструментов. Для этого в Visual Studio придется вводить разметку XAML вручную. Тем не менее, поставляемый в составе Visual Studio 2015 продукт Blend for Visual Studio на самом деле имеет встроенный редактор анимации, который способен существенно упростить решение задач.

Роль классов анимации

Чтобы разобраться в поддержке анимации WPF, необходимо начать с рассмотрения классов анимации из пространства имен `System.Windows.Media.Animation` сборки `PresentationCore.dll`. Здесь вы найдете свыше 100 разных классов, которые содержат в своем имени слово `Animation`.

Все эти классы могут быть отнесены к одной из трех обширных категорий. Во-первых, любой класс, который следует соглашению об именовании вида *ТипДанных*`Animation` (`ByteAnimation`, `ColorAnimation`, `DoubleAnimation`, `Int32Animation` и т.д.), позволяет работать с анимацией линейной интерполяцией. Она обеспечивает плавное изменение значения во времени от начального к конечному.

Во-вторых, классы, следующие соглашению об именовании вида *ТипДанных* `AnimationUsingKeyFrames` (`StringAnimationUsingKeyFrames`, `DoubleAnimation UsingKeyFrames`, `PointAnimationUsingKeyFrames` и т.д.), представляют анимацию ключевыми кадрами, которая позволяет проходить в цикле по набору определенных значений за указанный период времени. Например, ключевые кадры можно применять для изменения надписи на кнопке, проходя в цикле по последовательности индивидуальных символов.

В-третьих, классы, которые следуют соглашению об именовании вида *ТипДанных* `AnimationUsingPath` (`DoubleAnimationUsingPath`, `PointAnimationUsingPath` и т.п.), представляют анимацию на основе пути, позволяющую перемещать объекты по определенному пути. Например, в приложении глобального позиционирования (GPS) анимацию на основе пути можно использовать для перемещения элемента по кратчайшему маршруту к месту, указанному пользователем.

Вполне очевидно, эти классы *не* применяются для того, чтобы каким-то образом предоставить анимационную последовательность напрямую переменной определенного типа данных (в конце концов, как можно было бы выполнить анимацию значения 9, используя объект `Int32Animation`?).

Возьмем, к примеру, свойства `Height` и `Width` типа `Label`, которые являются свойствами зависимости, упаковывающими значение `double`. Чтобы определить анимацию, которая будет увеличивать высоту метки с течением времени, можно подключить объект `DoubleAnimation` к свойству `Height` и позволить WPF позаботиться о деталях выполнения действительной анимации. Или вот другой пример: если требуется реализовать переход цвета кисти от зеленого до желтого в течение 5 секунд, то это можно сделать с применением типа `ColorAnimation`.

Чтобы быть предельно ясным, классы `Animation` могут подключаться к любому *свойству зависимости* заданного объекта, которое имеет соответствующий тип. Как объяснялось в главе 27, свойства зависимости — это специальная форма свойств, которую требуют многие службы WPF, включая анимацию, привязку данных и стили.

По соглашению свойство зависимости определяется как статическое, доступное только для чтения поле класса, имя которого образуется добавлением слова `Property` к нормальному имени свойства. Например, для обращения к свойству зависимости для свойства `Height` класса `Button` в коде будет использоваться `Button.HeightProperty`.

Свойства `To`, `From` и `By`

Во всех классах `Animation` определены следующие ключевые свойства, которые управляют начальным и конечным значениями, применяемыми для выполнения анимации:

- `To` — представляет конечное значение анимации;
- `From` — представляет начальное значение анимации;
- `By` — представляет общую величину, на которую анимация изменяет начальное значение.

Несмотря на тот факт, что все классы поддерживают свойства `To`, `From` и `By`, они не получают их через виртуальные члены базового класса. Причина в том, что лежащие в основе типы, упакованные внутри указанных свойств, варьируются в широких пределах (целые числа, цвета, объекты `Thickness` и т.д.), и представление всех возможностей через единственный базовый класс привело бы к очень сложным кодовым конструкциям.

В связи с этим может также возникнуть вопрос: почему не использовались обобщения .NET для определения единственного обобщенного класса анимации с одиночным параметром типа (скажем, `Animate<T>`)? Опять-таки, поскольку существует огромное количество типов данных (цвета, векторы, целые числа, строки и т.д.), применяемых для анимации свойств зависимости, это решение оказалось бы не настолько ясным, как можно было бы ожидать (не говоря уже о том, что XAML обеспечивает лишь ограниченную поддержку обобщенных типов).

Роль базового класса `Timeline`

Хотя для определения виртуальных свойств `To`, `From` и `By` не использовался единственный базовый класс, классы `Animation` все же разделяют общий базовый класс — `System.Windows.Media.Animation.Timeline`. Данный тип предоставляет набор дополнительных свойств, которые управляют темпом продвижения анимации (табл. 29.1).

Таблица 29.1. Основные свойства базового класса `Timeline`

Свойства	Описание
`AccelerationRatio`, `DecelerationRatio`, `SpeedRatio`	Эти свойства применяются для управления общим темпом анимационной последовательности
`AutoReverse`	Это свойство получает или устанавливает значение, которое указывает, должна ли временная шкала воспроизводиться в обратном направлении после завершения итерации вперед (стандартным значением является `false`)
`BeginTime`	Это свойство получает или устанавливает время запуска временной шкалы. Стандартным значением является `0`, что запускает анимацию немедленно
`Duration`	Это свойство позволяет устанавливать продолжительность воспроизведения временной шкалы
`FileBehavior`, `RepeatBehavior`	Эти свойства используются для управления тем, что должно произойти после завершения временной шкалы (повторение анимации, ничего и т.д.)

Реализация анимации в коде C#

Мы построим окно, содержащее элемент Button, который обладает довольно странным поведением: когда на него наводится курсор мыши, он вращается вокруг своего левого верхнего угла. Начнем с создания в Visual Studio нового проекта WPF Application по имени SpinningButtonAnimationApp. Модифицируем начальную разметку, как показано ниже (обратите внимание на обработку события MouseEnter кнопки):

```
<Window x:Class="SpinningButtonAnimationApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:SpinningButtonanimationApp"
    mc:Ignorable="d"
  Title="Animations in C# code" Height="350"
  Width="525" WindowStartupLocation="CenterScreen">
  <Grid>
    <Button x:Name="btnSpinner" Height="50" Width="100" Content="I Spin!"
            MouseEnter="btnSpinner_MouseEnter"/>
  </Grid>
</Window>
```

Теперь импортируем пространство имен System.Windows.Animation и добавим в файл C# следующий код:

```
public partial class MainWindow : Window
{
  private bool _isSpinning = false;

  private void btnSpinner_MouseEnter(object sender, MouseEventArgs e)
  {
    if (!_isSpinning)
    {
      _isSpinning = true;
      //Создать объект DoubleAnimation и зарегистрировать с событием Completed.
      var dblAnim = new DoubleAnimation();
      dblAnim.Completed += (o, s) => { _isSpinning = false; };

      // Установить начальное и конечное значения.
      dblAnim.From = 0;
      dblAnim.To = 360;

      // Создать объект RotateTransform и присвоить
      // его свойству RenderTransform кнопки.
      var rt = new RotateTransform();
      btnSpinner.RenderTransform = rt;

      // Выполнить анимацию объекта RotateTransform.
      rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
    }
  }
}
```

Первая крупная задача метода btnSpinner_MouseEnter() связана с конфигурированием объекта DoubleAnimation, который будет начинать со значения 0 и заканчивать значением 360. Обратите внимание, что для этого объекта также обрабатывается событие Completed, где переключается булевская переменная уровня класса, которая применяется для того, чтобы выполняющаяся анимация не была сброшена в начало.

Затем создается объект RotateTransform, который подключается к свойству RenderTransform элемента управления Button (btnSpinner). И, наконец, объект RenderTransform информируется о начале анимации его свойства Angle с использованием объекта DoubleAnimation. Реализация анимации в коде обычно осуществляется путем вызова метода BeginAnimation() и передачи ему лежащего в основе свойства зависимости, к которому необходимо применить анимацию (вспомните, что по соглашению это статическое поле класса), и связанного объекта анимации.

Давайте добавим в программу еще одну анимацию, которая заставит кнопку после щелчка плавно становиться невидимой. Для начала создадим обработчик события Click кнопки btnSpinner и поместим в него приведенный ниже код:

```
private void btnSpinner_Click(object sender, RoutedEventArgs e)
{
  var dblAnim = new DoubleAnimation
  {
    From = 1.0,
    To = 0.0
  };
  btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}
```

В коде обработчика события изменяется свойство Opacity, чтобы постепенно скрыть кнопку из виду. Однако в настоящий момент это затруднительно, потому что кнопка вращается слишком быстро. Как можно управлять ходом анимации? Ответ на этот вопрос ищите ниже.

Управление темпом анимации

По умолчанию анимация будет занимать приблизительно одну секунду для перехода между значениями, которые присвоены свойствам From и To. Следовательно, кнопка располагает одной секундой, чтобы повернуться на 360 градусов, и в то же время в течение одной секунды она постепенно скроется из виду (после щелчка на ней).

Определить другой период времени для перехода анимации можно посредством свойства Duration объекта анимации, которому присваивается объект Duration. Обычно промежуток времени устанавливается путем передачи объекта TimeSpan конструктору класса Duration. Взгляните на показанное далее изменение, в результате которого кнопке будет выделено четыре секунды на вращение:

```
private void btnSpinner_MouseEnter(object sender, MouseEventArgs e)
{
  if (!_isSpinning)
  {
    _isSpinning = true;

    // Создать объект DoubleAnimation и зарегистрировать с событием Completed.
    var dblAnim = new DoubleAnimation();
    dblAnim.Completed += (o, s) => { _isSpinning = false; };

    // На завершение поворота кнопке отводится четыре секунды.
    dblAnim.Duration = new Duration(TimeSpan.FromSeconds(4));
    ...
  }
}
```

Благодаря такой модификации появится реальный шанс щелкнуть на кнопке во время ее вращения, после чего она плавно исчезнет.

На заметку! Свойство `BeginTime` класса `Animation` также принимает объект `TimeSpan`. Вспомните, что это свойство можно устанавливать для указания времени ожидания перед запуском анимационной последовательности.

Запуск в обратном порядке и циклическое выполнение анимации

За счет установки в `true` свойства `AutoReverse` объектам `Animation` указывается о необходимости запуска анимации в обратном порядке по ее завершении. Например, если необходимо, чтобы кнопка снова стала видимой после исчезновения, можно написать следующий код:

```
private void btnSpinner_Click(object sender, RoutedEventArgs e)
{
  DoubleAnimation dblAnim = new DoubleAnimation
  {
    From = 1.0,
    To = 0.0
  };
  // По завершении запустить в обратном порядке.
  dblAnim.AutoReverse = true;
  btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}
```

Если нужно, чтобы анимация повторялась несколько раз (или никогда не прекращалась), то для этого можно использовать свойство `RepeatBehavior`, общее для всех классов `Animation`. Передавая конструктору простое числовое значение, можно указать жестко закодированное количество повторений. С другой стороны, если передать конструктору объект `TimeSpan`, то можно задать время, в течение которого анимация должна повторяться. Наконец, чтобы выполнять анимацию бесконечно, свойство `RepeatBehavior` можно установить в `RepeatBehavior.Forever`. Рассмотрим следующие способы изменения поведения повтора одного из двух объектов `DoubleAnimation`, применяемых в примере:

```
// Повторять бесконечно.
dblAnim.RepeatBehavior = RepeatBehavior.Forever;

// Повторять три раза.
dblAnim.RepeatBehavior = new RepeatBehavior(3);

// Повторять в течение 30 секунд.
dblAnim.RepeatBehavior = new RepeatBehavior(TimeSpan.FromSeconds(30));
```

На этом завершается исследование приемов добавления анимации к аспектам какого-то объекта с использованием кода C# и API-интерфейса анимации WPF. Теперь посмотрим, как сделать то же самое с помощью разметки XAML.

Исходный код. Проект `SpinningButtonAnimationApp` доступен в подкаталоге `Chapter_29`.

Реализация анимации в разметке XAML

Реализация анимации в разметке подобна ее реализации в коде, во всяком случае, для простых анимационных последовательностей. Когда необходимо создать более сложную анимацию, которая включает изменение значений множества свойств одновременно, объем разметки может заметно увеличиться. Даже в случае применения какого-то

инструмента для генерирования анимации, основанной на разметке XAML, важно знать основы представления анимации в XAML, поскольку это облегчит задачу модификации и настройки сгенерированного инструментом содержимого.

На заметку! В подкаталоге `XamlAnimations` внутри `Chapter_29` есть несколько файлов XAML. Скопируйте содержимое этих файлов разметки в специальный редактор XAML или в редактор Kaxaml, чтобы просмотреть результаты.

Большей частью создание анимации подобно всему тому, что вы уже видели: по-прежнему производится конфигурирование объекта `Animation`, который затем ассоциируется со свойством объекта. Тем не менее, крупное отличие связано с тем, что разметка XAML не является дружественной к вызовам методов. В результате вместо вызова `BeginAnimation()` используется *раскадровка* как промежуточный уровень.

Давайте рассмотрим полный пример анимации, определенной в терминах XAML, и подробно ее проанализируем. Следующее определение XAML будет отображать окно, содержащее единственную метку. После того как объект `Label` загрузился в память, он начинает анимационную последовательность, во время которой размер шрифта увеличивается от 12 до 100 точек за период в четыре секунды. Анимация будет повторяться столько времени, сколько объект остается загруженным в память. Эта разметка находится в файле `GrowLabelFont.xaml`, поэтому его содержимое необходимо скопировать в приложение `MyXamlPad.exe` (или в редактор Kaxaml) и понаблюдать за поведением.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="200" Width="600" WindowStartupLocation="CenterScreen"
Title="Growing Label Font!">
  <StackPanel>
    <Label Content = "Interesting...">
      <Label.Triggers>
        <EventTrigger RoutedEvent = "Label.Loaded">
          <EventTrigger.Actions>
            <BeginStoryboard>
              <Storyboard TargetProperty = "FontSize">
                <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                  RepeatBehavior = "Forever"/>
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger.Actions>
        </EventTrigger>
      </Label.Triggers>
    </Label>
  </StackPanel>
</Window>
```

А теперь подробно разберем приведенный пример.

Роль раскадровок

Двигаясь от самого глубоко вложенного элемента наружу, первым мы встречаем элемент `<DoubleAnimation>`, обращающийся к тем же самым свойствам, которые устанавливались в процедурном коде (`From`, `To`, `Duration` и `RepeatBehavior`):

```
<DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                 RepeatBehavior = "Forever"/>
```

Как упоминалось ранее, элементы Animation помещаются внутрь элемента <Storyboard>, применяемого для отображения объекта анимации на заданное свойство родительского типа через свойство TargetProperty, которым в этом случае является FontSize. Элемент <Storyboard> всегда находится внутри родительского элемента по имени <BeginStoryboard>:

```
<BeginStoryboard>
  <Storyboard TargetProperty = "FontSize">
    <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                     RepeatBehavior = "Forever"/>
  </Storyboard>
</BeginStoryboard>
```

Роль триггеров событий

После того как элемент <BeginStoryboard> определен, должно быть указано действие какого-то вида, которое приведет к запуску анимации. Инфраструктура WPF предлагает несколько разных способов реагирования на условия времени выполнения в разметке, один из которых называется *триггером*. С высокоуровневой точки зрения триггер можно считать способом реагирования на событие в разметке XAML без необходимости в написании процедурного кода.

Обычно когда реализуется ответ на событие в коде C#, пишется специальный код, который будет выполнен при поступлении события. Однако триггер — это всего лишь способ получить уведомление о том, что некоторое событие произошло (загрузка элемента в память, наведение на него курсора мыши, получение им фокуса и т.д.).

Получив уведомление о появлении события, можно запускать раскадровку. В данном примере мы реагируем на факт загрузки элемента Label в память. Поскольку нас интересует событие Loaded элемента Label, элемент <EventTrigger> помещается в коллекцию триггеров элемента Label:

```
<Label Content = "Interesting...">
  <Label.Triggers>
    <EventTrigger RoutedEvent = "Label.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "FontSize">
            <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                             RepeatBehavior = "Forever"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Label.Triggers>
</Label>
```

Рассмотрим еще один пример определения анимации в XAML, на этот раз анимации ключевыми кадрами.

Анимация с использованием дискретных ключевых кадров

В отличие от объектов анимации линейной интерполяцией, обеспечивающих только перемещение между начальной и конечной точками, объекты анимации *ключевыми кадрами* позволяют создавать коллекции специальных значений, которые должны случаться в определенные моменты времени.

Чтобы проиллюстрировать применение типа дискретного ключевого кадра, предположим, что необходимо построить элемент управления Button, который выполняет

анимацию своего содержимого так, что на протяжении трех секунд появляется значение OK! по одному символу за раз. Показанная далее разметка находится в файле StringAnimation.xaml. Ее можно скопировать в программу MyXamlPad.exe (или в редактор Kaxaml) и просмотреть результаты:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="100" Width="300"
  WindowStartupLocation="CenterScreen" Title="Animate String Data!">
  <StackPanel>
    <Button Name="myButton" Height="40"
            FontSize="16pt" FontFamily="Verdana" Width = "100">
     <Button.Triggers>
       <EventTrigger RoutedEvent="Button.Loaded">
         <BeginStoryboard>
           <Storyboard>
             <StringAnimationUsingKeyFrames RepeatBehavior = "Forever"
               Storyboard.TargetProperty="Content"
               Duration="0:0:3">
               <DiscreteStringKeyFrame Value="" KeyTime="0:0:0" />
               <DiscreteStringKeyFrame Value="O" KeyTime="0:0:1" />
               <DiscreteStringKeyFrame Value="OK" KeyTime="0:0:1.5" />
               <DiscreteStringKeyFrame Value="OK!" KeyTime="0:0:2" />
             </StringAnimationUsingKeyFrames>
           </Storyboard>
         </BeginStoryboard>
       </EventTrigger>
     </Button.Triggers>
    </Button>
  </StackPanel>
</Window>
```

Первым делом обратите внимание, что для кнопки определяется триггер события, который обеспечивает запуск раскадровки при загрузке кнопки в память. Класс StringAnimationUsingKeyFrames отвечает за изменение содержимого кнопки через значение Storyboard.TargetProperty.

Внутри элемента <StringAnimationUsingKeyFrames> определены четыре элемента DiscreteStringKeyFrame, которые изменяют свойство Content на протяжении двух секунд (длительность, установленная объектом StringAnimationUsingKeyFrames, составляет в сумме три секунды, поэтому между финальным символом ! и следующим появлением O будет заметна небольшая пауза).

Теперь, когда вы получили некоторое представление о том, как строятся анимации в коде C# и разметке XAML, давайте уделим внимание роли стилей WPF, которые интенсивно задействуют графику, объектные ресурсы и анимацию.

Исходный код. Несвязанные файлы XAML доступны в подкаталоге XamlAnimations внутри Chapter_29.

Роль стилей WPF

При построении пользовательского интерфейса приложения WPF нередко требуется обеспечить общий вид и поведение для целого семейства элементов управления. Например, может понадобиться сделать так, чтобы все типы кнопок имели ту же самую высоту, ширину, цвет и размер шрифта для своего строкового содержимого. Хотя решить

задачу можно было бы установкой идентичных значений в индивидуальных свойствах, такой подход затрудняет внесение изменений, потому что при каждом изменении придется переустанавливать один и тот же набор свойств во множестве объектов.

К счастью, инфраструктура WPF предлагает простой способ ограничения внешнего вида и поведения связанных элементов управления с использованием *стилей*. Выражаясь просто, стиль WPF — это объект, который поддерживает коллекцию пар "свойство-значение". С точки зрения программирования отдельный стиль представляется с помощью класса `System.Windows.Style`. Класс `Style` имеет свойство по имени `Setters`, которое открывает доступ к строго типизированной коллекции объектов `Setter`. Именно объект `Setter` обеспечивает возможность определения пар "свойство-значение".

В дополнение к коллекции `Setters` класс `Style` также определяет несколько других важных членов, которые позволяют встраивать триггеры, ограничивать место применения стиля и даже создавать новый стиль на основе существующего (воспринимайте это как "наследование стилей"). Ниже перечислены наиболее важные члены класса `Style`:

- `Triggers` — открывает доступ к коллекции объектов триггеров, которая делает возможной фиксацию условий возникновения разнообразных событий в стиле;
- `BasedOn` — разрешает строить новый стиль на основе существующего;
- `TargetType` — позволяет ограничивать место применения стиля.

Определение и применение стиля

Почти в каждом случае объект `Style` упаковывается как объектный ресурс. Подобно любому объектному ресурсу его можно упаковывать на уровне окна или на уровне приложения, а также внутри выделенного словаря ресурсов (что замечательно, поскольку делает объект `Style` легко доступным во всех местах приложения). Вспомните, что цель заключается в определении объекта `Style`, который наполняет (минимум) коллекцию `Setters` набором пар "свойство-значение".

Создадим в Visual Studio новый проект WPF Application по имени `WpfStyles`. Давайте построим стиль, который фиксирует базовые характеристики шрифта элемента управления в нашем приложении. Откроем файл `App.xaml` и определим следующий именованный стиль:

```
<Application x:Class="WpfStyles.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WpfStyles"
    mc:Ignorable="d"
  StartupUri="MainWindow.xaml">

  <Application.Resources>
    <Style x:Key ="BasicControlStyle">
      <Setter Property = "Control.FontSize" Value ="14"/>
      <Setter Property = "Control.Height" Value = "40"/>
      <Setter Property = "Control.Cursor" Value = "Hand"/>
    </Style>
  </Application.Resources>
</Application>
```

Обратите внимание, что объект `BasicControlStyle` добавляет во внутреннюю коллекцию три объекта `Setter`. Теперь применим этот стиль к нескольким элементам управления в главном окне. Из-за того, что этот стиль является объектным ресурсом,

элементы управления, которым он необходим, по-прежнему должны использовать расширение разметки {StackResource} или {DynamicResource} для нахождения стиля. (Как вам уже известно из обсуждения статических и динамических ресурсов, статические ресурсы чуть более эффективны, но ограничены в своем обновлении, поэтому выбирайте тот вид ресурса, который лучше подходит в существующих обстоятельствах.) Когда они находят стиль, то устанавливают элемент ресурса в идентично именованное свойство Style. Взгляните на показанное ниже определение элемента <Window>:

```
<Window x:Class="WpfStyles.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WpfStyles"
    mc:Ignorable="d"
  Title="A Window with Style!" Height="229"
  Width="525" WindowStartupLocation="CenterScreen">
  <StackPanel>
    <Label x:Name="lblInfo" Content="This style is boring..."
          Style="{StaticResource BasicControlStyle}" Width="150"/>
    <Button x:Name="btnTestButton" Content="Yes, but we are reusing settings!"
           Style="{StaticResource BasicControlStyle}" Width="250"/>
  </StackPanel>
</Window>
```

После запуска приложения обнаружится, что оба элемента управления поддерживают тот же самый курсор, высоту и размер шрифта.

Переопределение настроек стиля

Здесь определены элементы Button и Label, которые подчиняются ограничениям, накладываемым нашим стилем. Разумеется, после применения стиля к элементу управления вполне допустимо изменять некоторые из определенных настроек. Например, Button теперь использует курсор Help (вместо курсора Hand, определенного в стиле):

```
<Button x:Name="btnTestButton" Content="Yes, but we are reusing settings!"
   Cursor="Help" Style="{StaticResource BasicControlStyle}" Width="250" />
```

Стили обрабатываются перед настройками индивидуальных свойств элемента управления, к которому применен стиль; следовательно, элементы управления могут "переопределять" настройки от случая к случаю.

Ограничение применения стиля с помощью TargetType

В настоящий момент наш стиль определен так, что его может задействовать любой элемент управления (и должен делать это явно, устанавливая свое свойство Style), поскольку каждое свойство уточнено посредством класса Control. Для программы, определяющей десятки настроек, в результате получился бы значительный объем повторяющегося кода. Один из способов несколько улучшить ситуацию предусматривает использование атрибута TargetType. Добавление атрибута TargetType к открывающему дескриптору <Style> позволяет точно указать, где стиль может быть применен (в данном примере это делается в файле App.xaml):

```
<Style x:Key ="BasicControlStyle" TargetType="Control">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Height" Value = "40"/>
  <Setter Property = "Cursor" Value = "Hand"/>
</Style>
```

На заметку! При построении стиля, использующего базовый класс, нет нужды беспокоиться о том, что значение присваивается свойству зависимости, которое не поддерживается производными типами. Если производный тип не поддерживает заданное свойство зависимости, то оно игнорируется.

Кое в чем это помогло, но все равно мы имеем стиль, который может применяться к любому элементу управления. Атрибут `TargetType` более удобен, когда необходимо определить стиль, который может быть применен только к отдельному типу элементов управления. Добавим в словарь ресурсов приложения следующий стиль:

```
<Style x:Key ="BigGreenButton" TargetType="Button">
  <Setter Property = "FontSize" Value ="20"/>
  <Setter Property = "Height" Value = "100"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Background" Value = "DarkGreen"/>
  <Setter Property = "Foreground" Value = "Yellow"/>
</Style>
```

Рис. 29.12. Элементы управления с разными стилями

Этот стиль будет работать только с элементами управления `Button` (или подклассами `Button`). Если применить его к несовместимому элементу, то возникнут ошибки разметки и компиляции. Когда элемент управления `Button` использует новый стиль так, как продемонстрировано ниже, вывод будет выглядеть похожим на то, что показано на рис. 29.12:

```
<Button x:Name="btnTestButton" Content="OK!" Cursor="Help"
Style="{StaticResource BigGreenButton}" Width="250" />
```

Еще одно действие атрибута `TargetType` заключается в том, что стиль будет применен ко всем элементам данного типа внутри области определения стиля при условии, что свойство `x:Key` *отсутствует*. Вскоре об этом пойдет речь более подробно.

Автоматическое применение стиля с помощью `TargetType`

Пусть необходимо гарантировать, что все элементы управления `TextBox` имеют одинаковый внешний вид и поведение. Предположим, что определен стиль в виде ресурса уровня приложения, доступ к которому имеют все окна программы. Хотя это шаг в правильном направлении, но если есть множество окон с многочисленными элементами управления `TextBox`, то свойство `Style` придется устанавливать много раз!

Стили WPF могут быть неявно применены ко всем элементам управления внутри заданной области XAML. Для создания такого стиля используется свойство `TargetType`, но ресурсу `Style` не присваивается значение `x:Key`. "Неименованный стиль" подобного рода теперь применяется ко всем элементам управления корректного типа. Вот еще один стиль уровня приложения, который будет автоматически применяться ко всем элементам управления `TextBox` в текущем приложении:

```
<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
</Style>
```

После этого можно определять любое количество элементов управления TextBox, и все они автоматически получат установленный внешний вид. Если какому-то элементу управления TextBox не нужен такой стандартный внешний вид, то он может отказаться от него, установив свойство Style в {x:Null}. Например, элемент txtTest будет иметь неименованный стандартный стиль, а элемент txtTest2 сделает все самостоятельно:

```
<TextBox x:Name="txtTest"/>
<TextBox x:Name="txtTest2" Style="{x:Null}" BorderBrush="Black"
         BorderThickness="5" Height="60" Width="100" Text="Ha!"/>
```

Создание подклассов существующих стилей

Новые стили можно также строить на основе существующего стиля посредством свойства BasedOn. Расширяемый стиль должен иметь подходящий атрибут x:Key в словаре, т.к. производный стиль будет ссылаться на него по имени, используя расширение разметки {StaticResource} или {DynamicResource}. Ниже представлен новый стиль, основанный на стиле BigGreenButton, который поворачивает элемент управления Button на 20 градусов:

Рис. 29.13. Использование производного стиля

```
<!-- Этот стиль основан на BigGreenButton -->
<Style x:Key ="TiltButton" TargetType="Button"
       BasedOn = "{StaticResource BigGreenButton}">
  <Setter Property = "Foreground" Value = "White"/>
  <Setter Property = "RenderTransform">
    <Setter.Value>
      <RotateTransform Angle = "20"/>
    </Setter.Value>
  </Setter>
</Style>
```

На этот раз вывод будет выглядеть так, как показано на рис. 29.13.

Определение стилей с триггерами

Стили WPF могут также содержать триггеры за счет упаковки объектов Trigger в коллекцию Triggers объекта Style. Использование триггеров в стиле позволяет определять некоторые элементы <Setter> таким образом, что они будут применяться только в случае истинности заданного условия триггера. Например, возможно требуется увеличить размер шрифта, когда курсор мыши находится над кнопкой. Или, скажем, нужно подсветить текстовое поле, имеющее фокус, с использованием фона указанного цвета. Триггеры очень полезны в ситуациях подобного рода, потому что они позволяют предпринимать специфические действия при изменении свойства, не требуя написания явной логики C# в файле отделенного кода.

Далее приведена модифицированная разметка для стиля элементов управления типа TextBox, которая обеспечивает установку фона желтого цвета, когда элемент TextBox получает фокус:

```
<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
```

```
  <!-- Следующий установщик будет применен, только
       когда текстовое поле находится в фокусе -->
  <Style.Triggers>
    <Trigger Property = "IsFocused" Value = "True">
      <Setter Property = "Background" Value = "Yellow"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

При тестировании этого стиля обнаружится, что по мере перехода с помощью клавиши <Tab> между элементами `TextBox` текущий выбранный `TextBox` получает фон желтого цвета (если только стиль не отключен путем присваивания `{x:Null}` свойству `Style`).

Триггеры свойств также весьма интеллектуальны в том смысле, что когда условие триггера *не истинно*, свойство автоматически получает стандартное значение. Следовательно, как только `TextBox` теряет фокус, он также автоматически принимает стандартный цвет без какой-либо работы с вашей стороны. В противоположность этому триггеры событий (которые исследовались при рассмотрении анимации WPF) не возвращаются автоматически в предыдущее состояние.

Определение стилей с несколькими триггерами

Триггеры могут быть спроектированы так, что определенные элементы `<Setter>` будут применяться, когда истинно *несколько условий*. Пусть необходимо устанавливать фон элемента `TextBox` в `Yellow` только в случае, если он имеет активный фокус и курсор мыши находится внутри его границ. Для этого можно использовать элемент `<MultiTrigger>` и определить в нем каждое условие:

```
<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
  <!-- Следующий установщик будет применен, только когда текстовое
       поле имеет фокус И над ним находится курсор мыши -->
  <Style.Triggers>
    <MultiTrigger>
      <MultiTrigger.Conditions>
          <Condition Property = "IsFocused" Value = "True"/>
          <Condition Property = "IsMouseOver" Value = "True"/>
      </MultiTrigger.Conditions>
      <Setter Property = "Background" Value = "Yellow"/>
    </MultiTrigger>
  </Style.Triggers>
</Style>
```

Анимированные стили

Стили также могут содержать в себе триггеры, которые запускают анимационную последовательность. Ниже показан последний стиль, который после применения к элементам управления `Button` заставит их увеличиваться и уменьшаться в размерах, когда курсор мыши находится внутри границ кнопки:

```
<!-- Стиль увеличивающейся кнопки -->
<Style x:Key = "GrowingButtonStyle" TargetType="Button">
  <Setter Property = "Height" Value = "40"/>
  <Setter Property = "Width" Value = "100"/>
  <Style.Triggers>
    <Trigger Property = "IsMouseOver" Value = "True">
      <Trigger.EnterActions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "Height">
            <DoubleAnimation From = "40" To = "200"
                             Duration = "0:0:2" AutoReverse="True"/>
          </Storyboard>
        </BeginStoryboard>
      </Trigger.EnterActions>
    </Trigger>
  </Style.Triggers>
</Style>
```

Здесь коллекция триггеров наблюдает за тем, когда свойство `IsMouseOver` возвратит значение `true`. После того как это произойдет, определяется элемент `<Trigger.EnterActions>` для выполнения простой раскадровки, которая заставляет кнопку за две секунды увеличиться до значения `Height`, равного `200` (и затем возвратиться к значению `Height`, равному `40`). Чтобы отслеживать другие изменения свойств, можно также добавить область `<Trigger.ExitActions>` и определить в ней любые специальные действия, которые должны быть выполнены, когда `IsMouseOver` изменяется на `false`.

Применение стилей в коде

Вспомните, что стиль может быть применен также во время выполнения. Это удобно, когда у конечных пользователей должна быть возможность выбора внешнего вида для их пользовательского интерфейса, требуется принудительно устанавливать внешний вид и поведение на основе настроек безопасности (например, стиль `DisableAllButton`) или еще в каких-то ситуациях.

В текущем проекте было определено порядочное количество стилей, многие из которых могут применяться к элементам управления `Button`. Давайте переделаем пользовательский интерфейс главного окна, чтобы позволить пользователю выбирать имена этих стилей в элементе управления `ListBox`. На основе выбранного имени будет применен соответствующий стиль. Вот финальная разметка для элемента `<Window>`:

```
<Window x:Class="WpfStyles.MainWindow"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:WpfStyles"
  mc:Ignorable="d"
  Height="350" Title="A Window with Style!"
  Width="525" WindowStartupLocation="CenterScreen">
  <DockPanel>
    <StackPanel Orientation="Horizontal" DockPanel.Dock="Top">
      <Label Content="Please Pick a Style for this Button" Height="50"/>
      <ListBox x:Name ="lstStyles" Height ="80" Width ="150" Background="LightBlue"
               SelectionChanged ="comboStyles_Changed" />
    </StackPanel>
    <Button x:Name="btnStyle" Height="40" Width="100" Content="OK!"/>
  </DockPanel>
</Window>
```

Элемент управления `ListBox` (по имени `lstStyles`) будет динамически заполняться внутри конструктора окна:

```
public MainWindow()
{
  InitializeComponent();
  // Заполнить окно со списком всеми стилями для элементов Button.
  lstStyles.Items.Add("GrowingButtonStyle");
  lstStyles.Items.Add("TiltButton");
  lstStyles.Items.Add("BigGreenButton");
  lstStyles.Items.Add("BasicControlStyle");}
}
```

Последней задачей является обработка события `SelectionChanged` в связанном файле кода. Обратите внимание, что в следующем коде имеется возможность извлечения текущего ресурса по имени с использованием унаследованного метода `TryFindResouce()`:

```
private void comboStyles_Changed(object sender, SelectionChangedEventArgs e)
{
  // Получить имя стиля, выбранное в окне со списком.
  var currStyle = (Style)TryFindResource(lstStyles.SelectedValue);
  if (currStyle == null) return;
  // Установить стиль для типа кнопки.
  this.btnStyle.Style = currStyle;
}
```

После запуска приложения появляется возможность выбора одного из четырех стилей кнопок на лету. На рис. 29.14 показано готовое приложение в действии.

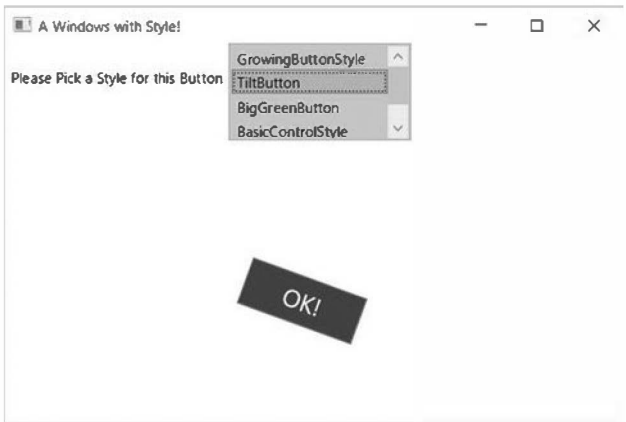


Рис. 29.14. Элементы управления с разными стилями

Исходный код. Проект `WpfStyles` доступен в подкаталоге `Chapter_29`.

Логические деревья, визуальные деревья и стандартные шаблоны

Теперь, когда вы понимаете, что собой представляют стили и ресурсы, есть еще несколько тем, которые потребуется рассмотреть, прежде чем приступать к изучению пос-

троения специальных элементов управления. В частности, необходимо выяснить разницу между *логическим деревом*, *визуальным деревом* и *стандартным шаблоном*. При вводе разметки XAML в Visual Studio или в редакторе вроде Kaxaml разметка является *логическим представлением* документа XAML. В случае написания кода C#, который добавляет новые элементы в элемент управления `StackPanel`, новые элементы вставляются в логическое дерево. По существу логическое представление отражает то, как содержимое будет позиционировано внутри разнообразных диспетчеров компоновки для главного элемента `Window` (или другого корневого элемента, такого как `Page` или `NavigationWindow`).

Однако за каждым логическим деревом стоит намного более сложное представление, которое называется *визуальным деревом* и внутренне применяется инфраструктурой WPF для корректной визуализации элементов на экране. Внутри любого визуального дерева будут находиться полные детали шаблонов и стилей, используемых для визуализации каждого объекта, включая все необходимые рисунки, фигуры, визуальные объекты и объекты анимации.

Полезно уяснить разницу между логическим и визуальным деревьями, потому что при построении специального шаблона элемента управления на самом деле производится замена всего или части стандартного визуального дерева элемента управления собственным вариантом. Следовательно, если необходимо, чтобы элемент управления `Button` визуализировался в виде звездообразной фигуры, то можно определить новый шаблон такого рода и подключить его к визуальному дереву `Button`. Логически тип останется тем же типом `Button`, поддерживая все ожидаемые свойства, методы и события. Но визуально он выглядит совершенно по-другому. Один лишь этот факт делает WPF исключительно полезным API-интерфейсом, поскольку другие инструментальные наборы для создания кнопки звездообразной формы потребовали бы построения совершенно нового класса. В инфраструктуре WPF нужно просто определить новую разметку.

На заметку! Элементы управления WPF часто описывают как *лишенные внешности*. Это относится к тому факту, что внешний вид элемента управления WPF совершенно не зависит от его поведения и допускает настройку.

Программное инспектирование логического дерева

Хотя анализ логического дерева окна во время выполнения — не слишком распространенное действие при программировании с применением WPF, полезно упомянуть о том, что в пространстве имен `System.Windows` определен класс `LogicalTreeHelper`, который позволяет инспектировать структуру логического дерева во время выполнения. Для иллюстрации связи между логическими деревьями, визуальными деревьями и шаблонами элементов управления создадим новый проект WPF Application по имени `TreesAndTemplatesApp`.

Добавим в разметку окна два элемента управления `Button` и доступный только для чтения элемент `TextBox` большого размера с включенными линейками прокрутки. Создадим в IDE-среде обработчики событий `Click` для каждой кнопки. Ниже показана результирующая разметка XAML:

```
<Window x:Class="TreesAndTemplatesApp.MainWindow"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:TreesAndTemplatesApp"
  mc:Ignorable="d"
  Title="Fun with Trees and Templates" Height="518"
  Width="836" WindowStartupLocation="CenterScreen">
```

```
  <DockPanel LastChildFill="True">
    <Border Height="50" DockPanel.Dock="Top" BorderBrush="Blue">
      <StackPanel Orientation="Horizontal">
        <Button x:Name="btnShowLogicalTree" Content="Logical Tree of Window"
                Margin="4" BorderBrush="Blue" Height="40"
                Click="btnShowLogicalTree_Click"/>
        <Button x:Name="btnShowVisualTree" Content="Visual Tree of Window"
                BorderBrush="Blue" Height="40"
                Click="btnShowVisualTree_Click"/>
      </StackPanel>
    </Border>
    <TextBox x:Name="txtDisplayArea" Margin="10"
             Background="AliceBlue" IsReadOnly="True"
             BorderBrush="Red" VerticalScrollBarVisibility="Auto"
             HorizontalScrollBarVisibility="Auto" />
  </DockPanel>
</Window>
```

Внутри файла кода C# определим переменную-член `_dataToShow` **типа** `string`. **В обработчике события** `Click` **объекта** `btnShowLogicalTree` **вызовем вспомогательную функцию, которая продолжит вызывать себя рекурсивно с целью заполнения строковой переменной логическим деревом** `Window`. **Для этого будет вызван статический метод** `GetChildren()` **объекта** `LogicalTreeHelper`. **Вот необходимый код:**

```
private string _dataToShow = string.Empty;

private void btnShowLogicalTree_Click(object sender, RoutedEventArgs e)
{
  _dataToShow = "";
  BuildLogicalTree(0, this);
  txtDisplayArea.Text = _dataToShow;
}

void BuildLogicalTree(int depth, object obj)
{
  // Добавить имя типа к переменной-члену _dataToShow.
  _dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";

  // Если элемент - не DependencyObject, то пропустить его.
  if (!(obj is DependencyObject))
    return;

  // Выполнить рекурсивный вызов для каждого логического дочернего элемента.
  foreach (var child in
    LogicalTreeHelper.GetChildren(
    (DependencyObject)obj))
  {
    BuildLogicalTree(depth + 5, child);
  }
}
```

После запуска приложения и щелчка на кнопке Logical Tree of Window (Логическое дерево окна) в текстовой области отобразится древовидное представление, которое выглядит как почти точная копия исходной разметки XAML (рис. 29.15).

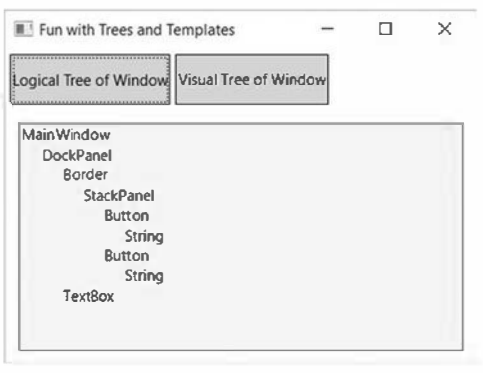


Рис. 29.15. Просмотр логического дерева во время выполнения

Программное инспектирование визуального дерева

Визуальное дерево объекта Window также можно инспектировать во время выполнения с использованием класса VisualTreeHelper из пространства имен System.Windows.Media. Далее приведена реализация обработчика события Click для второго элемента управления Button (btnShowVisualTree), которая выполняет похожую рекурсивную логику для построения текстового представления визуального дерева:

```
private void btnShowVisualTree_Click(object sender, RoutedEventArgs e)
{
  _dataToShow = "";
  BuildVisualTree(0, this);
  txtDisplayArea.Text = _dataToShow;
}
void BuildVisualTree(int depth, DependencyObject obj)
{
  // Добавить имя типа к переменной-члену _dataToShow.
  _dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";
  // Выполнить рекурсивный вызов для каждого визуально дочернего элемента.
  for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
  {
    BuildVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
  }
}
```

На рис. 29.16 видно, что визуальное дерево открывает доступ к нескольким низкоуровневым агентам визуализации, таким как ContentPresenter, AdornerDecorator, TextBoxLineDrawingVisual и т.д.

Программное инспектирование стандартного шаблона элемента управления

Вспомните, что визуальное дерево применяется инфраструктурой WPF для выяснения, каким образом визуализировать элемент Window и все содержащиеся в нем элементы. Каждый элемент управления WPF хранит собственный набор команд визуализации внутри своего стандартного шаблона. С точки зрения программирования любой шаблон может быть представлен как экземпляр класса ControlTemplate. Кроме того, стандартный шаблон элемента управления можно получить через свойство Template:

```
// Получить стандартный шаблон
// элемента Button.
Button myBtn = new Button();
ControlTemplate template
              = myBtn.Template;
```

Подобным же образом можно создать в коде новый объект ControlTemplate и подключить его к свойству Template элемента управления:

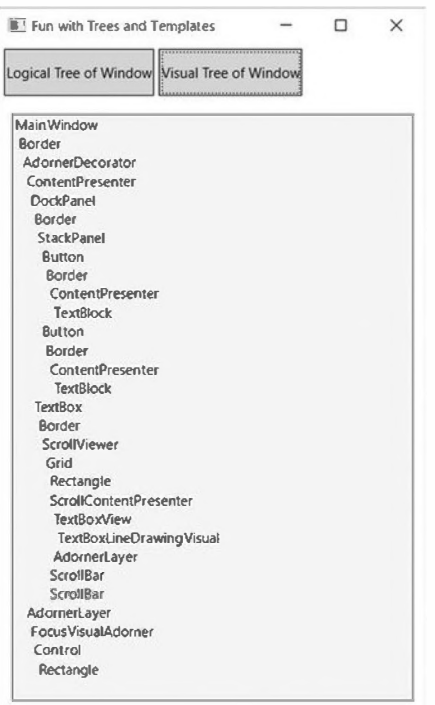


Рис. 29.16. Просмотр визуального дерева во время выполнения

```
// Подключить новый шаблон для использования в кнопке.
Button myBtn = new Button();
ControlTemplate customTemplate = new ControlTemplate();
// Предположим, что этот метод добавляет весь код для звездообразного шаблона.
MakeStarTemplate(customTemplate);
myBtn.Template = customTemplate;
```

Наряду с тем, что новый шаблон можно строить в коде, намного чаще это делается в разметке XAML. Тем не менее, прежде чем приступить к построению собственных шаблонов, давайте завершим текущий пример и добавим возможность просмотра стандартного шаблона для элемента управления WPF во время выполнения. Это может оказаться по-настоящему полезным способом ознакомления с общей структурой шаблона. Для начала добавим в разметку окна новый контейнер `StackPanel` с элементами управления; он стыкован с левой стороной главной панели `DockPanel` (находится прямо перед элементом `<TextBox>`) и определен следующим образом:

```
<Border DockPanel.Dock="Left" Margin="10" BorderBrush="DarkGreen"
        BorderThickness="4" Width="358">
  <StackPanel>
   <Label Content="Enter Full Name of WPF Control" Width="340" FontWeight="DemiBold" />
   <TextBox x:Name="txtFullName" Width="340" BorderBrush="Green"
            Background="BlanchedAlmond" Height="22"
            Text="System.Windows.Controls.Button" />
   <Button x:Name="btnTemplate" Content="See Template" BorderBrush="Green"
           Height="40" Width="100" Margin="5"
           Click="btnTemplate_Click" HorizontalAlignment="Left" />
   <Border BorderBrush="DarkGreen" BorderThickness="2" Height="260"
           Width="301" Margin="10" Background="LightGreen" >
     <StackPanel x:Name="stackTemplatePanel" />
    </Border>
  </StackPanel>
</Border>
```

Обратите внимание на пустой элемент `StackPanel` по имени `stackTemplatePanel`, поскольку мы будем на него ссылаться в коде. Добавим пустой обработчик события `Click` для элемента `btnTemplate`:

```
private void btnTemplate_Click(object sender, RoutedEventArgs e){ }
```

Теперь окно должно выглядеть похожим на то, что показано на рис. 29.17.

Рис. 29.17. Обновленный пользовательский интерфейс окна

Текстовая область слева вверху позволяет вводить полностью заданное имя элемента управления WPF, расположенного в сборке PresentationFramework.dll. После того как библиотека загружена, экземпляр элемента управления динамически создается и отображается в большом квадрате слева внизу. Наконец, в текстовой области справа будет отображаться стандартный шаблон элемента управления. Добавим в класс C# новую переменную-член типа Control:

```
private Control _ctrlToExamine = null;
```

Ниже показан остальной код, который требует импортирования пространств имен System.Reflection, System.Xml и System.Windows.Markup:

```
private void btnTemplate_Click(object sender, RoutedEventArgs e)
{
  _dataToShow = "";
  ShowTemplate();
  txtDisplayArea.Text = _dataToShow;
}
private void ShowTemplate()
{
  // Удалить элемент, который в текущий момент находится
  // в области предварительного просмотра.
  if (_ctrlToExamine != null)
    stackTemplatePanel.Children.Remove(_ctrlToExamine);
  try
  {
    // Загрузить PresentationFramework и создать экземпляр
    // указанного элемента управления. Установить его размеры для
    // отображения, а затем добавить в пустой контейнер StackPanel.
    Assembly asm = Assembly.Load("PresentationFramework, Version=4.0.0.0," +
      "Culture=neutral, PublicKeyToken=31bf3856ad364e35");
    _ctrlToExamine = (Control)asm.CreateInstance(txtFullName.Text);
    _ctrlToExamine.Height = 200;
    _ctrlToExamine.Width = 200;
    _ctrlToExamine.Margin = new Thickness(5);
    stackTemplatePanel.Children.Add(_ctrlToExamine);
    // Определить настройки XML для предохранения отступов.
    var xmlSettings = new XmlWriterSettings{Indent = true};
    // Создать объект StringBuilder для хранения разметки XAML.
    var strBuilder = new StringBuilder();
    // Создать объект XmlWriter на основе имеющихся настроек.
    var xWriter = XmlWriter.Create(strBuilder, xmlSettings);
    // Сохранить разметку XAML в объекте XmlWriter на основе ControlTemplate.
    XamlWriter.Save(_ctrlToExamine.Template, xWriter);
    // Отобразить разметку XAML в текстовом поле.
    _dataToShow = strBuilder.ToString();
  }
  catch (Exception ex)
  {
    _dataToShow = ex.Message;
  }
}
```

Большая часть работы связана с отображением скомпилированного ресурса BAML на строку разметки XAML. На рис. 29.18 демонстрируется финальное приложение в действии на примере вывода стандартного шаблона для элемента управления System.Windows.Controls.DatePicker. Здесь отображается календарь, который доступен по щелчку на кнопке в правой части элемента управления.

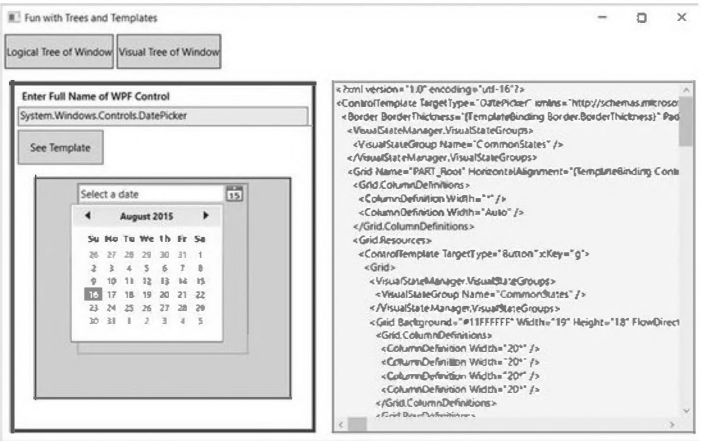


Рис. 29.18. Просмотр стандартного шаблона элемента управления во время выполнения

К настоящему моменту вы должны лучше понимать взаимосвязь между логическими деревьями, визуальными деревьями и стандартными шаблонами элементов управления. Остаток главы будет посвящен построению специальных шаблонов и пользовательских элементов управления.

Исходный код. Проект `TreesAndTemplatesApp` доступен в подкаталоге `Chapter_29`.

Построение шаблона элемента управления с помощью инфраструктуры триггеров

Специальный шаблон для элемента управления можно создавать с помощью только кода C#. Такой подход предусматривает добавление данных к объекту `ControlTemplate` и затем присваивание его свойству `Template` элемента управления. Однако большую часть времени внешний вид и поведение `ControlTemplate` будут определяться с использованием разметки XAML и фрагментов кода (мелких или крупных) для управления поведением во время выполнения.

В оставшемся материале главы вы узнаете, как строить специальные шаблоны с применением Visual Studio. Попутно вы ознакомитесь с инфраструктурой триггеров WPF и научитесь использовать анимацию для встраивания визуальных подсказок конечным пользователям. Применение при построении сложных шаблонов только IDE-среды Visual Studio может быть связано с довольно большим объемом набора и трудной работы. Конечно, шаблоны производственного уровня получат преимущество от использования продукта Blend for Visual Studio, который теперь является бесплатным установочным файлом, сопровождающим Visual Studio. Тем не менее, поскольку настоящее издание книги не включает описание этого продукта, время засучить рукава и приступить к написанию некоторой разметки.

Для начала создадим новый проект WPF Application по имени `ButtonTemplate`. Основной интерес в этом проекте представляют механизмы создания и применения шаблонов, так что разметка для главного окна очень проста:

```
<Window x:Class="ButtonTemplate.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

```
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:ButtonTemplate"
    mc:Ignorable="d"
  Title="Fun with Templates" Height="350" Width="525">
  <StackPanel>
    <Button x:Name="myButton" Width="100" Height="100"
            Click="myButton_Click"/>
  </StackPanel>
</Window>
```

В обработчике события `Click` мы просто отображаем окно сообщения (посредством вызова `MessageBox.Show()`) с подтверждением щелчка на элементе управления. При построении специальных шаблонов помните, что *поведение* элемента управления неизменно, но его *внешний вид* может варьироваться.

В настоящее время этот элемент `Button` визуализируется с использованием стандартного шаблона, который представляет собой ресурс BAML внутри заданной сборки WPF, как было проиллюстрировано в последнем примере. Определение собственного шаблона по существу сводится к замене стандартного визуального дерева своим вариантом. Для начала модифицируем определение элемента `<Button>`, указав новый шаблон с применением синтаксиса "свойство-элемент". Этот шаблон придаст элементу управления округлый вид:

```
<Button x:Name="myButton" Width="100" Height="100"
        Click="myButton_Click">
  <Button.Template>
    <ControlTemplate>
      <Grid x:Name="controlLayout">
        <Ellipse x:Name="buttonSurface" Fill = "LightBlue"/>
        <Label x:Name="buttonCaption" VerticalAlignment = "Center"
               HorizontalAlignment = "Center"
               FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
      </Grid>
    </ControlTemplate>
  </Button.Template>
</Button>
```

Здесь определен шаблон, который состоит из именованного элемента `Grid`, содержащего именованные элементы `Ellipse` и `Label`. Поскольку в `Grid` не определены строки и столбцы, каждый дочерний элемент укладывается поверх предыдущего элемента управления, позволяя центрировать содержимое. Если теперь запустить приложение, то можно заметить, что событие `Click` будет инициироваться *только* в ситуации, когда курсор мыши находится внутри границ элемента `Ellipse` (т.е. не на углах, окружающих эллипс). Это замечательная возможность архитектуры шаблонов WPF, т.к. нет нужды повторно вычислять попадание курсора, проверять граничные условия или предпринимать другие низкоуровневые действия. Таким образом, если шаблон использует объект `Polygon` для отображения какой-то необычной геометрии, то можно иметь уверенность в том, что детали проверки попадания курсора будут соответствовать форме элемента управления, а не более крупного ограничивающего прямоугольника.

Шаблоны как ресурсы

В текущий момент наш шаблон внедрен в специфический элемент управления `Button`, что ограничивает возможности его многократного применения. В идеале шаблон круглой кнопки следовало бы поместить в словарь ресурсов, чтобы его можно было использовать в разных проектах, или как минимум перенести в контейнер ресурсов

приложения для многократного применения внутри проекта. Давайте переместим локальный ресурс `Button` на уровень приложения. Отыщем свойство `Template` элемента `Button` в окне Properties (оно находится в разделе Miscellaneous (Разные)). Щелкнем на значке с небольшим черным квадратом и в открывшемся контекстом меню выберем пункт Convert to New Resource (Преобразовать в новый ресурс).

В результирующем диалоговом окне определим новый шаблон по имени `RoundButtonTemplate` и сохраним его на уровне `Application` (т.е. в файле `App.xaml`), как показано на рис. 29.19.

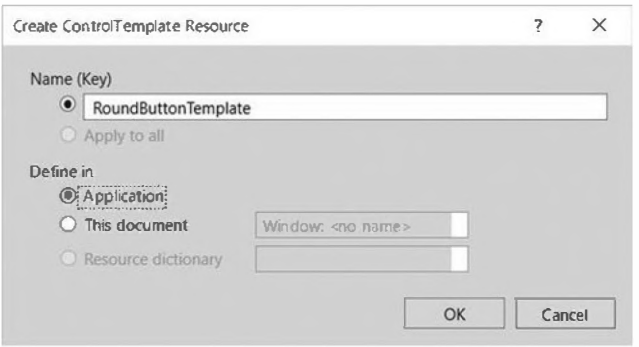


Рис. 29.19. Помещение ресурса в файл `App.xaml`

После этого в разметке объекта `Application` обнаружатся следующие данные:

```
<Application x:Class="ButtonTemplate.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:ButtonTemplate"
  StartupUri="MainWindow.xaml">
  <Application.Resources>
    <ControlTemplate x:Key="RoundButtonTemplate" TargetType="{x:Type Button}>
      <Grid x:Name="controlLayout">
        <Ellipse x:Name="buttonSurface" Fill = "LightBlue"/>
        <Label x:Name="buttonCaption" VerticalAlignment = "Center"
               HorizontalAlignment = "Center"
               FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
      </Grid>
    </ControlTemplate>
  </Application.Resources>
</Application>
```

Из-за того, что этот ресурс доступен всему приложению, можно определять любое количество круглых кнопок. В целях тестирования создадим два дополнительных элемента управления `Button`, использующих данный шаблон (обрабатывать событие `Click` для них не нужно):

```
<StackPanel>
  <Button x:Name="myButton" Width="100" Height="100"
          Click="myButton_Click"
          Template="{StaticResource RoundButtonTemplate}"></Button>
  <Button x:Name="myButton2" Width="100" Height="100"
          Template="{StaticResource RoundButtonTemplate}"></Button>
  <Button x:Name="myButton3" Width="100" Height="100"
          Template="{StaticResource RoundButtonTemplate}"></Button>
</StackPanel>
```

В Visual Studio 2015 и .NET 4.6 по умолчанию применяются расширения разметки {DynamicResource}. Однако в качестве стандартных расширений разметки предпочтительнее использовать {StaticResource}, переходя на {DynamicResource} только в случае необходимости. В рассматриваемых примерах понадобится изменить все вхождения {DynamicResource} на {StaticResource}.

Встраивание визуальных подсказок с использованием триггеров

При определении специального шаблона также удаляются все визуальные подсказки стандартного шаблона. Например, стандартный шаблон кнопки содержит разметку, которая задает внешний вид элемента управления при возникновении определенных событий пользовательского интерфейса, таких как получение фокуса, щелчок кнопкой мыши, включение (или отключение) и т.д. Пользователи довольно хорошо приучены к визуальным подсказкам подобного рода, т.к. они придают элементу управления некоторую осязаемую реакцию. Тем не менее, в шаблоне RoundButtonTemplate разметка такого типа не определена, так что вид элемента управления остается идентичным независимо от действий мыши. В идеальном случае элемент должен выглядеть немного по-другому, когда на нем совершается щелчок (возможно, за счет изменения цвета или отбрасывания тени), чтобы уведомить пользователя об изменении визуального состояния.

Когда появилась первая версия инфраструктуры WPF, единственным способом реализации таких визуальных подсказок было добавление к шаблону любого количества триггеров, которые в случае истинности их условий обычно изменяли значения свойств объекта или запускали раскадровку анимации (либо делали то и другое). Для примера модифицируем шаблон RoundButtonTemplate, добавив разметку, которая при нахождении курсора на поверхности элемента управления будет изменять цвет фона на синий, а цвет переднего плана на желтый:

```
<ControlTemplate x:Key="RoundButtonTemplate" TargetType="Button" >
  <Grid x:Name="controlLayout">
    <Ellipse x:Name="buttonSurface" Fill="LightBlue" />
    <Label x:Name="buttonCaption" Content="OK!" FontSize="20" FontWeight="Bold"
           HorizontalAlignment="Center" VerticalAlignment="Center" />
  </Grid>
    <ControlTemplate.Triggers>
      <Trigger Property = "IsMouseOver" Value = "True">
        <Setter TargetName = "buttonSurface" Property = "Fill" Value = "Blue"/>
<Setter TargetName = "buttonCaption" Property = "Foreground" Value = "Yellow"/>
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

Снова запустив программу, можно заметить, что цвет изменяется в зависимости от того, находится курсор мыши в области Ellipse или нет. Ниже показан еще один триггер, который при нажатии кнопки мыши на элементе управления будет уменьшать размеры контейнера Grid (и, следовательно, всех его дочерних элементов). Добавим в коллекцию <ControlTemplate.Triggers> такую разметку:

```
<Trigger Property = "IsPressed" Value="True">
  <Setter TargetName="controlLayout"
          Property="RenderTransformOrigin" Value="0.5,0.5"/>
  <Setter TargetName="controlLayout" Property="RenderTransform">
    <Setter.Value>
      <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
    </Setter.Value>
  </Setter>
</Trigger>
```

Роль расширения разметки {TemplateBinding}

Построенный шаблон может быть применен только к элементам Button, поэтому разумно ожидать, что должен существовать способ установки свойств элемента <Button>, которые приведут к визуализации шаблона в уникальной манере. Например, сейчас в свойстве Fill элемента Ellipse жестко закодирован синий цвет, а свойство Content элемента Label всегда устанавливается в строковое значение OK!. Если нужны кнопки с другими цветами и текстовыми значениями, то можно попробовать их определить в главном окне следующим образом:

```
<StackPanel>
  <Button x:Name="myButton" Width="100" Height="100"
          Background="Red" Content="Howdy!"
          Click="myButton_Click"
          Template="{StaticResource RoundButtonTemplate}" />
  <Button x:Name="myButton2" Width="100" Height="100"
          Background="LightGreen" Content="Cancel!"
          Template="{StaticResource RoundButtonTemplate}" />
  <Button x:Name="myButton3" Width="100" Height="100"
          Background="Yellow" Content="Format"
          Template="{StaticResource RoundButtonTemplate}" />
</StackPanel>
```

Однако независимо от того факта, что свойства Background и Content каждого элемента Button установлены в уникальные значения, все равно получаются три кнопки синего цвета с текстом OK!. Проблема в том, что свойства элемента управления, использующего шаблон (Button), не соответствуют в точности свойствам элементов шаблона (таким как свойство Fill элемента Ellipse). Кроме того, хотя элемент Label имеет свойство Content, значение, определенное в области <Button>, не направляется автоматически во внутренний дочерний элемент шаблона.

Во время построения шаблона такие проблемы можно решать с применением расширения разметки {TemplateBonding}. Оно позволяет захватывать настройки свойств, которые определены элементом управления, использующим шаблон, и задействовать их при установке значений в самом шаблоне. Ниже приведена переделанная версия шаблона RoundButtonTemplate, в которой это расширение разметки применяется для отображения свойства Background элемента Button на свойство Fill элемента Ellipse; здесь также обеспечивается действительная передача значения Content элемента Button свойству Content элемента Label:

```
<ControlTemplate x:Key="RoundButtonTemplate" TargetType="Button">
  <Grid x:Name="controlLayout">
    <Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}"/>
    <Label x:Name="buttonCaption" Content="{TemplateBinding Content}"
           FontSize="20" FontWeight="Bold"
           HorizontalAlignment="Center" VerticalAlignment="Center" />
  </Grid>
  <ControlTemplate.Triggers>
    ...
  </ControlTemplate.Triggers>
</ControlTemplate>
```

После такого обновления появляется возможность создания кнопок с разными цветами и текстом (рис. 29.20). Вот пример разметки XAML:

```
<Button x:Name="myButton" Width="100" Height="100"
        Background="Red" Content="Howdy!"
```

```
  Click="myButton_Click" Style="{StaticResource RoundButtonStyle}" />
<Button x:Name="myButton2" Width="100" Height="100"
        Background="LightGreen" Content="Cancel!"
  Style="{StaticResource RoundButtonStyle}" />
<Button x:Name="myButton3" Width="100" Height="100"
        Background="Yellow" Content="Format"
  Style="{StaticResource RoundButtonStyle}" />
```

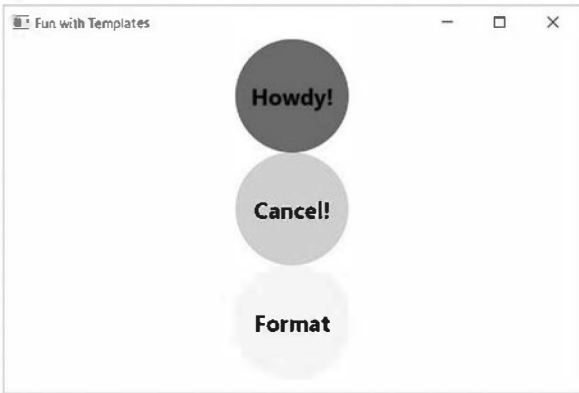


Рис. 29.20. Привязки шаблона позволяют передавать значения внутренним элементам управления

Роль класса ContentPresenter

При проектировании шаблона для отображения текстового значения элемента управления использовался элемент `Label`. Подобно `Button` он поддерживает свойство `Content`. Следовательно, если применяется расширение разметки `{TemplateBinding}`, то можно определять элемент `Button` со сложным содержимым, а не с простой строкой. Например:

```
<Button x:Name="myButton4" Width="100" Height="100" Background="Yellow"
        Template="{StaticResource RoundButtonTemplate}">
  <Button.Content>
    <ListBox Height="50" Width="75">
      <ListBoxItem>Hello</ListBoxItem>
      <ListBoxItem>Hello</ListBoxItem>
      <ListBoxItem>Hello</ListBoxItem>
    </ListBox>
  </Button.Content>
</Button>
```

Для этого конкретного элемента управления все работает ожидаемым образом. Но что, если необходимо передать сложное содержимое члену шаблона, который *не* имеет свойства `Content`? Когда в шаблоне требуется определить обобщенную *область отображения содержимого*, то вместо элемента управления специфического типа (`Label` или `TextBox`) можно использовать класс `ContentPresenter`. Хотя в рассматриваемом примере в этом нет нужды, ниже показана простая разметка, иллюстрирующая способ построения специального шаблона, который применяет класс `ContentPresenter` для отображения значения свойства `Content` элемента управления, использующего шаблон:

```
<!-- Этот шаблон кнопки отобразит то, что установлено
    в свойстве Content размещающей кнопки -->
<ControlTemplate x:Key="NewRoundButtonTemplate" TargetType="Button">
  <Grid>
    <Ellipse Fill="{TemplateBinding Background}"/>
    <ContentPresenter HorizontalAlignment="Center"
                      VerticalAlignment="Center"/>
  </Grid>
</ControlTemplate>
```

Встраивание шаблонов в стили

В текущий момент шаблон просто определяет базовый внешний вид и поведение элемента управления `Button`. Тем не менее, за процесс установки базовых свойств элемента управления (содержимого, размера шрифта, веса шрифта и т.д.) отвечает сам элемент `Button`:

```
<!-- Сейчас базовые значения свойств должен устанавливать сам элемент
Button, а не шаблон -->
<Button x:Name ="myButton" Foreground ="Black"
        FontSize ="20" FontWeight ="Bold"
        Template ="{StaticResource RoundButtonTemplate}"
        Click ="myButton_Click"/>
```

При желании значения базовых свойств можно устанавливать *в шаблоне*. По существу таким способом создаются стандартный внешний вид и поведение. Как вам уже должно быть понятно, эта работа предназначена для стилей WPF. Когда строится стиль (для учета настроек базовых свойств), можно определить шаблон *внутри стиля*! Ниже показан измененный ресурс приложения внутри файла `App.xaml`, которому назначен ключ `RoundButtonSyle`:

```
<!-- Стиль, содержащий шаблон -->
<Style x:Key ="RoundButtonStyle" TargetType ="Button">
  <Setter Property ="Foreground" Value ="Black"/>
  <Setter Property ="FontSize" Value ="14"/>
  <Setter Property ="FontWeight" Value ="Bold"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="100"/>
  <!-- Далее следует сам шаблон -->
  <Setter Property ="Template">
    <Setter.Value>
      <ControlTemplate TargetType ="Button">
        <Grid x:Name="controlLayout">
          <Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}"/>
          <Label x:Name="buttonCaption" Content ="{TemplateBinding Content}"
                 HorizontalAlignment="Center" VerticalAlignment="Center" />
        </Grid>
        <ControlTemplate.Triggers>
          <Trigger Property = "IsMouseOver" Value = "True">
    <Setter TargetName = "buttonSurface" Property = "Fill" Value = "Blue"/>
            <Setter TargetName = "buttonCaption" Property = "Foreground"
                    Value = "Yellow"/>
          </Trigger>
          <Trigger Property = "IsPressed" Value="True">
            <Setter TargetName="controlLayout"
                    Property="RenderTransformOrigin" Value="0.5,0.5"/>
            <Setter TargetName="controlLayout" Property="RenderTransform">
```

```
            <Setter.Value>
              <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
            </Setter.Value>
          </Setter>
        </Trigger>
      </ControlTemplate.Triggers>
    </ControlTemplate>
  </Setter.Value>
 </Setter>
</Style>
```

После такого обновления кнопочные элементы управления можно создавать с установкой свойства `Style` следующим образом:

```
<Button x:Name="myButton" Background="Red" Content="Howdy!"
        Click="myButton_Click" Style="{StaticResource RoundButtonStyle}"/>
```

Несмотря на то что внешний вид и поведение кнопки остаются теми же, преимущество внедрения шаблонов внутрь стилей связано с тем, что появляется возможность предоставить готовый набор значений для общих свойств. На этом обзор применения Visual Studio и инфраструктуры триггеров при построении специальных шаблонов для элемента управления завершен. Хотя об инфраструктуре WPF можно еще много чего сказать, теперь у вас имеется хороший фундамент для дальнейшего самостоятельного изучения.

Исходный код. Проект `ButtonTemplate` доступен в подкаталоге `Chapter_29`.

Резюме

Первой в главе рассматривалась система управления ресурсами WPF. Мы начали с исследования работы с двоичными ресурсами и роли объектных ресурсов. Вы узнали, что объектные ресурсы — это именованные фрагменты разметки XAML, которые могут быть сохранены в разнообразных местах с целью многократного использования этого содержимого.

Затем был описан API-интерфейс анимации WPF. В приведенных примерах анимация создавалась с помощью кода C#, а также посредством разметки XAML. Для управления выполнением анимации, определенной в разметке, применяются элементы `<Storyboard>` и триггеры. Далее был продемонстрирован механизм стилей WPF, который интенсивно использует графику, объектные ресурсы и анимацию.

После этого исследовалось отношение между логическим и визуальным деревьями. В своей основе логическое дерево является однозначным соответствием разметке, которая создана для описания корневого элемента WPF. Позади этого логического дерева находится гораздо более глубокое визуальное дерево, содержащее детальные инструкции визуализации.

Кроме того, была изучена роль стандартного шаблона. Не забывайте, что при построении специальных шаблонов вы в действительности заменяете все визуальное дерево (или его часть) элемента управления специальной реализацией.

ГЛАВА 30

Уведомления, команды, проверка достоверности и MVVM

Настоящая глава завершает исследование программной модели WPF представлением шаблона проектирования "Наблюдатель" (Observer) и системы уведомлений, встроенной в инфраструктуру WPF. Вы также расширите свои знания команд и проверки достоверности. Эти три элемента являются основой для реализации шаблона "модель-представление-модель представления" (Model View ViewModel — MVVM) в WPF.

Первым делом мы рассмотрим наблюдаемые модели и коллекции. Когда классы и коллекции реализуют интерфейсы `INotifyPropertyChanged` и `INotifyCollectionChanged` (соответственно), диспетчер привязки поддерживает значения пользовательского интерфейса в синхронизированном состоянии со связанными данными. За счет того, что значения пользовательского интерфейса точно отображают текущее состояние данных, значительно улучшается его восприятие конечными пользователями и сокращается объем ручного кодирования, которое требуется для достижения тех же результатов в более старых технологиях (таких как Windows Forms).

Во время разработки на основе шаблона проектирования "Наблюдатель" вы будете исследовать механизмы добавления проверки достоверности в свои приложения. Проверка достоверности — это жизненно важная часть любого приложения, которая позволяет не только сообщать пользователю о том, что что-то пошло не так, но и указывать, в *чем именно* заключается проблема. Вы также научитесь встраивать проверку достоверности в разметку представления для информирования пользователя о возникающих ошибках.

Затем мы более глубоко погрузимся в систему команд WPF и создадим специальные команды для инкапсуляции программной логики почти так, как это делалось в главе 27 со встроенными командами. С созданием специальных команд связано несколько преимуществ, включая (но не ограничиваясь) возможность многократного использования кода, инкапсуляцию логики и получение более ясного кода.

Наконец, вы узнаете о шаблоне "модель-представление-модель представления" (MVVM) и о том, как весь рассмотренный материал содействует поддержке данного шаблона. На рис. 30.1 показано окно приложения, которое будет построено в этой главе.

Введение в шаблон MVVM

Прежде чем приступить к детальному исследованию уведомлений, проверки достоверности и команд в WPF, было бы неплохо пролить свет на конечную цель настоящей главы, которой является шаблон "модель-представление-модель представления" (MVVM).

Будучи производным от шаблона проектирования "Модель представления" (Presentation Model) Мартина Фаулера, шаблон MVVM задействует обсуждаемые в этой главе возможности, специфичные для XAML, чтобы сделать процесс разработки приложений WPF более быстрым и ясным. Название шаблона отражает основные его компоненты: модель (Model), представление (View) и модель представления (ViewModel).

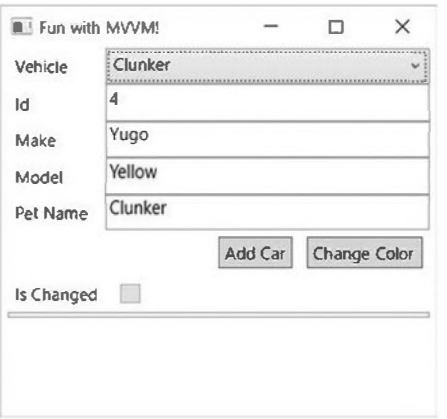


Рис. 30.1. Приложение, реализующее шаблон MVVM

Модель

Модель — это объектное представление имеющихся данных. В примерах, приводимых в главе, моделью является класс `Inventory`. Модель описывает не действительное хранилище данных (такое как SQL Server), а классы, которые *представляют* данные, хранящиеся на уровне постоянства.

Модели обычно обладают встроенной проверкой достоверности (как вы увидите в разделе "Проверка достоверности" далее в главе) и сконфигурированы как наблюдаемые классы, которые также раскрываются позже в главе.

Представление

Представление — это пользовательский интерфейс приложения, который спроектирован так, чтобы быть чрезвычайно легковесным. Вспомните о стенде меню в ресторане для автомобилистов. На стенде отображаются позиции меню и цены, а также имеется механизм взаимодействия клиента с внутренними системами. Однако в стенд не внедрены какие-либо интеллектуальные возможности, если только он не изолирован от самого ресторана, в случае чего в нем может быть предусмотрен светочувствительный датчик, который включает освещение в темное время суток.

Представления MVVM должны разрабатываться с учетом тех же целей. Любые интеллектуальные возможности должны встраиваться в какие-то другие места приложения. Иметь прямое отношение к манипулированию пользовательским интерфейсом должен только код в файле отделенного кода (например, в `MainWindow.xaml.cs`). Он не должен быть основан на бизнес-правилах или на чем-то еще, что нуждается в предохранении для будущего применения. Хотя это не является целью MVVM, приятным побочным эффектом оказывается совсем небольшой объем отделенного кода (если вообще какой-либо).

Модель представления

В WPF и других технологиях XAML модель представления служит двум целям.

- Модель представления обеспечивает единственное местоположение для всех данных, необходимых представлению. Это вовсе не означает, что модель представления отвечает за получение действительных данных; взамен она обращается к

соответствующему коду (такому как классы хранилищ EF), чтобы получить все данные вместе там, где они легко доступны. В результате между окнами и моделями представлений обычно сохраняется однозначное отношение, но архитектурные отличия существуют, и в каждом конкретном случае они могут варьироваться.

- Вторая задача модели представления касается ее действия в качестве контроллера для представления. Почти как стенд меню модель представления принимает указание от пользователя и вызывает код для выполнения подходящих действий. Такой код обычно имеет форму команд.

Анемичные модели или модели представлений

На заре развития WPF, когда разработчики все еще трудились над тем, как лучше реализовать шаблон MVVM, велись бурные (а временами и жаркие) дискуссии о том, где реализовывать элементы, подобные проверке достоверности и наблюдаемым классам. Один лагерь (сторонников анемичной (иногда называемой бескровной) модели) аргументировал, что все элементы должны находиться в моделях представлений, т.к. добавление таких возможностей к модели нарушает принцип разделения ответственности. Другой лагерь (сторонников анемичной модели представления) утверждал, что все элементы должны находиться в моделях, поскольку это сокращает дублирование кода.

Реалистичным ответом, естественно, будет "когда как". После того как интерфейсы `INotifyPropertyChanged`, `IDataErrorInfo` и `INotifyDataErrorInfo` реализованы классами моделей, это гарантирует, что соответствующий код близок к своей цели и реализован только однократно для каждой модели. Другими словами, есть ситуации, когда сами классы моделей представлений необходимо разрабатывать как наблюдаемые. По большому счету вы должны самостоятельно выяснить, что имеет больший смысл, без чрезмерного усложнения кода или принесения в жертву преимуществ MVVM.

На заметку! Для WPF доступны многочисленные инфраструктуры MVVM, такие как MVVMLite, Caliburn.Micro и Prism (хотя Prism — нечто намного большее, чем просто инфраструктура MVVM). Каждая инфраструктура характеризуется своими достоинствами и (в известной степени) недостатками. В настоящей главе обсуждается шаблон MVVM и функциональные средства WPF, которые поддерживают его реализацию. Исследование других инфраструктур и выбор среди них наиболее подходящей для нужд разрабатываемого приложения остается за вами.

Система уведомлений привязки WPF

Значительным недостатком системы привязки Windows Forms является отсутствие уведомлений. Когда данные в источнике данных изменяются, пользовательский интерфейс не обновляется автоматически. Вместо этого разработчику необходимо вызывать метод `Refresh()` элемента управления, чтобы он повторно загрузил данные из источника данных. В результате количество вызовов метода `Refresh()` может оказаться больше, чем фактически необходимо, поскольку большинство разработчиков хотят иметь гарантию того, что пользовательский интерфейс и данные не утратят синхронизацию. Вообще говоря, наличие излишних вызовов не приводит к серьезной проблеме с производительностью, но если их количество не будет достаточным, то это может отрицательно повлиять на пользовательский интерфейс.

Система привязки, встроенная в приложения на основе XAML, устраняет указанную проблему, позволяя привязывать объекты данных к системе уведомлений. Классы и коллекции, привязанные к этой системе, называются наблюдаемыми моделями и наблюдаемыми коллекциями или все вместе — наблюдаемыми объектами.

Всякий раз, когда изменяется значение свойства в наблюдаемой модели либо происходит изменение в наблюдаемой коллекции (например, добавление, удаление или переупорядочение элементов), инициируется событие (`NotifyPropertyChanged` либо `NotifyCollectionChanged`). Инфраструктура привязки автоматически прослушивает эти события и в случае их появления обновляет привязанные элементы управления. Более того, разработчики могут управлять тем, для каких свойств должны выдаваться уведомления. Выглядит безупречно, не так ли? На самом деле, как вскоре будет показано, все не *настолько* безупречно. Настройка наблюдаемых моделей вручную требует написания довольно большого объема кода. К счастью, существует инфраструктура с открытым кодом, которая значительно упрощает работу.

Наблюдаемые модели и коллекции

В этом разделе мы построим приложение, в котором используются наблюдаемые модели и коллекции. Для начала создадим новый проект WPF Application (Приложение WPF) по имени `Notifications`. В приложении будет применяться форма "главная–подробности", что позволит пользователю выбирать объект автомобиля в элементе управления `ComboBox` и просматривать детальную информацию о нем в расположенных ниже элементах `TextBox`. Поместим в файл `MainWindow.xaml` следующую разметку:

```
<Window x:Class="Notifications.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:Notifications"
    mc:Ignorable="d"
    Title="Fun with Notifications!" Height="225" Width="325"
    WindowStartupLocation="CenterOwner">
  <Grid IsSharedSizeScope="True" Margin="5,0,5,5">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid Grid.Row="0">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" SharedSizeGroup="CarLabels"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
      <Label Grid.Column="0" Content="Vehicle"/>
      <ComboBox Name="cboCars" Grid.Column="1" DisplayMemberPath="PetName" />
    </Grid>
    <Grid Grid.Row="1">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" SharedSizeGroup="CarLabels"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
      </Grid.RowDefinitions>
      <Label Grid.Column="0" Grid.Row="0" Content="Make"/>
      <TextBox Grid.Column="1" Grid.Row="0" />
```

```
      <Label Grid.Column="0" Grid.Row="1" Content="Color"/>
      <TextBox Grid.Column="1" Grid.Row="1" />
      <Label Grid.Column="0" Grid.Row="2" Content="Pet Name"/>
      <TextBox Grid.Column="1" Grid.Row="2" />
      <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="3"
          HorizontalAlignment="Right" Orientation="Horizontal"
          Margin="0,5,0,5">
        <Button x:Name="btnAddCar" Content="Add Car"
          Margin="5,0,5,0" Padding="4, 2" />
        <Button x:Name="btnChangeColor" Content="Change Color" Margin="5,0,5,0"
          Padding="4, 2"/>
      </StackPanel>
      </Grid>
    </Grid>
</Window>
```

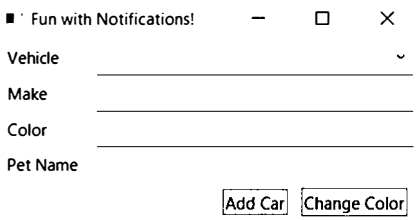


Рис. 30.2. Отображение складской информации

Окно должно напоминать показанное на рис. 30.2. Щелкнем правой кнопкой мыши на имени проекта в окне Solution Explorer, выберем в контекстном меню пункт Add⇨New Folder (Добавить⇨Новая папка), назначим новой папке имя `Models` и создадим в ней класс под названием `Inventory`. Первоначально код класса выглядит так:

```
public class Inventory
{
  public int CarId { get; set; }
  public string Make { get; set; }
  public string Color { get; set; }
  public string PetName { get; set; }
}
```

Добавление привязок и данных

Следующий шаг заключается в создании операторов привязки для элементов управления. Добавим к элементу управления `Grid`, содержащему текстовые поля и метку, свойство `DataContext` и присвоим ему свойство `SelectedItem` элемента `ComboBox`. Модифицируем определение элемента `Grid`, содержащего элементы управления с информацией об автомобиле, как показано ниже:

```
<Grid Grid.Row="1"
      DataContext="{Binding ElementName=cboCars, Path=SelectedItem}">
```

Вспомните, что если свойство `DataContext` в элементе управления не указано, то он будет искать его выше в дереве элементов. Это позволяет указывать в элементах `TextBox` только путь для привязки. Добавим соответствующие атрибуты `Text` и подходящие привязки к элементам управления `TextBox`:

```
<TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Path=Make}" />
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=Color}" />
<TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Path=PetName}" />
```

Наконец, поместим нужные данные в элемент управления `ComboBox`. В файле `MainWindow.xaml.cs` создадим новый список записей `Inventory` и установим свойство `ItemsSource` элемента `ComboBox` в этот список. Кроме того, добавим оператор `using` для пространства имен `Notifications.Models`.

```
using Notifications.Models;
public class MainWindow : Window
{
  readonly IList<Inventory> _cars;
  public MainWindow()
  {
    InitializeComponent();
    _cars = new List<Inventory>
    {
      new Inventory {CarId=1,Color="Blue",Make="Chevy",PetName="Kit" },
      new Inventory {CarId=2,Color="Red",Make="Ford",PetName="Red Rider" },
    };
    cboCars.ItemsSource = _cars;
  }
}
```

Запустим приложение. Как видно, в поле со списком Vehicle (Автомобиль) для выбора доступны два варианта автомобилей. Выбор одного из них приводит к автоматическому заполнению текстовых полей сведениями об автомобиле (рис. 30.3). Изменим цвет одного из автомобилей, выберем другой автомобиль и затем возвратимся к автомобилю, запись о котором редактировалась. Выяснится, что новый цвет по-прежнему связан с автомобилем. В этом нет ничего примечательного; просто здесь демонстрируется мощь привязки данных XAML.

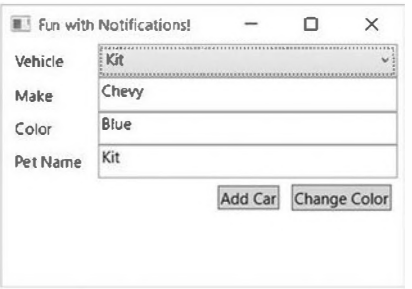


Рис. 30.3. Окно с данными об автомобиле

Изменение данных об автомобиле в коде

Хотя предыдущий пример работает ожидаемым образом, когда данные изменяются программно, пользовательский интерфейс *не* отразит изменения до тех пор, пока в приложении не будет предусмотрен код для обновления данных. Чтобы проиллюстрировать это, обработаем событие `Click` для кнопки `btnChangeColor`:

```
<Button x:Name="btnAddCar" Content="Add Car" Margin="5,0,5,0"
        Padding="4, 2" Click="btnAddCar_Click"/>
```

Создадим (или поручим создание IDE-среде Visual Studio) обработчик события `btnChangeColor_Click()`:

```
private void btnChangeColor_Click(object sender, RoutedEventArgs e)
{
}
```

Внутри обработчика события с помощью свойства `SelectedItem` элемента управления `ComboBox` выполним поиск выбранной записи в списке автомобилей. Если она найдена, то изменим цвет на розовый (`Pink`). Ниже приведен код:

```
private void btnChangeColor_Click(object sender, RoutedEventArgs e)
{
  var car = _cars.FirstOrDefault(x => x.CarId ==
            ((Inventory)cboCars.SelectedItem)?.CarId);
  if (car != null)
  {
    car.Color = "Pink";
  }
}
```

Запустим приложение, выберем автомобиль и щелкнем на кнопке Change Color (Изменить цвет). Никаких видимых изменений не произойдет. Выберем другой автомобиль и затем снова первоначальный. Теперь можно заметить обновленное значение. Для пользователя такое поведение не особенно подходит.

Теперь обработаем событие `Click` для кнопки `btnAddCar`:

```
<Button x:Name="btnAddCar" Content="Add Car" Margin="5,0,5,0"
        Padding="4,2" Click="btnAddCar_Click"/>
```

Добавим обработчик события `btnAddCar_Click()` (или позволим сделать это IDE-среде Visual Studio):

```
private void btnAddCar_Click(object sender, RoutedEventArgs e)
{
}
```

Внутри обработчика события добавим новую запись в список `Inventory`:

```
private void btnAddCar_Click(object sender, RoutedEventArgs e)
{
  var maxCount = _cars?.Max(x => x.CarId) ?? 0;
  _cars?.Add(new Inventory
      { CarId=++maxCount,Color="Yellow",Make="VW",PetName="Birdie"});
}
```

Запустим приложение, щелкнем на кнопке Add Car (Добавить автомобиль) и просмотрим содержимое элемента управления `ComboBox`. Хотя известно, что в списке имеется три автомобиля, в элементе `ComboBox` отображаются только два!

Чтобы устранить обе проблемы, мы будем использовать наблюдаемые модели и наблюдаемую коллекцию. Необходимые изменения раскрываются в последующих разделах.

Наблюдаемые модели

Проблема с тем, что изменение значения свойства модели не отображается в пользовательском интерфейсе, решается за счет реализации классом модели `Inventory` интерфейса `INotifyPropertyChanged`. Интерфейс `INotifyPropertyChanged` открывает доступ к единственному событию — `PropertyChangedEvent`. Механизм привязки XAML прослушивает это событие для каждого привязанного свойства при условии, что класс реализует интерфейс `INotifyPropertyChanged`. Вот как он определен:

```
public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}
```

Событие `PropertyChanged` принимает объектную ссылку и новый экземпляр класса `PropertyChangedEventArgs`:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Model"));
```

Первый параметр представляет собой объект, который инициирует событие. Конструктор класса `PropertyChangedEventArgs` получает в качестве параметра строку, указывающую свойство, которое было изменено и нуждается в обновлении. Когда инициировано событие, механизм привязки ищет элементы управления, связанные с именованным свойством данного объекта. В случае передачи конструктору `PropertyChangedEventArgs` значения `string.Empty` обновляются все привязанные свойства объекта.

Для управления перечнем свойств, вовлеченных в процесс автоматического обновления, необходимо генерировать событие PropertyChanged внутри блока set свойства, подлежащего обновлению. Обычно в перечень входят все свойства классов модели, но в зависимости от требований приложения некоторые свойства можно опускать. Вместо инициирования события PropertyChanged прямо в блоке set для каждого задействованного свойства распространенный подход предусматривает написание вспомогательного метода (обычно называемого OnPropertyChanged()), который генерирует событие от имени свойств обычно в базовом классе для моделей.

В версиях, предшествующих .NET 4.5, вспомогательному методу нужно было передавать строковое имя свойства. Когда имя свойства в классе изменялось, приходилось помнить о необходимости корректировки строки, переданной вспомогательному методу, иначе обновление не работало. Начиная с версии .NET 4.5, появилась возможность применять атрибут [CallerMemberName], который присваивает параметру propertyName имя метода (т.е. блока set свойства), вызывающего вспомогательный метод. Добавим в класс Inventory метод по имени OnPropertyChanged() и сгенерируем событие PropertyChangedEvent, как показано ниже:

```
internal void OnPropertyChanged([CallerMemberName] string propertyName = "")
{
  PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

Обновим каждое автоматическое свойство класса Inventory, чтобы оно имело полноценные блоки get и set, а также поддерживающее поле. В случае если значение изменилось, вызовем вспомогательный метод OnPropertyChanged(). Вот модифицированное свойство CarId:

```
private int _carId;
public int CarId
{
  get { return _carId; }
  set
  {
    if (value == _carId) return;
    _carId = value;
    OnPropertyChanged();
  }
}
```

Кроме того, понадобится внести несколько изменений в код класса Inventory, включая реализацию интерфейса INotifyPropertyChanged, инициирование события PropertyChanged и преобразование всех автоматических свойств в явные свойства с поддерживающими полями. Обновленный класс Inventory выглядит следующим образом:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
namespace Notifications.Models
{
  public class Inventory : INotifyPropertyChanged
  {
    private int _carId;
    public int CarId
    {
      get { return _carId; }
```

```
        set
        {
          if (value == _carId) return;
          _carId = value;
          OnPropertyChanged();
        }
      }
      private string _make;
      public string Make
      {
        get { return _make; }
        set
        {
          if (value == _make) return;
          _make = value;
          OnPropertyChanged();
        }
      }
      private string _color;
      public string Color
      {
        get { return _color; }
        set
        {
          if (value == _color) return;
          _color = value;
          OnPropertyChanged();
        }
      }
      private string _petName;
      public string PetName
      {
        get { return _petName; }
        set
        {
          if (value == _petName) return;
          _petName = value;
          OnPropertyChanged();
        }
      }
      public event PropertyChangedEventHandler PropertyChanged;
      internal void OnPropertyChanged([CallerMemberName] string propertyName = "")
      {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
      }
    }
}
```

Снова запустим приложение, выберем автомобиль и щелкнем на кнопке Change Color. Изменение немедленно отобразится в пользовательском интерфейсе. Первая проблема решена!

Использование операции `nameof`

В версии C# 6 появилась операция `nameof`, которая возвращает строковое имя переданного ей элемента. Ее можно применять в вызовах метода `OnPropertyChanged()` внутри блоков `set`, например:

```
private string _color;
public string Color
{
  get { return _color; }
  set
  {
    if (value == _color) return;
    _color = value;
    OnPropertyChanged(nameof(Color));
  }
}
```

Обратите внимание на то, что в случае использования операции `nameof` удалять атрибут `[CallerMemberName]` из метода `OnPropertyChanged()` не обязательно (хотя он становится излишним). В конечном счете, выбор между применением операции `nameof` или атрибута `CallerMemberName` зависит от личных предпочтений.

Наблюдаемые коллекции

Следующей проблемой, которую необходимо решить, является обновление пользовательского интерфейса, когда изменяется содержимое коллекции. Это делается за счет реализации интерфейса `INotifyCollectionChanged`. Подобно `INotifyPropertyChanged` данный интерфейс открывает доступ к единственному событию `CollectionChanged`. В отличие от `INotifyPropertyChanged` реализация интерфейса `INotifyCollectionChanged` вручную предполагает больший объем действий, чем просто вызов метода в блоке `set` свойства. Понадобится создать реализацию полного списка объектов и генерировать событие `CollectionChanged` каждый раз, когда он изменяется.

Событие `CollectionChanged` ожидает один параметр — экземпляр класса `CollectionChangedEventArgs`. Конструктор класса `CollectionChangedEventArgs` принимает один или большее число параметров в зависимости от операции. Первый параметр всегда является значением перечисления `NotifyCollectionChangedAction`, которое информирует механизм привязки о том, что конкретно изменилось в списке. Значения перечисления `NotifyCollectionChangedAction` кратко описаны в табл. 30.1.

Таблица 30.1. Значения перечисления `NotifyCollectionChangedAction`

Значение	Описание
`Add`	В коллекцию был добавлен один или более элементов
`Move`	В коллекции был перемещен один или более элементов
`Remove`	Из коллекции был удален один или более элементов
`Replace`	В коллекции был заменен один или более элементов
`Reset`	Изменений слишком много, поэтому лучше начать сначала и повторно привязать все, что относится к коллекции

Дополнительные параметры конструктора `NotifyCollectionChangedEventArgs` варьируются на основе указанного действия. В табл. 30.2 перечислены операции и дополнительные параметры, передаваемые конструктору.

Таблица 30.2. Дополнительные параметры конструктора `NotifyCollectionChangedEventArgs`

Операция	Дополнительные параметры
`Reset`	Отсутствуют
`Add` (одиночный элемент)	Элемент, подлежащий добавлению, (необязательный) индекс позиции для добавления
`Add` (список)	Элементы, подлежащие добавлению, (необязательный) индекс позиции для добавления
`Remove` (одиночный элемент)	Элемент, подлежащий удалению, (необязательный) индекс удаляемого элемента
`Remove` (список)	Элементы, подлежащие удалению, (необязательный) начальный индекс позиции для удаления
`Move` (одиночный элемент)	Элемент, подлежащий перемещению, исходный индекс, целевой индекс
`Move` (список)	Элементы, подлежащие перемещению, исходный индекс начального элемента, целевой индекс
`Replace` (одиночный элемент)	Элемент, подлежащий добавлению, элемент, подлежащий удалению, (необязательный) индекс изменения
`Replace` (список)	Элементы, подлежащие добавлению, элементы, подлежащие удалению, (необязательный) начальный индекс изменения

Итак, существует много вариаций. В следующем разделе показаны параметры, применимые к реализации `IList<T>`.

Построение специальной реализации `IList<Inventory>`

В качестве примера создадим внутри папки `Models` новый класс по имени `InventoryList`. Он должен реализовывать интерфейсы `IList<Inventory>` и `INotifyCollectionChanged`, давая в результате следующий код:

```
public class InventoryList : IList<Inventory>, INotifyCollectionChanged
{
  public IEnumerator<Inventory> GetEnumerator()
  {
    throw new System.NotImplementedException();
  }
  IEnumerator IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }
  public void Add(Inventory item)
  {
    throw new System.NotImplementedException();
  }
  public void Clear()
  {
    throw new System.NotImplementedException();
  }
  public bool Contains(Inventory item)
  {
    throw new System.NotImplementedException();
  }
```

```
  public void CopyTo(Inventory[] array, int arrayIndex)
  {
    throw new System.NotImplementedException();
  }
  public bool Remove(Inventory item)
  {
    throw new System.NotImplementedException();
  }
  public int Count { get; }
  public bool IsReadOnly { get; }
  public int IndexOf(Inventory item)
  {
    throw new System.NotImplementedException();
  }
  public void Insert(int index, Inventory item)
  {
    throw new System.NotImplementedException();
  }
  public void RemoveAt(int index)
  {
    throw new System.NotImplementedException();
  }
  public Inventory this[int index]
  {
    get { throw new System.NotImplementedException(); }
    set { throw new System.NotImplementedException(); }
  }
  public event NotifyCollectionChangedEventHandler CollectionChanged;
}
```

Начнем модификацию этого класса с добавления вспомогательного метода по имени `OnCollectionChanged()` **для события** `CollectionChanged`. **В этом методе вызовем метод** `Invoke()` **события** `CollectionChanged`, **если оно не равно** `null`:

```
public event OnCollectionChangedEventHandler CollectionChanged;
private void OnCollectionChanged(NotifyCollectionChangedEventArgs args)
{
  CollectionChanged?.Invoke(this, args);
}
```

Добавим закрытое поле по имени `_inventories` **типа** `IList<Inventory>`, **а также новый конструктор, который принимает экземпляр реализации** `IList<Inventory>` **и инициализирует это поле:**

```
private readonly IList<Inventory> _inventories;
public InventoryList(IList<Inventory> inventories)
{
  _inventories = inventories;
}
```

Добавим несколько вспомогательных методов для списка. Эти методы не изменяют состояние списка, но необходимы для реализации `IList<T>`:

```
public IEnumerator<Inventory> GetEnumerator() => _inventories.GetEnumerator();

IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

public bool Contains(Inventory item) => _inventories.Contains(item);
```

```
public void CopyTo(Inventory[] array, int arrayIndex)
{
  _inventories.CopyTo(array, arrayIndex);
}
public int Count => _inventories.Count;
public bool IsReadOnly => _inventories.IsReadOnly;
public int IndexOf(Inventory item) => _inventories.IndexOf(item);
```

Первым таким методом является `Add()`. Внутри него элемент добавляется в список `_inventories` и вызывается метод `OnCollectionChanged()`, которому передается значение `NotificationCollectionChangedAction.Add` и новый элемент, подлежащий добавлению. Ниже показан код:

```
public void Add(Inventory item)
{
  _inventories.Add(item);
  OnCollectionChanged(new NotifyCollectionChangedEventArgs(
     NotifyCollectionChangedAction.Add, item));
}
```

Метод `Insert()` добавляет запись по указанному индексу. В этом методе нужно вставить элемент в список `_inventories` по корректному индексу. Затем необходимо вызвать метод `OnCollectionChanged()`, передав ему значение `NotificationCollectionChangedAction.Add`, добавляемый новый элемент и индекс, по которому должно производиться изменение. Вот код метода `Insert()`:

```
public void Insert(int index, Inventory item)
{
  _inventories.Insert(index, item);
  OnCollectionChanged(new NotifyCollectionChangedEventArgs(
     NotifyCollectionChangedAction.Add, item, index));
}
```

Чтобы выполнить метод замены по индексу, методу `OnCollectionChanged()` следует передать значение `NotifyCollectionChangedAction.Replace` и обновляемый элемент:

```
public Inventory this[int index]
{
  get { return _inventories?[index]; }
  set
  {
    _inventories[index] = value;
    OnCollectionChanged(new NotifyCollectionChangedEventArgs(
      NotifyCollectionChangedAction.Replace, _inventories[index]));
  }
}
```

Теперь займемся методом `Remove()`. Удалим элемент из списка `_inventories` и вызовем метод `OnCollectionChanged()` с передачей ему значения `NotificationCollectionChangedAction.Remove` и элемента, подлежащему удалению. Далее приведен код:

```
public bool Remove(Inventory item)
{
  var removed = _inventories.Remove(item);
  if (removed)
  {
    OnCollectionChanged(new NotifyCollectionChangedEventArgs(
      NotifyCollectionChangedAction.Remove, item));
  }
  return removed;
}
```

Перейдем к реализации метода `RemoveAt()`. Метод `RemoveAt()` интерфейса `IList<T>` принимает только индекс, а версия конструктора `NotifyCollectionEventArgs`, которая принимала бы только индекс, отсутствует. Следовательно, сначала необходимо получить ссылку на корректный объект из списка `_inventories`, прежде чем удалять его. Затем можно удалить элемент из списка `_inventories`, используя предоставленное значение индекса. Наконец, понадобится вызвать метод `OnCollectionChanged()`, передав ему значение `NotificationCollectionChangedAction.Remove`, удаляемый элемент и индекс.

На заметку! Если не получить ссылку на корректный объект, то методу `OnCollectionChanged()` придется передавать значение `NotifyCollectionChangedEventArgs.Reset`, что может отрицательно повлиять на производительность, т.к. будет произведено обновление всех элементов управления пользовательского интерфейса, которые привязаны к данному списку.

Ниже показан код:

```
public void RemoveAt(int index)
{
  var itm = _inventories[index];
  _inventories.RemoveAt(index);
  OnCollectionChanged(new NotifyCollectionChangedEventArgs(
    NotifyCollectionChangedAction.Remove,itm,index));
}
```

Последним мы реализуем метод `Clear()`. Поскольку он полностью изменяет список, методу `OnCollectionChanged()` передается значение `NotifyCollectionChangedAction.Reset`:

```
public void Clear()
{
  _inventories.Clear();
  OnCollectionChanged(new NotifyCollectionChangedEventArgs(
     NotifyCollectionChangedAction.Reset));
}
```

Теперь, когда создан специальный класс `InventoryList`, реализующий `IList<Inventory>`, а также `INotifyCollectionChanged`, наступило время применить его в приложении. Откроем файл `MainForm.xaml.cs`, изменим тип переменной `_cars` на `InventoryList` и модифицируем конструктор для создания экземпляра `InventoryList`:

```
private readonly InventoryList _cars;
public MainWindow()
{
  InitializeComponent();
  _cars = new InventoryList(new List<Inventory>
  {
    // Свойство IsChanged должно быть указано последним в списке
    new Inventory {CarId=1,Color="Blue",Make="Chevy",PetName="Kit",
                   IsChanged = false},
    new Inventory {CarId=2,Color="Red",Make="Ford",PetName="Red Rider",
                   IsChanged = false },
  });
  cboCars.ItemsSource = _cars;
}
```

Запустим приложение, щелкнем на кнопке Add Car и удостоверимся в том, что количество автомобилей в поле со списком действительно увеличилось. Нам нужно протестировать удаление записей, поэтому добавим после кнопки Change Color еще одну кнопку. Назначим ей имя `btnRemoveCar`, установим свойство `Content` в `"Remove Car"` и укажем для события `Click` обработчик `btnRemoveCar_Click()`:

```
<Button x:Name="btnRemoveCar" Content="Remove Car" Margin="5,0,5,0"
        Padding="4,2" Click="btnRemoveCar_Click"/>
```

Создадим обработчик события (или позволим сделать это IDE-среде Visual Studio) и добавим в него вызов метода `RemoveAt()` для удаления первого элемента из списка `_cars`:

```
private void btnRemoveCar_Click(object sender, RoutedEventArgs e)
{
  _cars.RemoveAt(0);
}
```

Запустим приложение, щелкнем на кнопке Remove Car (Удалить автомобиль) и удостоверимся в том, что количество автомобилей в поле со списком действительно уменьшилось. Проблема, связанная с тем, что изменения в коллекции не приводили к обновлению пользовательского интерфейса, успешно решена, хотя и потребовала написания порядочного объема кода.

Использование класса ObservableCollection<T>

К счастью, существует намного более легкий способ, чем создание собственных коллекций. Класс `ObservableCollection<T>` является производным от класса `Collection<T>`, реализует интерфейсы `INotifyCollectionChanged` и `INotifyPropertyChanged` и входит в состав инфраструктуры. Никакой дополнительной работы выполнять не придется. Чтобы продемонстрировать его применение, добавим оператор `using` для пространства имен `System.Collections.ObjectModel`, после чего модифицируем закрытое поле `_cars` и конструктор в файле `MainWindow.xaml.cs` следующим образом:

```
readonly ObservableCollection<Inventory> _cars;
public MainWindow()
{
  InitializeComponent();
  _cars = new ObservableCollection<Inventory>
  {
    new Inventory {CarId=1,Color="Blue",Make="Chevy",PetName="Kit" },
    new Inventory {CarId=2,Color="Red",Make="Ford",PetName="Red Rider" },
  };
  cboCars.ItemsSource = _cars;
}
```

Снова запустим приложение и щелкнем на кнопках Add Car и Remove Car. Новые записи будут появляться (и исчезать) соответствующим образом.

Реализация флага изменения

Еще одним преимуществом наблюдаемых моделей является отслеживание изменений состояния. В то время как некоторые инфраструктуры объектно-реляционного отображения (ORM) вроде Entity Framework обеспечивают рудиментарное отслеживание состояния, благодаря наблюдаемым моделям отслеживание флагов изменения (когда изменено одно или более значений объекта) становится тривиальным. Добавим в класс `Inventory` свойство по имени `IsChanged` типа `bool`. Внутри его блока `set` вызовем метод `OnPropertyChanged()`, как это делалось для других свойств класса `Inventory`.

```
private bool _isChanged;
public bool IsChanged {
  get { return _isChanged;}
  set
  {
    if (value == _isChanged) return;
    _isChanged = value;
    OnPropertyChanged();
  }
}
```

Откроем файл `MainWindows.xaml` и добавим дополнительное определение `RowDefinition` в элемент управления `Grid`, который содержит текстовые поля с подробной информацией об автомобиле. Ниже показано начало определения `Grid`:

```
<Grid Grid.Row="1" DataContext="{Binding ElementName=cboCars,
Path=SelectedItem}">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" SharedSizeGroup="CarLabels"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
```

В конце этого элемента управления `Grid` добавим элементы `Label` и `CheckBox`, а затем привяжем `CheckBox` к свойству `IsChanged`:

```
<Label Grid.Column="0" Grid.Row="4" Content="Is Changed"/>
<CheckBox Grid.Column="1" Grid.Row="4" VerticalAlignment="Center"
  Margin="10,0,0,0" IsEnabled="False" IsChecked="{Binding Path=IsChanged}" />
```

Свойство `IsChanged` необходимо устанавливать в `true` каждый раз, когда изменяется другое свойство. Лучше всего это делать во вспомогательном методе `OnPropertyChanged()`, т.к. он вызывается при любом изменении свойства. Важно не устанавливать свойство `IsChanged` в `true` в случае изменения его самого, иначе сгенерируется исключение переполнения стека! Откроем файл `Inventory.cs` и модифицируем метод `OnPropertyChanged()` следующим образом (здесь используется описанная ранее операция `nameof`):

```
protected virtual void OnPropertyChanged([CallerMemberName]
          string propertyName = "")
{
  if (propertyName != nameof(IsChanged))
  {
    IsChanged = true;
  }
  PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

Если запустить приложение прямо сейчас, то окажется, что каждая отдельная запись отображается как измененная, хотя ничего пока не изменялось! Причина в том, что во время создания объекта устанавливаются значения свойств, а при установке любых значений происходит вызов метода `OnPropertyChanged()`. Это приводит к установке

свойства `IsChanged` объекта. Чтобы устранить проблему, установим свойство `IsChanged` в `false` последним в коде инициализации объекта. Откроем файл `MainWindow.xaml.cs` и модифицируем код создания списка, как показано ниже:

```
_cars = new ObservableCollection<Inventory>
{
  // Свойство IsChanged должно быть указано последним в списке
  new Inventory {CarId=1,Color="Blue",Make="Chevy",PetName="Kit",
                 IsChanged = false},
  new Inventory {CarId=2,Color="Red",Make="Ford",PetName="Red Rider",
                 IsChanged = false },
};
```

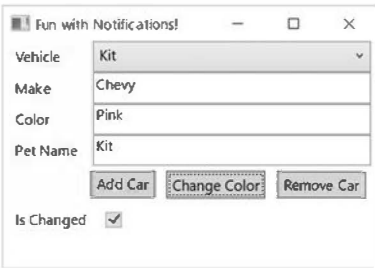


Рис. 30.4. Отображение флага изменения

Снова запустим приложение, выберем автомобиль и щелкнем на кнопке Change Color. Флажок Is Changed (Изменено) становится отмеченным наряду с изменением цвета (рис. 30.4).

Обновление источника через взаимодействие с пользовательским интерфейсом

Во время выполнения приложения можно заметить, что при вводе в текстовых полях флажок Is Changed не становится отмеченным до тех пор, пока фокус не покинет элемент управления, в котором производился ввод. Причина кроется в свойстве `UpdateSourceTrigger` привязок элементов `TextBox`. Свойство `UpdateSourceTrigger` определяет, какое событие (изменение значения, переход фокуса и т.д.) является основанием для обновления пользовательским интерфейсом лежащих в основе данных. Перечисление `UpdateSourceTrigger` принимает значения, описанные в табл. 30.3.

Таблица 30.3. Значения перечисления `UpdateSourceTrigger`

Значение	Описание
`Default`	Устанавливает стандартное значение, принятое для элемента управления (например, `LostFocus` для `TextBox`)
`Explicit`	Обновляет источник только при вызове метода `UpdateSource()`
`LostFocus`	Обновляет источник, когда элемент управления теряет фокус. Является стандартным для `TextBox`
`PropertyChanged`	Обновляет при изменении свойства. Является стандартным для `CheckBox`

Стандартным значением для элементов управления `TextBox` является `LostFocus`. Изменим его на `PropertyChanged`, модифицировав привязку для элемента `TextBox`, который отвечает за ввод цвета, следующим образом:

```
<TextBox Grid.Column="1" Grid.Row="1"
         Text="{Binding Path=Color, UpdateSourceTrigger=PropertyChanged}" />
```

Если теперь запустить приложение и начать ввод в текстовом поле Color (Цвет), то флажок Is Changed немедленно отметится. Может возникнуть вопрос о том, почему для элементов управления `TextBox` в качестве стандартного выбрано значение `LostFocus`. Дело в том, что проверка достоверности (рассматриваемая в следующем разделе) для модели запускается каждый раз, когда инициируется событие `PropertyChanged`. В случае `TextBox` это может потенциально вызывать ошибки, которые будут постоянно воз-

никать до тех пор, пока пользователь не введет корректное значение. Например, если правила проверки достоверности не разрешают вводить в элементе TextBox менее пяти символов, то сообщение об ошибке будет отображаться при каждом нажатии клавиши, пока пользователь не введет пять или более символов. В таких случаях с генерацией события PropertyChanged лучше подождать до момента, когда пользователь переместит фокус из элемента TextBox (завершив изменение текста).

Заключительное слово

Применение интерфейсов INotifyPropertyChanged и INotifyCollectionChanged улучшает пользовательский интерфейс приложения за счет поддержания его в синхронизированном состоянии с данными. В то время как ни один из интерфейсов не является сложным, они требуют обновлений кода. К счастью, в инфраструктуре предусмотрен класс ObservableCollection, поддерживающий все необходимое для создания наблюдаемых коллекций. Тем не менее, связующий код для наблюдаемых моделей придется писать самостоятельно. Хотя эта задача не отличается особой сложностью, она затрагивает блок set каждого свойства модели. Проблема может возникнуть, если модели создаются на основе существующей базы данных с использованием инфраструктуры ORM (подобной Entity Framework), т.к. они будут перезаписаны при повторной генерации классов моделей. Однако позже в главе будет описана библиотека с открытым кодом, которая позволяет решить эту проблему.

Исходный код. Проект Notifications доступен в подкаталоге Chapter_30.

Проверка достоверности

Теперь, когда интерфейс INotifyPropertyChanged реализован и задействован класс ObservableCollection, самое время заняться добавлением проверки достоверности в приложение. Приложениям необходимо проверять пользовательский ввод и обеспечивать обратную связь с пользователем, если введенные им данные оказываются некорректными. В настоящем разделе будут раскрыты наиболее распространенные механизмы проверки достоверности для современных приложений WPF, но это лишь часть возможностей, встроенных в инфраструктуру WPF.

На заметку! За полным описанием всех методов проверки достоверности в WPF обращайтесь к книге *WPF: Windows Presentation Foundation в .NET 4.5 с примерами на C# 5.0 для профессионалов* (ИД "Вильямс", 2013 г.)

Проверка достоверности происходит, когда привязка данных пытается обновить источник данных. В дополнение к встроенным проверкам, таким как исключения в блоках set для свойств, можно создавать специальные правила проверки достоверности. Если *любое* правило проверки достоверности (встроенное или специальное) нарушается, то в игру вступает класс Validation, который обсуждается позже в главе.

Модификация примера для демонстрации проверки достоверности

Для демонстрации проверки достоверности можно продолжить работу с текущим проектом. Модифицируем файл MainWindow.xaml, добавив дополнительную строку, которая содержит элементы управления Label и TextBox для свойства CarId. Лучше всего это делать, открыв визуальный конструктор окна и с помощью мыши добавив еще одну строку. Выберем нижний элемент управления Grid в окне Document Outline и поместим курсор мыши близко к его левой стороне. Щелкнем левой кнопкой мыши как можно ближе к верху элемента, где отображается линия желтого цвета (рис. 30.5).

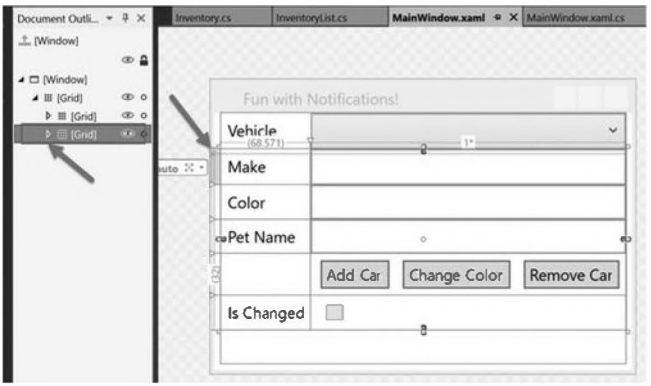


Рис. 30.5. Добавление новой строки в `Grid` посредством визуального конструктора

В элементе управления `Grid` будет создана новая строка, а номера строк в расположенных ниже элементах увеличатся на единицу. Конечно, придется еще очистить разметку и удалить поля и другие значения, вставленные визуальным конструктором, но это самый легкий способ добавления строки в верхнюю часть `Grid`. Добавим в новую строку элемент `Label` с содержимым `Id` и элемент `TextBox`, привязанный к свойству `CarId`. Ниже приведена разметка для новых элементов управления:

```
<Label Grid.Column="0" Grid.Row="0" Content="Id"/>
<TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Path=CarId}" />
```

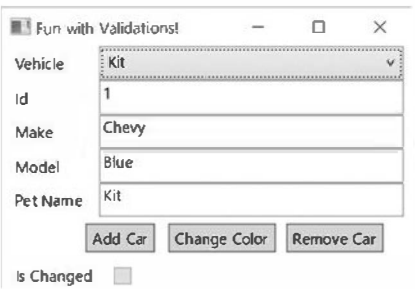


Рис. 30.6. Обновление окна для отображения `CarId`

Если запустить приложение и выбрать запись, то текстовое поле `Id` автоматически заполнится значением первичного ключа (как и ожидалось). Обновленное окно должно быть похожим на то, что показано на рис. 30.6.

Далее мы займемся исследованием процесса проверки достоверности в WPF.

Класс `Validation`

Прежде чем добавлять проверку достоверности в проект, важно понять назначение класса `Validation`. Этот класс входит в состав инфраструктуры проверки достоверности и предоставляет методы и присоединяемые свойства, которые могут применяться для отображения результатов проверки. При обработке ошибок проверки обычно используются три основных свойства класса `Validation`, кратко описанные в табл. 30.4. Они будут применяться далее в главе.

Таблица 30.4. Основные члены класса `Validation`

Член	Описание
`HasError`	Присоединяемое свойство, которое указывает, что правило проверки достоверности где-то было нарушено
`Errors`	Коллекция всех активных объектов `ValidationError`
`ErrorTemplate`	Шаблон элемента управления, который становится видимым и декорирует связанный элемент, когда свойство `HasError` установлено в `true`

Варианты проверки достоверности

Как упоминалось ранее, технологии XAML поддерживают несколько механизмов для встраивания логики проверки достоверности внутрь приложения. В последующих разделах рассматриваются три наиболее распространенных варианта проверки.

Уведомление об исключении

Хотя исключения не должны использоваться для обеспечения выполнения бизнес-логики, они могут (и будут) возникать, а потому требуют обработки подходящим образом. В случае если исключения не обработаны в коде, пользователь должен получить визуальную обратную связь об имеющейся проблеме. В отличие от Windows Forms в инфраструктуре WPF исключения привязки (по умолчанию) пользователю не отображаются, а взамен "поглощаются". Разработчики должны выбирать вариант отображения информации об ошибках.

Чтобы протестировать это, запустим приложение, выберем запись в элементе `ComboBox` и очистим значение в текстовом поле `Id`. Вспомните, что добавленное ранее свойство `CarId` определено как имеющее тип `int` (не тип `int`, допускающий `null`), поэтому требуется числовое значение. После покидания поля `Id` механизм привязки отправляет свойству `CarId` пустую строку, но из-за того, что пустая строка не может быть преобразована в значение `int`, внутри блока `set` генерируется исключение. Тем не менее, если еще не было решено отображать исключения, то никакое указание о возникшей ошибке пользователю не выдается. Включить отображение исключений легко; нужно лишь добавить `ValidatesOnExceptions=true` к операторам привязки. Изменим операторы привязки в `MainWindow.xaml`, добавив `ValidatesOnExceptions=true`:

```
<TextBox Grid.Column="1" Grid.Row="0"
  Text="{Binding Path=CarId, ValidatesOnExceptions=True}" />
<TextBox Grid.Column="1" Grid.Row="1"
  Text="{Binding Path=Make, ValidatesOnExceptions=True}" />
<TextBox Grid.Column="1" Grid.Row="2"
  Text="{Binding Path=Color, ValidatesOnExceptions=True}" />
<TextBox Grid.Column="1" Grid.Row="3"
  Text="{Binding Path=PetName, ValidatesOnExceptions=True}" />
```

Снова запустим приложение, выберем автомобиль и очистим текстовое поле `Id`. После покидания поля `Id` вокруг него появится прямоугольник красного цвета. Поскольку операторы привязки были модифицированы для проверки достоверности при исключениях, текстовое поле `Id` декорируется прямоугольником красного цвета, свидетельствующим об ошибке (рис. 30.7).

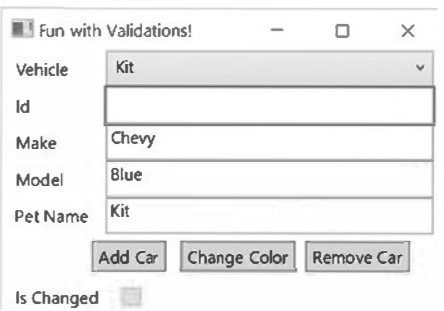


Рис. 30.7. Стандартный шаблон отображения ошибки

Прямоугольник красного цвета — это свойство `ErrorTemplate` объекта `Validation`, которое действует в качестве декоратора для связанного элемента управления. Несмотря на то что стандартный внешний вид говорит о наличии ошибки, нет никакого указания на то, *что именно* пошло не так. Хорошая новость в том, что шаблон отображения ошибки в свойстве `ErrorTemplate` является полностью настраиваемым, как вы увидите позже в главе.

Интерфейс IDataErrorInfo

Интерфейс IDataErrorInfo предоставляет механизм для добавления проверки достоверности в классы моделей. Так как данный интерфейс реализуется классами моделей напрямую и код проверки помещается внутрь классов моделей (или в частичные классы), это помогает сократить повторяющийся код проверки в рамках проекта. Скажем, в Windows Forms проверка достоверности обычно делалась в самом пользовательском интерфейсе, т.е. каждая форма, работающая с классом Inventory (в рассматриваемом примере), должна была иметь тот же самый код проверки. Когда правила изменялись, приходилось модифицировать каждую такую форму. Тот факт, что реализация может находиться в отдельном файле как частичный класс, предотвращает перезаписывание кода при воссоздании моделей на основе существующей базы данных (см. главу 23).

Показанный ниже интерфейс IDataErrorInfo содержит два свойства: индексатор и строковое свойство по имени Error. Следует отметить, что механизм привязки WPF не применяет свойство Error.

```
public interface IDataErrorInfo
{
  string this[string columnName] { get; }
  string Error { get; }
}
```

Вскоре мы добавим частичный класс Inventory, но сначала необходимо модифицировать класс Inventory.cs и пометить его как частичный. Добавим в папку Models еще один класс по имени InventoryPartial.cs. Переименуем его в Inventory, пометим как partial и реализуем интерфейс IDataErrorInfo. Затем реализуем члены этого интерфейса. Вот начальный код:

```
public partial class Inventory : IDataErrorInfo
{
  public string this[string columnName]
  {
    get { return string.Empty; }
  }
  public string Error { get; }
}
```

Индексатор вызывается каждый раз, когда объект генерирует событие PropertyChanged. В качестве параметра columnName индексатора используется имя свойства из события. Если индексатор возвращает string.Empty, то инфраструктура предполагает, что все проверки достоверности прошли успешно и какие-либо ошибки отсутствуют. Если индексатор возвращает значение, отличающееся от string.Empty, то в свойстве для данного объекта существует ошибка, из-за чего каждый элемент управления, привязанный к этому свойству (и специфичному экземпляру класса), считается содержащим ошибку. Одно предостережение: элемент управления будет задействован объектом Validation, только если свойство ValidatesOnDataErrors установлено в true внутри оператора привязки. В противном случае (подобно исключениям в блоке set в предыдущем примере) ошибка проверки достоверности поглощается, а пользователь не уведомляется. Если свойство ValidatesOnDataErrors установлено в true, то свойство HasError объекта Validation устанавливается в true и активизируется декоратор ErrorTemplate. Добавим простую логику проверки достоверности к индексатору в файле InventoryPartial.cs. Правила проверки элементарны:

- если Make равно ModelT, то установить сообщение об ошибке в "Too Old";
- если Make равно Chevy и Color равно Pink, то установить сообщение об ошибке в $"{Make}'s don't come in {Color}".

Начнем с добавления оператора switch для каждого свойства. Чтобы избежать применения "магических" строк в операторах case, мы снова будем использовать операцию nameof. В случае сквозного прохода через оператор switch возвращается string.Empty. Вот как выглядит код:

```
public string this[string columnName]
{
  get
  {
    switch (columnName)
    {
      case nameof(CarId):
        break;
      case nameof(Make):
        break;
      case nameof(Color):
        break;
      case nameof(PetName):
        break;
    }
    return string.Empty;
  }
}
```

Далее добавим правила проверки достоверности. В подходящих операторах case реализуем проверку значения свойства на основе приведенных выше правил. В операторе case для свойства Make первым делом проверим, равно ли значение ModelT. Если это так, то возвращается сообщение об ошибке. В противном случае вызывается вспомогательный метод, который возвращает сообщение об ошибке, если нарушено второе правило, или string.Empty, если нет. В операторе case для свойства Color просто вызовем тот же вспомогательный метод. Ниже показан код:

```
public string this[string columnName]
{
  get
  {
    switch (columnName)
    {
      case nameof(CarId):
        break;
      case nameof(Make):
        if (Make == "ModelT")
        {
          return "Too Old";
        }
        return CheckMakeAndColor();
      case nameof(Color):
        return CheckMakeAndColor();
      case nameof(PetName):
        break;
    }
    return string.Empty;
  }
}

internal string CheckMakeAndColor()
{
```

```
    if (Make == "Chevy" && Color == "Pink")
    {
      return $"{Make}'s don't come in {Color}";
      // AddError(nameof(Color), $"{Make}'s don't come in {Color}");
      // hasError = true;
    }
    return string.Empty;
  }
```

В качестве финального шага обновим операторы привязки, чтобы включить `ValidatesOnDataErrors=true`:

```
<TextBox Grid.Column="1" Grid.Row="0"
   Text="{Binding Path=CarId, ValidatesOnExceptions=True,
          ValidatesOnDataErrors=True}" />
<TextBox Grid.Column="1" Grid.Row="1"
   Text="{Binding Path=Make, ValidatesOnExceptions=True,
          ValidatesOnDataErrors=True}" />
<TextBox Grid.Column="1" Grid.Row="2"
   Text="{Binding Path=Color, ValidatesOnExceptions=True,
          ValidatesOnDataErrors=True}" />
<TextBox Grid.Column="1" Grid.Row="3"
   Text="{Binding Path=PetName, ValidatesOnExceptions=True,
          ValidatesOnDataErrors=True}" />
```

Запустим приложение, выберем автомобиль `Red Rider` (`Ford`) и изменим значение в поле Make (Производитель) на `ModelT`. После того, как фокус покинет поле, появится декоратор ошибки красного цвета. Выберем в поле со списком автомобиль `Kit` (`Chevy`) и щелкнем на кнопке Change Color, чтобы изменить его цвет на `Pink`. Вокруг поля Color незамедлительно появится декоратор ошибки красного цвета, но возле поля Make он будет отсутствовать. Изменим значение в поле Make на `Ford` и переместим фокус из этого поля; декоратор ошибки красного цвета *не* появляется!

Причина в том, что индексатор выполняется, только если для свойства сгенерировано событие `PropertyChanged`. Как обсуждалось в разделе "Система уведомлений привязки WPF", событие `PropertyChanged` инициируется, когда изменяется исходное значение свойства объекта, а это происходит либо посредством кода (вроде обработчика события `Click` для кнопки Change Color), либо через взаимодействие с пользователем (это синхронизируется с помощью `UpdateSourceTrigger`). При изменении цвета свойство `Make` не изменяется, поэтому событие `PropertyChanged` для него не генерируется. Поскольку событие отсутствует, индексатор не вызывается и проверка достоверности для свойства `Make` не выполняется.

Решить проблему можно двумя путями. Первый предусматривает изменение объекта `PropertyChangedEventArgs`, которое обеспечит обновление всех привязанных свойств, за счет передачи его конструктору значения `string.Empty` вместо имени поля. Как упоминалось ранее, это заставит механизм привязки обновить *каждое* свойство в данном экземпляре. Модифицируем метод `OnPropertyChanged()` следующим образом:

```
protected virtual void OnPropertyChanged([CallerMemberName]
          string propertyName = "")
{
  if (propertyName != nameof(IsChanged))
  {
    IsChanged = true;
  }
  // PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
  PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(string.Empty));
}
```

Теперь при проведении того же самого теста текстовые поля Make и Color декорируются с помощью шаблона отображения ошибки, когда одно из них обновляется. Так почему бы ни генерировать событие всегда в такой манере? В значительной степени это связано с вопросом производительности. Вполне *возможно*, что обновление каждого свойства объекта приведет к снижению производительности. Разумеется, без тестирования об этом утверждать нельзя, и конкретные ситуации могут (и, скорее всего, будут) варьироваться.

Другое решение предполагает генерацию события `PropertyChanged` для зависимого поля (полей), когда одно из полей изменяется. Недостаток применения такого приема в том, что вы (или другие разработчики, сопровождающие ваше приложение) должны знать, что в классе `InventoryPartial.cs` свойства `Make` и `Color` связаны через код проверки достоверности. Добавим показанные ниже вызовы метода `OnPropertyChanged()` в блоки `set` для свойств `Make` и `Color`:

```
private string _make;
public string Make
{
  get { return _make; }
  set
  {
    if (value == _make) return;
    _make = value;
    OnPropertyChanged(nameof(Make));
    OnPropertyChanged(nameof(Color));
  }
}
private string _color;
public string Color
{
  get { return _color; }
  set
  {
    if (value == _color) return;
    _color = value;
    OnPropertyChanged(nameof(Color));
    OnPropertyChanged(nameof(Make));
  }
}
```

Снова запустим приложение, выберем автомобиль `Chevy` и щелкнем на кнопке Change Color. Текстовые поля Make и Color получат декоратор ошибки. Изменим значение в поле Make на что-то другое кроме `Chevy` и после перехода фокуса с этого поля декоратор ошибки исчезает с обоих текстовых полей.

Интерфейс INotifyDataErrorInfo

Интерфейс `INotifyDataErrorInfo`, появившийся в версии .NET 4.5, построен на основе интерфейса `IDataErrorInfo` и предлагает дополнительные возможности для проверки достоверности. Конечно, возросшая мощь сопровождается дополнительной работой! По разительному контрасту с предшествующими приемами проверки достоверности, которые вы видели до сих пор, свойство привязки `ValidatesOnNotifyDataErrors` имеет стандартное значение `true`, поэтому добавлять его к операторам привязки не обязательно.

Интерфейс `INotifyDataErrorInfo` чрезвычайно мал, но требует написания значительного объема связующего кода, как вскоре будет показано. Вот этот интерфейс:

```
public interface INotifyDataErrorInfo
{
  bool HasErrors { get; }
  event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
  IEnumerable GetErrors(string propertyName);
}
```

Свойство `HasErrors` используется механизмом привязки для выяснения, имеются ли *любые* ошибки в любых свойствах экземпляра, которые являются источниками привязки. Если метод `GetErrors()` вызывается со значением `null` или пустой строкой в параметре `propertyName`, то он возвращает все ошибки, существующие в экземпляре. Если методу передан параметр `propertyName`, то возвращаются только ошибки для конкретного свойства. Событие `ErrorsChanged` (подобно событиям `PropertyChanged` и `CollectionChanged` events) уведомляет механизм привязки о необходимости обновления пользовательского интерфейса.

Реализация поддерживающего кода

Реализация сопряжена с написанием порядочного объема связующего кода. Хорошая новость в том, что весь этот код может быть помещен в базовый класс модели и пишется только один раз. Начнем с замены `IDataErrorInfo` интерфейсом `INotifyDataErrorInfo` в классе `InventoryPartial.cs` (код для `IDataErrorInfo` в классе можно оставить; он не будет мешать).

После добавления реализации членов интерфейса добавим закрытую переменную, которая будет хранить любые ошибки. Ниже приведен текущий код:

```
private readonly Dictionary<string,List<string>> _errors =
     new Dictionary<string, List<string>>();
public IEnumerable GetErrors(string propertyName)
{
  throw new NotImplementedException();
}
public bool HasErrors
{
  get
  {
    throw new NotImplementedException();
  }
}
public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
```

Свойство `HasErrors` должно возвращать `true`, если в словаре присутствуют *любые* ошибки. Этого легко достичь следующим образом:

```
public bool HasErrors => _errors.Count != 0;
```

Создадим вспомогательный метод, предназначенный для генерации события `ErrorsChanged` (как это делалось ранее для события `PropertyChanged`):

```
private void OnErrorsChanged(string propertyName)
{
  ErrorsChanged?.Invoke(this, new DataErrorsChangedEventArgs(propertyName));
}
```

Метод `GetErrors()` должен возвращать любые ошибки в словаре, когда в параметре передается пустая строка или `null`. Если передается допустимое значение `propertyName`, то возвращаются ошибки, обнаруженные для этого свойства. Если параметр не соответствует какому-либо свойству (или ошибки для свойства отсутствуют), то метод возвратит `null`.

```
public IEnumerable GetErrors(string propertyName)
{
  if (string.IsNullOrEmpty(propertyName))
  {
    return _errors.Values;
  }
  return _errors.ContainsKey(propertyName) ? _errors[propertyName] : null;
}
```

Последний набор вспомогательных методов будет добавлять одну или большее число ошибок для свойства либо очищать все ошибки для свойства. Каждый раз, когда словарь изменяется, не следует забывать о вызове вспомогательного метода `OnErrorsChanged()`:

```
protected void ClearErrors(string propertyName = "")
{
  _errors.Remove(propertyName);
  OnErrorsChanged(propertyName);
}
private void AddError(string propertyName, string error)
{
  AddErrors(propertyName,new List<string> {error});
}
private void AddErrors(string propertyName, IList<string> errors)
{
  var changed = false;
  if (!_errors.ContainsKey(propertyName))
  {
    _errors.Add(propertyName, new List<string>());
    changed = true;
  }
  errors.ToList().ForEach(x =>
  {
    if (_errors[propertyName].Contains(x)) return;
    _errors[propertyName].Add(x);
    changed = true;
  });
  if (changed)
  {
    OnErrorsChanged(propertyName);
  }
}
```

Механизм привязки прослушивает событие `ErrorsChanged` и обновляет пользовательский интерфейс, если возникает изменение в коллекции ошибок для оператора привязки. Это устраняет необходимость в генерации события `PropertyChanged` или `CollectionChanged` просто для обновления пользовательского интерфейса в случае ошибок, как делалось при реализации `IDataErrorInfo`. Кроме того, больше не нужны дополнительные вызовы для инициирования события `PropertyChanged` в блоках `set` для свойств `Make` и `Color` (чтобы обеспечить отображение ошибок, когда изменяются другие свойства). Разумеется, генерировать события `PropertyChanged` и `CollectionChanged` для пользовательского интерфейса по-прежнему придется в ответ на изменения *данных*, но не *ошибок*. Удалим дополнительные вызовы `OnPropertyChanged()` в блоках `set` для событий `Make` и `Color`:

```
private string _make;
public string Make
{
  get { return _make; }
  set
  {
    if (value == _make) return;
    _make = value;
    OnPropertyChanged(nameof(Make));
  }
}
private string _color;
public string Color
{
  get { return _color; }
  set
  {
    if (value == _color) return;
    _color = value;
    OnPropertyChanged(nameof(Color));
  }
}
```

Использование интерфейса `INotifyDataErrorInfo` для проверки достоверности

Теперь, когда реализован весь поддерживающий код, можно заняться добавлением проверки достоверности к приложению с применением `INotifyDataErrorInfo`. Несмотря на то что код для поддержки интерфейса `INotifyDataErrorInfo` уже написан, по-прежнему необходимо определить, где и когда выполнять проверку на предмет ошибок и добавлять ошибки в соответствующий список. В число мест выполнения проверки входят блоки `set` для свойств, как демонстрируется в приведенном ниже примере, упрощенном до единственной проверки на равенство свойства `Make` значению `ModelT`:

```
public string Make
{
  get { return _make; }
  set
  {
    if (value == _make) return;
    _make = value;
    if (Make == "ModelT")
    {
      AddError(nameof(Make), "Too Old");
    }
    else
    {
      ClearErrors(nameof(Make));
    }
    OnPropertyChanged(nameof(Make));
    // OnPropertyChanged(nameof(Color));
  }
}
```

Одна из проблем этого подхода связана с тем, что в случае обновления модели из существующей базы данных класс будет перезаписан, приводя к утере всего кода проверки достоверности. Даже если для генерации и обновления базы данных используется прием Code First, все равно код модели смешивается с кодом проверки достоверности, затрудняя его поддержку.

В предыдущем разделе было показано, что реализацию интерфейса IDataErrorInfo можно добавить к частичному классу, т.е. обновлять блоки set не придется. В итоге код модели становится чище, т.к. в блоках set остаются только вызовы для генерации события PropertyChanged. Комбинирование IDataErrorInfo и INotifyDataErrorInfo предоставляет дополнительные возможности для проверки достоверности из INotifyDataErrorInfo, а также отделение от блоков set, обеспечиваемое IDataErrorInfo. Добавим реализацию интерфейса IDataErrorInfo в класс Inventory, код которого находится в файле InventoryPartial.cs:

```
public partial class Inventory : IDataErrorInfo, INotifyDataErrorInfo
```

Цель применения IDataErrorInfo не в том, чтобы запускать проверку достоверности, а в том, чтобы гарантировать вызов кода проверки, который задействует INotifyDataErrorInfo, каждый раз, когда для объекта генерируется событие PropertyChanged. Поскольку интерфейс IDataErrorInfo не используется для проверки достоверности, необходимо всегда возвращать string.Empty, потому что событие ErrorsChanged теперь отвечает за уведомление механизма привязки о наличии ошибок. Модифицируем индексатор и вспомогательный метод CheckMakeAndColor() следующим образом:

```
public string this[string columnName]
{
  get
  {
    bool hasError = false;
    switch (columnName)
    {
      case nameof(CarId):
        break;
      case nameof(Make):
        hasError = CheckMakeAndColor();
        if (Make == "ModelT")
        {
          AddError(nameof(Make),"Too Old");
          hasError = true;
        }
        if (!hasError) ClearErrors(nameof(Make));
        break;
      case nameof(Color):
        hasError = CheckMakeAndColor();
        if (!hasError) ClearErrors(nameof(Color));
        break;
      case nameof(PetName):
        break;
    }
    return string.Empty;
  }
}
internal bool CheckMakeAndColor()
{
  if (Make == "Chevy" && Color == "Pink")
  {
    // return $"{Make}'s don't come in {Color}";
    AddError(nameof(Make), $"{Make}'s don't come in {Color}");
    AddError(nameof(Color), $"{Make}'s don't come in {Color}");
    return true;
  }
  return false;
}
```

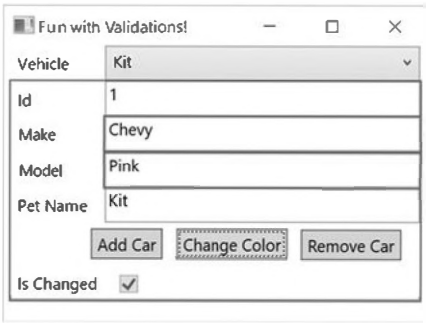


Рис. 30.8. Обновленный декоратор ошибки

Запустим приложение, выберем автомобиль `Chevy` и изменим цвет на `Pink`. В дополнение к декораторам красного цвета вокруг текстовых полей **Make** и **Model** будет также отображаться декоратор в виде красного прямоугольника, охватывающего целиком всю сетку, в которой находятся поля с детальной информацией из таблицы `Inventory` (рис. 30.8). Это еще одно преимущество применения интерфейса `INotifyDataErrorInfo`. В качестве напоминания: контекст данных установлен в элемент, выбранный в `ComboBox`, что устанавливает источник данных для `Grid` и всех его дочерних элементов управления. Когда инициируется событие `PropertyChange`, контекст данных вызывает метод `GetErrors()`, чтобы проверить, есть ли *какие-то* ошибки в экземпляре, и при их наличии активизирует объект `Validation`.

Отображение всех ошибок

Свойство `Errors` класса `Validation` возвращает все ошибки проверки достоверности для конкретного объекта. Это свойство дает список объектов `ValidationError`, каждый из которых имеет свойство `ErrorContent`, содержащее список сообщений об ошибках для каждого свойства. Поскольку действительные сообщения об ошибках, находящиеся в данном списке, необходимо отобразить внутри списка, для элемента управления `ListBox` понадобится создать элемент `DataTemplate`, содержащий `ListBox`. Звучит слегка запутанно, но вскоре все прояснится.

Для начала добавим еще одну строку в `Grid` и увеличим значение свойства `Height` элемента `Window` до 300. В последнюю строку поместим элемент управления `ListBox` и привяжем его свойство `ItemsSource` к элементу `Grid`, используя `Validation.Errors` для пути:

```
<ListBox Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="2"
   ItemsSource="{Binding ElementName=testGrid, Path=(Validation.Errors)}">
</ListBox>
```

Добавим элемент `DataTemplate`, а в него — элемент управления `ListBox`, который привязан к свойству `ErrorContent`. Контекстом данных для каждого элемента `ListBoxItem` в этом случае является объект `ValidationError`, так что устанавливать контекст данных не придется, а только путь. Установим путь привязки в `ErrorContent`:

```
<ListBox.ItemTemplate>
  <DataTemplate>
    <ListBox ItemsSource="{Binding Path=ErrorContent}"/>
  </DataTemplate>
</ListBox.ItemTemplate>
```

Ниже показана окончательная разметка:

```
<ListBox Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="2"
   ItemsSource="{Binding ElementName=testGrid, Path=(Validation.Errors)}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <ListBox ItemsSource="{Binding Path=ErrorContent}"/>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Запустим приложение, выберем автомобиль Chevy и установим цвет в Pink. В окне отобразятся ошибки (рис. 30.9).

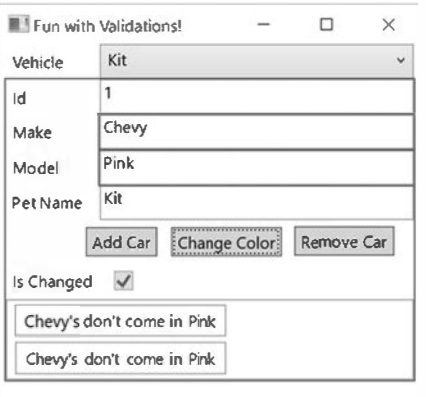


Рис. 30.9. Отображение коллекции ошибок

Перемещение поддерживающего кода в базовый класс

Как вы, вероятно, заметили, в настоящий момент в классе InventoryPartial.cs присутствует много кода. Учитывая, что в рассматриваемом примере имеется только один класс модели, в этом нет проблемы. Но по мере появления новых моделей в реальном приложении добавлять весь связующий код в каждый частичный класс для моделей нежелательно. Гораздо эффективнее поместить поддерживающий код в базовый класс, что мы и сделаем.

Создадим в папке Models новый файл класса по имени EntityBase.cs. Добавим в него операторы using для пространств имен System.Collections и System.ComponentModel. Пометим класс как открытый и реализуем интерфейс INotifyDataErrorInfor:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
namespace Validations.Models
{
  public class EntityBase : INotifyDataErrorInfo
}
```

Переместим весь относящийся к INofityDataErrorInfo код из файла InventoryPartial.cs в новый класс. Любые закрытые методы понадобится объявить как защищенные. Ниже показан модифицированный код:

```
public class EntityBase : INotifyDataErrorInfo
{
  // INotifyDataErrorInfo
  protected readonly Dictionary<string, List<string>> _errors =
     new Dictionary<string, List<string>>();
  protected void ClearErrors(string propertyName = "")
  {
    _errors.Remove(propertyName);
    OnErrorsChanged(propertyName);
  }
  protected void AddError(string propertyName, string error)
  {
    AddErrors(propertyName, new List<string> { error });
  }
  protected void AddErrors(string propertyName, IList<string> errors)
  {
    var changed = false;
    if (!_errors.ContainsKey(propertyName))
    {
      _errors.Add(propertyName, new List<string>());
      changed = true;
    }
```

```
      errors.ToList().ForEach(x =>
      {
        if (_errors[propertyName].Contains(x)) return;
        _errors[propertyName].Add(x);
        changed = true;
      });
      if (changed)
      {
        OnErrorsChanged(propertyName);
      }
    }
    public IEnumerable GetErrors(string propertyName)
    {
      if (string.IsNullOrEmpty(propertyName))
      {
        return _errors.Values;
      }
      return _errors.ContainsKey(propertyName) ? _errors[propertyName] : null;
    }
    public bool HasErrors => _errors.Count != 0;
    public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

    protected void OnErrorsChanged(string propertyName)
    {
      ErrorsChanged?.Invoke(this, new DataErrorsChangedEventArgs(propertyName));
    }
  }
```

Удалим реализацию интерфейса `INotifyDataErrorInfo` **из класса** `InventoryPartial.cs` **и добавим** `EntityBase` **в качестве базового класса:**

```
public partial class Inventory : EntityBase, IDataErrorInfo
{
  // Код для краткости не показан
}
```

Теперь любые создаваемые классы моделей унаследуют весь связующий код `INotifyDataErrorInfo`**. В базовый класс можно поместить и много другого кода, в том числе код** `INotifyPropertyChanged`**, но мы займемся этим позже в главе.**

Использование аннотаций данных

В главе 23 вы узнали, что инфраструктура EF интенсивно задействует аннотации данных. В WPF их можно также применять для проверки достоверности в пользовательском интерфейсе. Давайте добавим несколько аннотаций данных к модели `Inventory`**.**

Добавление аннотаций данных

Добавим ссылку на сборку `System.ComponentModel.DataAnnotations.dll`**, откроем файл** `Inventory.cs` **и поместим в него оператор** `using` **для пространства имен** `System.ComponentModel.DataAnnotations`**. Добавим к свойствам** `CarId`, `Make` **и** `Color` **атрибут** `[Required]`**, а к свойствам** `Make`, `Color`, **and** `PetName` **— атрибут** `[StringLength(50)]`**. Атрибут** `Required` **добавляет правило проверки достоверности, которое регламентирует, что значение свойства не должно быть** `null` **(надо сказать, оно избыточно для свойства** `CarId`**, т.к. свойство не относится к типу** `int`**, допускающему** `null`**). Атрибут** `StringLength` **добавляет правило проверки достоверности, которое ограничивает длину значения свойства 30 символами. Вот сокращенное представление кода:**

```
[Required]
public int CarId

[Required, StringLength(50)]
public string Make

[Required, StringLength(50)]
public string Color

[StringLength(50)]
public string PetName
```

Контроль ошибок проверки достоверности на основе аннотаций данных

Теперь добавленные дополнительные правила проверки необходимо вовлечь в процесс проверки достоверности. В отличие от инфраструктур ASP.NET MVC и ASP.NET Web Forms (которые обе способны автоматически контролировать наличие ошибок проверки достоверности на основе аннотаций данных) в WPF это приходится делать программно.

Прежде чем добавить код для контроля ошибок проверки достоверности, есть пара объектов, которые следует обсудить. Первый из них — `ValidationContext`. Он предоставляет контекст для проверки класса на предмет ошибок проверки достоверности с использованием класса `Validator`. Класс `Validator` позволяет контролировать наличие в объекте ошибок, основанных на атрибутах, внутри `ValidationContext`.

Откроем файл `EntityBase.cs` и добавим оператор `using` для пространства имен `System.ComponentModel.DataAnnotations`. Создадим новый метод по имени `GetErrorsFromAnnotations()`. Этот метод является обобщенным, принимает в качестве параметров строковое имя свойства и значение типа `T`, а возвращает строковый массив. Метод должен быть помечен как `protected`. Ниже показана его сигнатура:

```
protected string[] GetErrorsFromAnnotations<T>(string propertyName, T value)
```

Внутри метода создадим переменную типа `List<ValidationResult>`, которая будет хранить результаты выполненных проверок достоверности. Затем создадим объект `ValidationContext` с областью действия, ограниченной именем свойства, которое передано методу. Теперь вызовем метод `Validate.TryValidateProperty()`, возвращающий значение `bool`. Если все проверки (на основе аннотаций данных) успешно проходят, то метод возвратит `true`. В противном случае он возвратит `false` и наполнит `List<ValidationResult>` возникшими ошибками. Полный код выглядит следующим образом:

```
protected string[] GetErrorsFromAnnotations<T>(string propertyName,
                                               T value)
{
  var results = new List<ValidationResult>();
  var vc = new ValidationContext(this, null, null) { MemberName = propertyName };
  var isValid = Validator.TryValidateProperty(value, vc, results);
  return (isValid)?null:Array.ConvertAll(results.ToArray(),
         o => o.ErrorMessage);
}
```

Далее можно модифицировать метод индексатора, чтобы проверять наличие любых ошибок, основанных на аннотациях данных. В случае обнаружения ошибки добавляются в коллекцию ошибок, поддерживаемую интерфейсом `INotifyDataErrorInfo`. Ниже показан обновленный код индексатора:

```
public string this[string columnName]
{
  get
  {
    string[] errors = null;
    bool hasError = false;
    switch (columnName)
    {
      case nameof(CarId):
        errors = GetErrorsFromAnnotations(nameof(CarId), CarId);
        break;
      case nameof(Make):
        hasError = CheckMakeAndColor();
        if (Make == "ModelT")
        {
          AddError(nameof(Make), "Too Old");
          hasError = true;
        }
        errors = GetErrorsFromAnnotations(nameof(Make), Make);
        break;
      case nameof(Color):
        hasError = CheckMakeAndColor();
        errors = GetErrorsFromAnnotations(nameof(Color), Color);
        break;
      case nameof(PetName):
        errors = GetErrorsFromAnnotations(nameof(PetName), PetName);
        break;
    }
    if (errors != null && errors.Length != 0)
    {
      AddErrors(columnName, errors);
      hasError = true;
    }
    if (!hasError) ClearErrors(columnName);
    return string.Empty;
  }
}
```

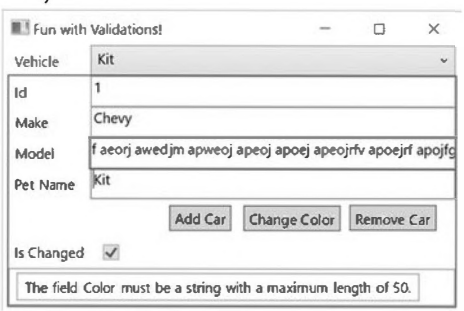


Рис. 30.10. Проверка достоверности на основе аннотаций данных

Запустим приложение, выберем один из автомобилей и введем в поле Model (Модель) текст, содержащий более 50 символов. После того, как фокус покинет поле Model, аннотация данных `StringLength` сообщит об ошибке проверки достоверности посредством метода `GetErrorsFromAnnotations()`. С помощью интерфейса `INotifyDataErrorInfo` информация об ошибках передается дальше и приводит к отображению декораторов и сообщения в элементе управления `ListBox` (рис. 30.10).

Настройка свойства `ErrorTemplate`

Финальной задачей является настройка свойства `ErrorTemplate` для элементов управления с целью отображения более значащей информации об ошибках с данными. Как объяснялось в главе 29, элементы управления допускают настройку посредством своих шаблонов элементов управления. Ранее в этой главе вы узнали, что класс `Validation` имеет свойство `ErrorTemplate`, которое применяется для декорирования элемента управления, содержащего ошибку привязки.

Начнем с добавления в раздел <Window.Resources> файла MainWindow.xaml нового стиля с целевым типом TextBox:

```
<Window.Resources>
  <Style TargetType="{x:Type TextBox}">
  </Style>
</Window.Resources>
```

Добавим к стилю триггер, который устанавливает свойства, когда свойство Validation.HasError имеет значение true. Свойствами и устанавливаемыми значениями являются Background (Pink), Foreground (Black) и Tooltip (ErrorContent). В элементах Setter для свойств Background и Foreground нет ничего нового, но Setter для свойства ToolTip требует пояснения. Привязка (Binding) указывает обратно на элемент управления TextBox, для которого этот стиль выступает как источник данных. Путь представляет собой первое значение ErrorContent в коллекции Validation.Errors. Разметка выглядит следующим образом:

```
<Style TargetType="{x:Type TextBox}">
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="true">
      <Setter Property="Background" Value="Pink" />
      <Setter Property="Foreground" Value="Black" />
      <Setter Property="ToolTip"
        Value="{Binding RelativeSource={RelativeSource Self},
        Path=(Validation.Errors)[0].ErrorContent}"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

Модифицируем свойство ErrorTemplate класса Validation,чтобы отображать восклицательный знак красного цвета, и установим для него свойство ToolTip. Поместим элемент Setter непосредственно после закрывающего дескриптора Style.Triggers внутри только что созданного стиля. Мы создадим шаблон элемента управления, состоящий из элемента TextBlock (для отображения восклицательного знака) и элемента BorderBrush, который окружает TextBox, содержащий сообщение об ошибке (или несколько таких сообщений). В языке XAML предусмотрен специальный дескриптор для элемента управления, декорированного с помощью ErrorTemplate, под названием AdornedElementPlaceholder. Добавляя имя этого элемента управления, можно получить доступ к ошибкам, которые ассоциированы с элементом управления. В данном примере нам необходим доступ к свойству Validation.Errors, чтобы получить ErrorContent (подобно тому, как это делалось в Style.Trigger). Вот полная разметка для элемента Setter:

```
<Setter Property="Validation.ErrorTemplate">
  <Setter.Value>
    <ControlTemplate>
      <DockPanel LastChildFill="True">
        <TextBlock Foreground="Red" FontSize="20" Text="!"
          ToolTip="{Binding ElementName=controlWithError,
          Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"/>
        <Border BorderBrush="Red" BorderThickness="1">
          <AdornedElementPlaceholder Name="controlWithError" />
        </Border>
      </DockPanel>
    </ControlTemplate>
  </Setter.Value>
</Setter>
```

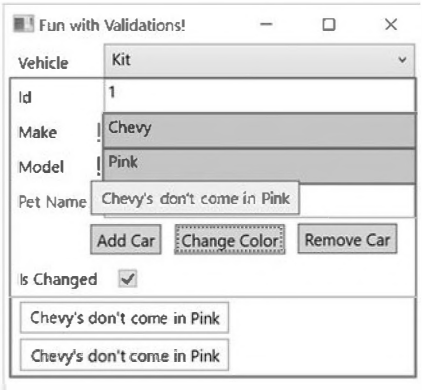


Рис. 30.11. Отображение специального свойства `ErrorTemplate`

Запустим приложение и создадим условие, при котором возникнет ошибка. Результат будет подобен представленному на рис. 30.11.

Исходный код. Проект `Validations` доступен в подкаталоге `Chapter_30`.

Создание специальных команд

Как было показано в главе 27, команды являются неотъемлемой частью WPF. Команды могут привязываться к элементам управления WPF (таким как `Button` и `MenuItem`) для обработки пользовательских событий, подобных щелчку. Вместо создания обработчика события напрямую при поступлении события выполняется метод `Execute()` команды. Метод `CanExecute()` используется для включения или отключения элемента управления на основе специального кода. В дополнение к встроенным командам, которые применялись в главе 27, можно создавать собственные команды, реализуя интерфейс `ICommand`. Использование команд взамен обработчиков событий обеспечивает преимущества инкапсуляции кода приложения, а также автоматического включения и отключения элементов управления с помощью бизнес-логики.

Реализация интерфейса `ICommand`

Вспомните из главы 27, что интерфейс `ICommand` определен следующим образом:

```
public interface ICommand
{
  event EventHandler CanExecuteChanged;
  bool CanExecute(object parameter);
  void Execute(object parameter);
}
```

Далее мы создадим команду, которая изменяет значение цвета в объекте `Inventory`. Для начала щелкнем правой кнопкой мыши на имени проекта в окне Solution Explorer и добавим новую папку под названием `Cmds`. Поместим в нее новый класс по имени `ChangeColorCommand.cs`. Класс должен реализовывать интерфейс `ICommand`, поэтому реализуем его члены и добавим оператор `using` для пространства имен `Validations.Models`. Код должен выглядеть примерно так:

```
public class ChangeColorCommand : ICommand
{
  public bool CanExecute(object parameter)
  {
    throw new NotImplementedException();
  }
  public void Execute(object parameter)
  {
    throw new NotImplementedException();
  }
  public event EventHandler CanExecuteChanged;
}
```

Параметр, передаваемый методам `CanExecute()` и `Execute()`, получается из пользовательского интерфейса через свойство `CommandParameter`, установленное в операторах привязки. Мы будем иметь дело с ним позже, а пока достаточно знать, что в рассматриваемом примере ожидается объект типа `Inventory`. Если передается `null` или объект другого типа, то метод `CanExecute()` должен возвратить `false`, и любые элементы управления, к которым привязана команда, отключаются. Если параметр не равен `null` и *является* объектом типа `Inventory`, то метод должен возвратить `true`, и любые элементы управления, к которым привязана команда, включаются. Модифицируем метод `CanExecute()`, как показано ниже:

```
public override bool CanExecute(object parameter) =>
   (parameter as Inventory) != null;
```

Поведение параметра метода `Execute()` аналогично поведению параметра метода `CanExecute()`. Метод `Execute()` запускается только в случае щелчка пользователем на элементе управления, к которому привязана команда, а пользователь может щелкать на элементе управления, только когда метод `CanExecute()` возвращает `true`. Параметр имеет тип `object`, поэтому его необходимо привести к типу `Inventory`. После приведения параметра изменим цвет автомобиля на `Pink`. Обновим метод `Execute()` следующим образом:

```
public override void Execute(object parameter)
{
  ((Inventory)parameter).Color="Pink";
}
```

Изменение файла `MainWindow.xaml.cs`

Следующее изменение связано с созданием экземпляра данного класса, к которому может иметь доступ элемент управления `Button`. В настоящий момент мы сделаем это в файле отделенного кода для `MainWindow` (позже в главе код будет перемещен в модель представления). Откроем файл `MainWindow.xaml.cs` и удалим обработчик события `Click` для кнопки Change Color, т.к. его функциональность будет заменена реализацией команды.

Добавим открытое свойство по имени `ChangeColorCmd` типа `ICommand` с поддерживающим полем. В теле выражения для свойства возвратим значение поддерживающего поля (создавая экземпляр `ChangeColorCommand`, если поддерживающее поле равно `null`):

```
private ICommand _changeColorCommand = null;
public ICommand ChangeColorCmd =>
   _changeColorCommand ?? (_changeColorCommand = new ChangeColorCommand());
```

Изменение файла `MainWindow.xaml`

Как было показано в главе 27, реагирующие на щелчки элементы управления WPF (вроде `Button`) имеют свойство `Command`, которое позволяет назначать объект команды элементу управления. После того, как объект команды присоединен к элементу управления, метод `CanExecute()` определяет, включен ли элемент управления, а событие `Click` связывается с методом `Execute()`.

Присоединим объект команды, созданный в файле отделенного кода, к кнопке `btnChangeColor`. Поскольку свойство для команды находится в классе `MainWindow`, с помощью синтаксиса привязки `RelativeSource` получается окно, содержащее нужную кнопку:

```
Command="{Binding Path=ChangeColorCmd,
  RelativeSource={RelativeSource Mode=FindAncestor,
  AncestorType={x:Type Window}}}"
```

Кнопка также нуждается в передаче объекта Inventory в качестве параметра для методов CanExecute() и Execute(). В элементах управления, реагирующих на щелчки, есть еще одно свойство по имени CommandParameter. Установим его в свойство SelectedItem элемента ComboBox с именем cboCars:

```
CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"
```

Вот завершенная разметка для кнопки:

```
<Button x:Name="btnhangeColor" Content="Change Color"
     Margin="5,0,5,0" Padding="4,2"
  Command="{Binding Path=ChangeColorCmd,
  RelativeSource={RelativeSource Mode=FindAncestor,
     AncestorType={x:Type Window}}}"
  CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
```

Присоединение команды к объекту CommandManager

Если в настоящий момент запустить приложение, то можно будет заметить, что при первой загрузке окна кнопка Change Color не включена. Ситуация вполне ожидаема, т.к. свойство SelectedItem поля со списком равно null. Поскольку именно это значение передается методам CanExectue() и Execute(), элемент управления отключен. В случае выбора записи в поле со списком разумно рассчитывать на то, что кнопка станет доступной, раз свойство SelectedItem больше не равно null. Однако кнопка по-прежнему отключена.

Причина в том, что метод CanExecute() запускается, когда окно впервые загружается, и затем, когда его запуск инициируется диспетчером команд. Каждый класс команды должен быть присоединен к диспетчеру команд. Это делается посредством события CanExecuteChanged и сводится просто к добавлению в класс ChangeColorCommand.cs следующего кода:

```
public event EventHandler CanExecuteChanged
{
  add { CommandManager.RequerySuggested += value; }
  remove { CommandManager.RequerySuggested -= value; }
}
```

Создание класса CommandBase

Приведенный выше код должен присутствовать в каждой специальной команде, которая будет строиться, поэтому для его содержания лучше создать абстрактный базовый класс. Создадим внутри папки Cmds новый класс по имени CommandBase, пометим его как абстрактный и реализуем интерфейс ICommand. Добавим оператор using для пространства имен System.Windows.Input и превратим методы Execute() и CanExecute() в абстрактные. Наконец, добавим только что написанный код события CanExecuteChanged. Ниже показана полная реализация:

```
public abstract class CommandBase : ICommand
{
  public abstract void Execute(object parameter);
  public abstract bool CanExecute(object parameter);

  public event EventHandler CanExecuteChanged
  {
    add { CommandManager.RequerySuggested += value; }
    remove { CommandManager.RequerySuggested -= value; }
  }
}
```

Изменение класса ChangeColorCommand

Откроем файл ChangeColorCommand.cs и укажем CommandBase в качестве базового класса. Удалим код события CanExecuteChanged и добавим переопределенные версии методов CanExecute() и Execute():

```
internal class ChangeColorCommand : CommandBase
{
  public override void Execute(object parameter)
  {
    ((Inventory)parameter).Color="Pink";
  }
  public override bool CanExecute(object parameter) =>
    (parameter as Inventory) != null;
}
```

Тестирование приложения

Запустим приложение. Кнопка Change Color *не* будет включенной (рис. 30.12), т.к. автомобиль еще не выбран.

Выберем автомобиль, в результате чего кнопка Change Color становится *включенной* (рис. 30.13).

Добавление оставшихся команд

Теперь, после освоения команд, можно заменить ими обработчики события Click для оставшихся двух кнопок.

Добавление команды RemoveCarCommand

Подобно ChangeColorCommand команда RemoveCarCommand имеет компонент C#, а также компонент XAML. Начнем с удаления обработчика события btnRemoveCar_Click() из класса MainWindow.xaml.cs.

Добавление класса команды

Добавим в папку Cmds новый класс по имени RemoveCarCommand, сделаем его внутренним и унаследуем от CommandBase. Данная команда будет действовать на списке записей Inventory в представлении, поэтому создадим поле _cars типа IList<Inventory> и конструктор, который принимает существующий список. Код должен выглядеть следующим образом:

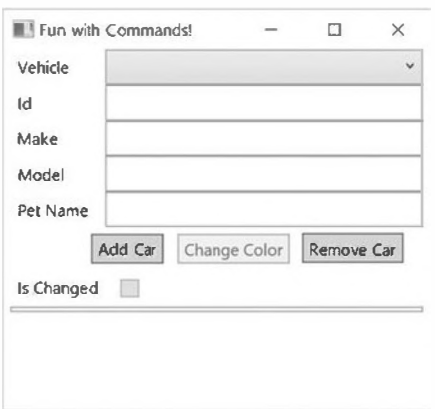


Рис. 30.12. Окно, где ничего не выбрано

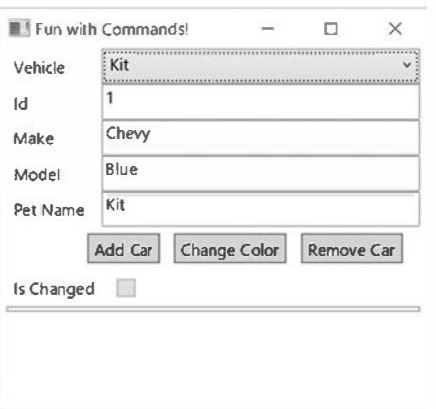


Рис. 30.13. Окно с выбранным автомобилем

```
internal class RemoveCarCommand : CommandBase
{
  private readonly IList<Inventory> _cars;
  public RemoveCarCommand(IList<Inventory> cars)
  {
    _cars = cars;
  }
}
```

Как и метод `Execute()` класса `ChangeColorCommand`, метод `Execute()` класса `RemoveCarCommand` получает в качестве параметра запись `Inventory`. Переопределим этот метод, выполним приведение параметра к типу `Inventory` и удалим его из списка:

```
public override void Execute(object parameter)
{
  _cars.Remove((Inventory)parameter);
}
```

Наконец, переопределим метод `CanExecute()` и добавим в него тот же самый код, который применялся в классе `ChangeColorCommand`: в случае, если параметр не равен `null` или не является объектом `Inventory`, возвращается значение `false`, а иначе `true`. Ниже показан код:

```
public override bool CanExecute(object parameter) =>
  (parameter as Inventory) != null && _cars != null && _cars.Count != 0;
```

Обновление разметки XAML

Удалим свойство `Click` из элемента управления `Button` по имени `btnRemoveCar` и добавим свойства `Command` и `CommandParameter`:

```
<Button x:Name="btnRemoveCar" Content="Remove Car" Margin="5,0,5,0" Padding="4,2"
  Command="{Binding Path=RemoveCarCmd,
    RelativeSource={RelativeSource Mode=FindAncestor,
    AncestorType={x:Type Window}}}"
  CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
```

Добавление команды `AddCarCommand`

Удалим обработчик события `AddCar_Click()` из файла `MainWindow.xaml.cs`.

Добавление класса команды

Добавим в папку `Cmds` еще один класс по имени `AddCarCommand`, сделаем его внутренним и унаследуем от `CommandBase`. Данная команда также будет действовать на списке записей `Inventory` в представлении, поэтому создадим поле `_cars` типа `IList<Inventory>` и конструктор, который принимает существующий список. Код должен выглядеть так:

```
internal class AddCarCommand : CommandBase
{
  private readonly IList<Inventory> _cars;
  public RemoveCarCommand(IList<Inventory> cars)
  {
    _cars = cars;
  }
}
```

В отличие от других команд, которые уже созданы, какие-либо значения из представления здесь получаться не будут. Таким образом, параметр в методах `CanExecute()` и `Execute()` можно проигнорировать. Внутри метода `Execute()` добавим запись в список `_cars`:

```
public override void Execute(object parameter)
{
  var maxCount = _cars?.Max(x => x.CarId) ?? 0;
  _cars?.Add(new Inventory { CarId = ++maxCount, Color = "Yellow",
       Make = "VW", PetName = "Birdie",
       IsChanged = false });
}
```

Переопределим метод `CanExecute()`, чтобы он просто возвращал `true`:

```
public override bool CanExecute(object parameter) => true;
```

Обновление разметки XAML

Наконец, модифицируем разметку XAML, удалив свойство `Click` и добавив свойство `Command`:

```
<Button x:Name="btnAddCar" Content="Add Car" Margin="5,0,5,0" Padding="4,2"
   Command="{Binding Path=AddCarCmd,
     RelativeSource={RelativeSource Mode=FindAncestor,
     AncestorType={x:Type Window}}}"/>
```

Исходный код. Проект `Commands` доступен в подкаталоге `Chapter_30`.

Полная реализация MVVM

Финальной задачей в рассматриваемом примере будет завершение его преобразования в MVVM. Вас может заинтересовать, почему была выбрана такая формулировка — "завершение" преобразования. Причина в том, что все действия, выполненные до сих пор в главе, относятся к тому, каким образом шаблон MVVM работает в WPF, и все они тесно переплетены внутри шаблона. Начнем с создания новой папки под названием `ViewModels`. Добавим в эту папку класс по имени `MainWindowViewModel`.

На заметку! Популярное соглашение предусматривает именование моделей представлений согласно окну, которое их поддерживает. Тем не менее, как и любой шаблон или соглашение, это не норма, и вы найдете широкий спектр мнений на этот счет.

Перемещение источника данных из представления

Вспомните из объяснения шаблона MVVM, что прямое отношение к пользовательскому интерфейсу должен иметь только код в файле отделенного кода. Любые данные, необходимые представлению, должны быть ему доступны из модели представления (и в оптимальном случае доставляться в модель представления из хранилища). В текущем проекте данные жестко закодированы в файле отделенного кода, поэтому первым делом понадобится переместить коллекцию `Cars` из файла отделенного кода в модель представления.

Начнем с добавления открытого свойства по имени `Cars` типа `IList<Inventory>`. В конструкторе класса `MainWindowViewModel` присвоим свойству новый экземпляр `ObservableCollection<Inventory>`.

На заметку! В реальном приложении для получения данных будет производиться обращение к классу хранилища или веб-службе.

Код класса должен иметь следующий вид:

```
public class MainWindowViewModel
{
  public IList<Inventory> Cars { get; set; }
  public MainWindowViewModel()
  {
    Cars = new ObservableCollection<Inventory>
    {
      new Inventory {CarId=1,Color="Blue",Make="Chevy",PetName="Kit",
        IsChanged = false},
      new Inventory {CarId=2,Color="Red",Make="Ford",PetName="Red Rider",
        IsChanged = false },
    };
  }
}
```

Удалим из файла `MainWindow.xaml.cs` старый код создания списка (находящийся в конструкторе) и присваивания списка свойству `ItemSource` элемента управления `ComboBox`. Оставим пока что поддерживающее поле (`_cars`); нам оно не нужно, но двум командам это поле требуется, и если удалить его сейчас, то код попросту не скомпилируется. В конструкторе должен остаться только вызов метода `InitializeComponent()`:

```
public MainWindow()
{
  InitializeComponent();
}
```

Вспомните, что если в выражении привязки не указан контекст данных, то оно будет подниматься вверх по дереву элементов до тех пор, пока не найдет его. В рамках шаблона MVVM класс модели представления служит контекстом данных для целого окна, поэтому присвоим контексту данных объект модели представления в конструкторе. Добавим оператор `using` для пространства имен `MVVM.ViewModels`:

```
using MVVM.ViewModels;
public partial class MainWindow:Window
{
  public MainWindow()
  {
    InitializeComponent();
    this.DataContext = new MainWindowViewModel();
  }
}
```

Последнее изменение, которое необходимо внести в элемент окна, связано с добавлением свойства `ItemSource` к элементу управления `ComboBox`. Откроем файл `MainWindow.xaml`, добавим свойство `ItemsSource` к `ComboBox` и привяжем его к свойству `Cars` модели представления. Указывать источник данных не придется, потому что модель представления является контекстом данных для окна. Разметка должна выглядеть так:

```
<ComboBox Name="cboCars" Grid.Column="1" DisplayMemberPath="PetName"
          ItemsSource="{Binding Path=Cars}"/>
```

Запустим приложение и удостоверимся в том, что поле со списком действительно заполнилось начальным содержимым коллекции `Cars`. После тестирования приложения удалим поле `_cars` из `MainWindow.xaml.cs`. Далее мы переместим команды и связанный код в модель представления.

Перемещение команд в модель представления

Вырежем код реализации команд из MainWindow.xaml.cs и вставим его в файл MainWindowViewModel.cs. Конструкторы классов AddCarCommand и RemoveCarCommand понадобится модифицировать для использования свойства Cars вместо поля _cars. Добавим оператор using для пространства имен MVVM.Cmds. Ниже показан код в классе MainWindowViewModel:

```
private ICommand _changeColorCommand = null;
public ICommand ChangeColorCmd =>
  _changeColorCommand ?? (_changeColorCommand = new ChangeColorCommand());
private ICommand _addCarCommand = null;
public ICommand AddCarCmd =>
    _addCarCommand ?? (_addCarCommand = new AddCarCommand(Cars));
private ICommand _removeCarCommand = null;
public ICommand RemoveCarCmd =>
    _removeCarCommand ?? (_removeCarCommand = new RemoveCarCommand(Cars));
private bool CanAddCar() => Cars != null;
```

Наконец, откроем файл MainWindow.xaml и добавим DataContext к пути оператора привязки Command каждого элемента управления Button:

```
<Button x:Name="cmdAddCar" Content="Add Car" Margin="5,0,5,0" Padding="4,2"
  Command="{Binding Path=DataContext.AddCarCmd,
  RelativeSource={RelativeSource Mode=FindAncestor,
  AncestorType={x:Type Window}}}"/>
<Button x:Name="cmdChangeColor" Content="Change Color"
  Margin="5,0,5,0" Padding="4,2"
  Command="{Binding Path=DataContext.ChangeColorCmd,
  RelativeSource={RelativeSource Mode=FindAncestor,
  AncestorType={x:Type Window}}}"
  CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
<Button x:Name="btnRemoveCar" Content="Remove Car" Margin="5,0,5,0" Padding="4,2"
  Command="{Binding Path=DataContext.RemoveCarCmd,
  RelativeSource={RelativeSource Mode=FindAncestor,
  AncestorType={x:Type Window}}}"
  CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
```

Запустим приложение и убедимся в том, что кнопки функционируют ожидаемым образом. Файл отделенного кода теперь содержит только две строки — для вызова InitializeComponent() и для установки DataContext.

Исходный код. Проект MVVM доступен в подкаталоге Chapter_30.

Изменение сборки AutoLotDAL для MVVM

Наступило время привнести в нашу смесь реальные данные. Прежде чем подключать сборку AutoLotDAL к приложению MVVM, необходимо предпринять несколько оптимизаций.

Изменение моделей AutoLotDAL

Первое, что понадобится сделать — модифицировать все модели, добавив код проверки достоверности, который был создан для поддержки интерфейсов IDataErrorInfo и INotifyDataErrorInfo. Поскольку большая часть этого кода уже находится в базовом классе, изменений оказывается совсем немного.

Изменение базового класса

Базовый класс был создан в предшествующих примерах. Этот базовый класс будет добавлен в сборку AutoLotDAL и в него будет внесен ряд изменений. Однако сначала нужно скопировать финальный проект AutoLotDAL из главы 23 (либо проект MVVMFinal из подкаталога Chapter_30) в каталог с прорабатываемыми примерами WPF. Щелкнем правой кнопкой на папке Models в окне Solution Explorer, выберем в контекстном меню пункт Add⇨Existing Item (Добавить⇨Существующий элемент) и укажем файл EntityBase из примера, рассмотренного в предыдущем разделе. Изменим пространство имен на AutoLotDAL.Models, а также добавим операторы using для пространств имен System.ComponentModel.DataAnnotations и System.ComponentModel.DataAnnotations.Schema. Теперь добавим свойство TimeStamp:

```
[Timestamp]
public byte[] Timestamp { get; set; }
```

Затем добавим свойство IsChanged (не забыв указать атрибут NotMapped, т.к. это свойство не должно сохраняться в базе данных):

```
[NotMapped]
public bool IsChanged { get; set; }
```

Последним изменением в классе EntityBase.cs будет реализация им интерфейса IDataErrorInfo. Метод индексатора должен быть виртуальным, потому что действительная реализация будет предоставлена в классах моделей:

```
public virtual string this[string columnName]
   { get { throw new NotImplementedException(); } }
public string Error { get; }
```

По очереди откроем каждый класс модели (Inventory, Customer, Order и CreditRisk), удалим поле TimeStamp и добавим EntityBase в качестве базового класса. В реальном приложении для каждого класса модели обычно создаются частичные классы, к которым добавляется базовый класс EntityBase. Ради простоты в текущем примере мы указываем базовый класс EntityBase в классах моделей, не создавая частичные классы.

Изменение класса InventoryPartial

Скопируем код строкового индексатора из класса InventoryPartial.cs, созданного в проекте Commands, в класс InventoryPartial.cs проекта AutoLotDAL. Он должен выглядеть так:

```
public string this[string columnName]
{
  get
  {
    string[] errors = null;
    bool hasError = false;
    switch (columnName)
    {
      case nameof(CarId):
        errors = GetErrorsFromAnnotations(nameof(CarId), CarId);
        break;
      case nameof(Make):
        hasError = CheckMakeAndColor();
        if (Make == "ModelT")
        {
          AddError(nameof(Make), "Too Old");
          hasError = true;
        }
```

```
        errors = GetErrorsFromAnnotations(nameof(Make), Make);
        break;
      case nameof(Color):
        hasError = CheckMakeAndColor();
        errors = GetErrorsFromAnnotations(nameof(Color), Color);
        break;
      case nameof(PetName):
        errors = GetErrorsFromAnnotations(nameof(PetName), PetName);
        break;
    }
    if (errors != null && errors.Length != 0)
    {
      AddErrors(columnName, errors);
      hasError = true;
    }
    if (!hasError) ClearErrors(columnName);
    return string.Empty;
  }
}
```

В итоге сборка `AutoLotDAL` получает (большей частью) возможности, о которых шла речь в настоящей главе. Пока отсутствует только реализация интерфейса `INotifyPropertyChanged`, которая будет добавлена в следующем разделе.

Реализация интерфейса `INotifyPropertyChanged`

Вы могли заметить, что мы не добавляли `INotifyPropertyChanged` в класс `EntityBase` или любые классы моделей. Если добавить `INotifyPropertyChanged` к моделям, то придется обновить каждый блок `set` для генерации события `PropertyChangedEvent` (либо вызывать `OnPropertyChanged()`). Это не особо крупная задача для мелкого проекта вроде рассматриваемого, но в проектах значительных размеров она превращается в большую проблему. Кроме того, возникнет риск утраты всей проделанной работы, если классы моделей сгенерированы инфраструктурой ORM и нуждаются в повторной генерации.

К счастью, выход из затруднительного положения имеется. Эту весьма специфическую проблему решает проект с открытым кодом под названием `PropertyChanged.Fody`. Проект является расширением Fody `https://github.com/Fody/Fody/`) — инструмента с исходным кодом для соединения сборок .NET. Соединение представляет собой процесс манипулирования кодом IL, сгенерированным во время процесса построения. Проект `PropertyChanged.Fody` добавляет связующий код для интерфейса `INotifyPropertyChanged`, и при наличии свойства по имени `IsChanged` оно будет обновлено, когда изменяется другое свойство, подобно тому, как это делалось в предшествующем примере главы.

На заметку! Дополнительные сведения о проекте `PropertyChanged.Fody` доступны по ссылке `https://github.com/Fody/PropertyChanged`.

Для установки необходимых пакетов щелкнем правой кнопкой мыши на имени проекта, выберем в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet) и отыщем пакет `propertychanged.fody`. Результирующее окно будет похоже на то, что показано на рис. 30.14 (версия пакета может отличаться).

После завершения установки откроем код класса `ModelBase` и добавим атрибут `[ImplementPropertyChanged]` на уровне класса. Вот и все, что потребовалось сделать!

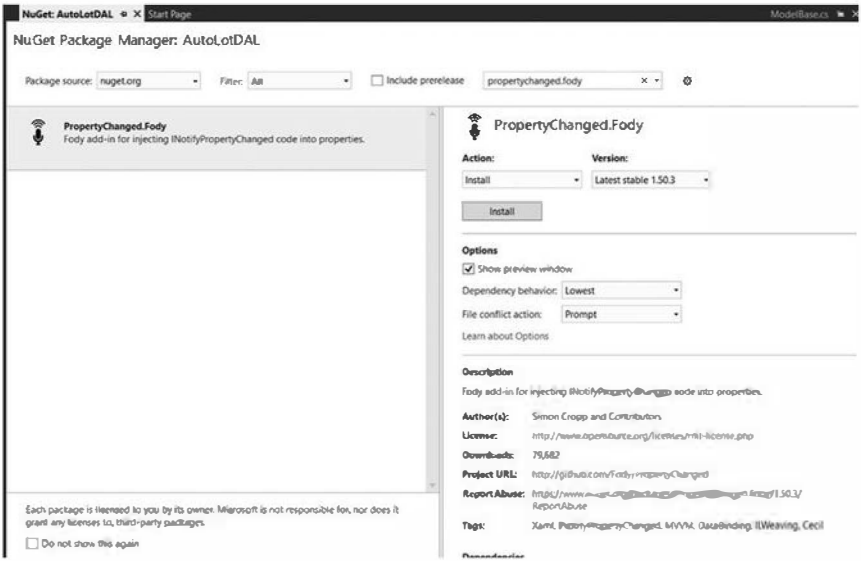


Рис. 30.14. Установка пакета `PropertyChanged.Fody`

Полный пример MVVM

После обновления сборки `AutoLotDAL` мы будем интегрировать в пример реальные данные. Скопируем содержимое каталога проекта из раздела "Полная реализация MVVM" в новое местоположение. Откроем проект и добавим сборку `AutoLotDAL`, щелкнув правой кнопкой мыши на имени проекта и выбрав в контекстном меню пункт Add⇨Existing Project (Добавить⇨Существующий проект).

Добавим ссылку на сборку `AutoLotDAL`. Щелкнем правой кнопкой мыши на имени решения, выберем в контекстном меню пункт Manage NuGet Packages и добавим в проект WPF инфраструктуру Entity Framework. Модифицируем файл `App.config`, поместив в него новый раздел `connectionStrings` с определением строки подключения к базе данных. Строка подключения может варьироваться, но должна выглядеть примерно так:

```
<connectionStrings>
 <add name="AutoLotConnection" connectionString="data source=.\SQLEXPRESS2014;
initial catalog=AutoLot;integrated security=True;
MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

Откроем файл `MainWindowViewModel.cs` и добавим операторы `using` для пространств имен `AutoLotDAL.Models` и `AutoLotDAL.Repos`. Модифицируем конструктор для получения всех записей `Inventory` из класса `InventoryRepo` вместо создания их списка вручную. Ниже приведен новый код конструктора:

```
public MainWindowViewModel()
{
  Cars = new ObservableCollection<Inventory>(new InventoryRepo().GetAll());
}
```

Откроем файлы классов команд, обновим пространства имен для применения `AutoLotDAL.Models` и удалим пространство имен `MVVM.Models` (или то, что в действительности использовалось в примере). Наконец, удалим все классы из папки `Models` проекта приложения WPF.

Запустим приложение и удостоверимся в том, что в поле со списком присутствуют все записи из базы данных (рис. 30.15). Здесь заметна одна проблема: при отображении каждой записи об автомобиле флажок Is Changed оказывается отмеченным. Причина в том, что инфраструктура EF материализует каждую запись путем установки свойств, что, конечно же, приводит к выполнению кода обработки события `PropertyChanged` и установке флага `IsChanged`. В следующем разделе объясняется, как это элегантно обработать в EF.

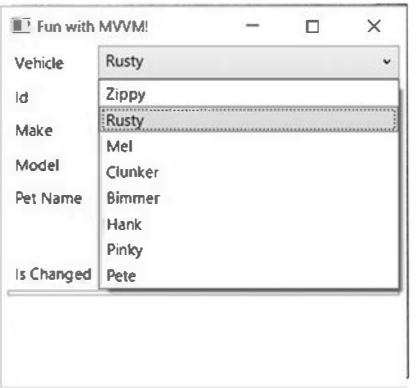


Рис. 30.15. Извлечение информации из базы данных

Использование события `ObjectMaterialized` вместе с Entity Framework

Вспомните из главы 23, что событие `ObjectMaterialized` инициируется всякий раз, когда инфраструктура EF завершает воссоздание объекта из базы данных. Откроем файл `AutoLotEntities.cs` и добавим внутрь обработчика события `OnObjectMaterialized` код для проверки, имеет ли свойство `Entity` объекта `ObjectMaterializedEventArgs` тип `EntityBase`. В таком случае свойство `IsChanged` должно быть установлено в `false`. Код выглядит следующим образом:

```
private void OnObjectMaterialized(object sender, ObjectMaterializedEventArgs e)
{
  var model = (e.Entity as EntityBase);
  if (model != null)
  {
    model.IsChanged = false;
  }
}
```

Запустим приложение. После выбора записи в поле со списком можно заметить, что флажок Is Changed больше не отмечается при загрузке окна в первый раз. Если внести изменение в какое-нибудь поле, то флажок Is Changed становится соответственно отмеченным.

Резюме

В этой главе рассматривались аспекты WPF, относящиеся к поддержке шаблона MVVM. Сначала было показано, каким образом связывать классы моделей и коллекции с помощью системы уведомлений в диспетчере привязки. Демонстрировалась реализация интерфейса `INotifyPropertyChanged` и применение наблюдаемых коллекций для обеспечения синхронизации пользовательского интерфейса и связанных с ним данных. Вы узнали, как использовать проект `PropertyChanged.Fody`, чтобы выполнять такую работу автоматически.

Вы научились добавлять код проверки достоверности к модели с применением интерфейсов `IDataErrorInfo` и `INotifyDataErrorInfo`, а также проверять наличие ошибок, основанных на аннотациях данных. Было показано, как отображать обнаруженные ошибки проверки достоверности в пользовательском интерфейсе, чтобы пользователь знал о проблеме и мог ее устранить. Наконец, вы узнали, каким образом собрать все компоненты вместе за счет добавления модели представления и очистить разметку и отделенный код пользовательского интерфейса, чтобы усилить разделение ответственности. В качестве примера была обновлена сборка `AutoLotDAL` для добавления к ней проверки достоверности и уведомлений с обеспечением очистки объектов в обработчике события `ObjectMaterialized` по мере их материализации.

ЧАСТЬ VIII

ASP.NET

В этой части

ГЛАВА 31

Введение в ASP.NET Web Forms

Все рассмотренные до сих пор примеры были либо консольными, либо настольными приложениями с графическим пользовательским интерфейсом, созданными с применением инфраструктуры WPF. В оставшихся главах книги вы узнаете, как платформа .NET облегчает построение основанных на Интернете приложений с использованием технологии под названием ASP.NET. В этой и последующих двух главах раскрывается инфраструктура ASP.NET Web Forms, а в главе 34 — ASP.NET MVC и ASP.NET Web API. Для начала будет дан краткий обзор ряда основных концепций веб-разработки (HTTP, HTML, сценарии клиентской стороны, обратные отправки) и описана роль коммерческого веб-сервера Microsoft (IIS), а также IIS Express.

На заметку! В этой и последующих двух главах рассматривается ASP.NET Web Forms — первоначальная инфраструктура для веб-разработки в .NET. В главе 34 исследуются ASP.NET MVC и ASP.NET Web API — два (относительно недавних) дополнения к семейству инфраструктур ASP.NET.

После краткого введения в оставшемся материале главы внимание будет сосредоточено на структуре модели программирования с применением веб-форм ASP.NET (включая однофайловую модель и модель отделенного кода) и функциональности базового класса `Page`. Попутно вы ознакомитесь с ролью веб-элементов управления ASP.NET, структурой каталогов веб-сайта ASP.NET, а также использованием файла `Web.config` для управления веб-сайтами во время выполнения.

Роль протокола HTTP

Веб-приложения значительно отличаются от настольных графических приложений. Первая очевидная разница связана с тем, что веб-приложение производственного уровня всегда вовлекает, по меньшей мере, две подключенные к сети машины: на одной располагается веб-сайт, а на другой просматриваются данные внутри веб-браузера. Разумеется, во время разработки вполне возможно, что единственная машина будет исполнять роль и клиента на основе браузера, и веб-сервера, который обслуживает содержимое веб-сайта. Учитывая природу веб-приложений, подключенные к сети машины должны согласовать конкретный сетевой протокол для определения способа отправки и получения данных. Сетевым протоколом, который соединяет такие машины, является HTTP (Hypertext Transfer Protocol — протокол передачи гипертекста).

Цикл "запрос/ответ" HTTP

Когда на клиентской машине запускается веб-браузер (такой как Google Chrome, Opera, Mozilla Firefox, Apple Safari или Microsoft Internet Explorer/Edge), выполняется *HTTP-запрос* для доступа к определенному ресурсу (обычно веб-странице) на удаленной серверной машине. По существу HTTP представляет собой текстовый протокол на основе стандартной парадигмы "запрос/ответ". Например, в случае перехода на `http://www.facebook.com` программное обеспечение браузера задействует веб-технологию под названием *служба доменных имен* (Domain Name Service — DNS), которая преобразует зарегистрированный URL-адрес в числовое значение, называемое *IP-адресом*. После этого браузер открывает сокетное подключение (обычно через порт 80 для незащищенных подключений) и отправляет HTTP-запрос для обработки на целевом сайте.

Веб-сервер получает входящий HTTP-запрос и может обработать любые отправленные клиентом входные значения (вроде значения внутри текстового поля, состояния флажка или переключателя), чтобы скомпоновать подходящий *HTTP-ответ*. Программисты веб-приложений могут применять любое количество серверных технологий (PHP, ASP.NET, JSP и т.д.) для динамической генерации содержимого, подлежащего встраиванию в HTTP-ответ. Затем браузер клиентской стороны визуализирует HTML-разметку, отправленную веб-сервером. На рис. 31.1 показан базовый цикл "запрос/ответ" HTTP.

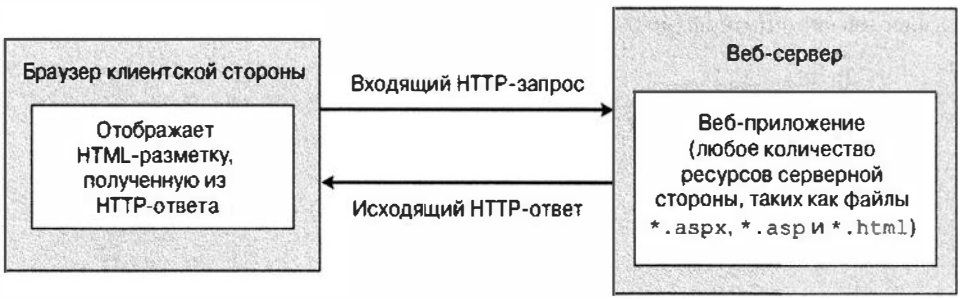


Рис. 31.1. Цикл "запрос/ответ" HTTP

HTTP является протоколом без хранения состояния

Еще один аспект веб-разработки, который заметно отличает ее от программирования традиционных настольных приложений, связан с тем фактом, что HTTP на самом деле является сетевым протоколом *без хранения состояния*. Как только веб-сервер отправляет ответ клиентскому браузеру, предыдущее взаимодействие забывается. В традиционном настольном приложении, безусловно, все не так: состояние исполняемой программы почти всегда активно до тех пор, пока пользователь не закроет главное окно приложения.

С учетом этого на разработчика веб-приложений возлагается ответственность за реализацию специфических действий по "запоминанию" важной информации (такой как элементы корзины покупок, номера кредитных карт и домашние адреса), связанной с пользователями, которые в текущий момент находятся на сайте. Как будет показано в главе 33, инфраструктура Web Forms предлагает несколько способов поддержки состояния: переменные сеанса, cookie-наборы и кеш приложения, а также API-интерфейс управления профилями Web Forms.

Веб-приложения и веб-серверы

Веб-приложение можно воспринимать как коллекцию файлов (например, `*.html`, `*.aspx`, файлов изображений, файлов данных XML) и связанных компонентов (таких как библиотека кода .NET), хранящихся в определенном наборе каталогов на веб-сервере. В главе 33 вы увидите, что приложения Web Forms обладают специфическим жизненным циклом и предоставляют многочисленные события (наподобие начального запуска или финального останова), которые можно перехватывать для выполнения специализированной обработки во время функционирования веб-сайта.

Веб-сервер — это программный продукт, отвечающий за размещение веб-приложений. Он обычно предоставляет множество взаимосвязанных служб, таких как интегрированная система безопасности, поддержка FTP (File Transfer Protocol — протокол передачи файлов), службы обмена почтой и т.д. Информационные службы Интернета (Internet Information Services — IIS) представляют собой веб-серверный продукт производственного уровня от Microsoft, который обладает встроенной поддержкой приложений Web Forms.

При наличии на рабочей станции корректно установленного сервера IIS с ним можно взаимодействовать через папку Administrative Tools (Администрирование) панели управления, дважды щелкнув на значке Internet Information Services Manager (Диспетчер служб IIS). На рис. 31.2 показан узел Default Web Site (Веб-сайт по умолчанию) в IIS, где находится большинство параметров конфигурации (в предшествующих версиях IIS пользовательский интерфейс будет выглядеть по-другому).

Рис. 31.2. Диспетчер служб IIS позволяет конфигурировать поведение Microsoft IIS во время выполнения

Роль виртуальных каталогов IIS

Единственная установленная копия IIS способна размещать многочисленные веб-приложения, каждое из которых располагается в своем *виртуальном каталоге*. Каждый виртуальный каталог отображается на физический каталог на жестком диске машины. Например, если создается новый виртуальный каталог по имени `CarsAreUs`, то извне на сайт можно перейти с использованием URL вида `http://www.MyDomain.com/CarsAreUs` (предполагая, что IP-адрес `www.MyDomain.com` был зарегистрирован в DNS).

"За кулисами" этот виртуальный каталог отображается на физический корневой каталог, в котором находится содержимое веб-приложения `CarsAreUs`.

Как будет показано далее в главе, при создании приложений Web Forms с применением Visual Studio можно заставить IDE-среду автоматически создавать новый виртуальный каталог для текущего веб-сайта. Однако при необходимости создать виртуальный каталог можно и вручную, щелкнув правой кнопкой мыши на узле Default Web Site в диспетчере служб IIS и выбрав в контекстном меню пункт Add Virtual Directory (Добавить виртуальный каталог).

Веб-сервер IIS Express

В ранних версиях платформы .NET разработчики ASP.NET обязаны были использовать виртуальные каталоги IIS во время построения и тестирования своих веб-приложений. Во многих случаях такая тесная зависимость от IIS излишне усложняла коллективную разработку, не говоря уже о том, что многие сетевые администраторы неодобрительно относились к установке IIS на машине каждого разработчика.

К счастью, теперь появился вариант в виде легковесного веб-сервера под названием IIS Express. Данная утилита позволяет разработчикам размещать приложения Web Forms за границами IIS. С применением IIS Express веб-страницы можно строить и тестировать в любом каталоге на машине. Это довольно удобно в сценариях коллективной разработки, а также при построении приложений Web Forms на машинах с версиями Windows, которые не поддерживают установку IIS.

В большинстве примеров настоящей книги вместо размещения веб-содержимого в виртуальном каталоге IIS используется веб-сервер IIS Express (через соответствующий тип проекта Visual Studio). В то время как такой подход может упростить разработку веб-приложения, имейте в виду, что этот веб-сервер *не* рассчитан на размещение веб-приложений производственного уровня. Он предназначен только для целей разработки и тестирования. Когда веб-приложение готово к эксплуатации, сайт понадобится скопировать в виртуальный каталог IIS.

На заметку! Среда Visual Studio предлагает встроенный инструмент для копирования локального веб-приложения на производственный веб-сервер. Работа сводится к паре щелчков на кнопках. Чтобы запустить процесс копирования, необходимо выбрать веб-проект в окне Solution Explorer, щелкнуть правой кнопкой мыши и выбрать в контекстном меню пункт Publish (Опубликовать). Затем можно указать желаемое местоположение для развертывания, в том числе Microsoft Azure.

Роль языка HTML

Теперь, когда сконфигурирован каталог для размещения веб-приложения и выбран веб-сервер, который будет служить хостом, понадобится создать само содержимое. Вспомните, что веб-приложение — это просто набор файлов, составляющих функциональность сайта. В действительности многие файлы в наборе будут содержать операторы HTML (Hypertext Markup Language — язык разметки гипертекста). Важно знать, что HTML является стандартным языком разметки для описания того, как литеральный текст, изображения, внешние ссылки и разнообразные элементы управления HTML должны визуализироваться внутри браузера клиентской стороны.

Хотя современные IDE-среды (включая Visual Studio) и платформы для разработки веб-приложений (такие как ASP.NET) генерируют большой объем HTML-разметки автоматически, при работе с ASP.NET вам потребуются практические знания языка HTML.

На заметку! Вспомните из главы 2, что компания Microsoft выпустила несколько бесплатных IDE-сред в виде семейства продуктов Express, а также версию Visual Studio Community Edition, которая объединяет все редакции Express в один пакет. Для проработки материала последующих глав, посвященных Web Forms, MVC и Web API, можно загрузить Visual Studio Express for Web или Visual Studio Community Edition.

В этом разделе рассматриваются основы HTML, которые способствуют пониманию разметки, генерируемой моделью программирования Web Forms.

Структура документа HTML

Типичный файл HTML состоит из набора дескрипторов, описывающих внешний вид и поведение веб-страницы. Базовая структура документа HTML имеет тенденцию оставаться той же самой. Например, файлы `*.html` начинаются и завершаются с помощью дескрипторов `<html>` и `</html>`, обычно имеют раздел `<body>` и т.д.

Начните с открытия IDE-среды Visual Studio и выбора пункта меню New⇨Project (Создать⇨Проект). В узле Other Project Types (Другие типы проектов) выберите элемент Visual Studio Solutions (Решения Visual Studio) и затем вариант Blank Solution (Пустое решение), как показано на рис. 31.3 (обратите внимание, что веб-проект в этот момент не строится, а просто создается пустое решение для хранения файлов).

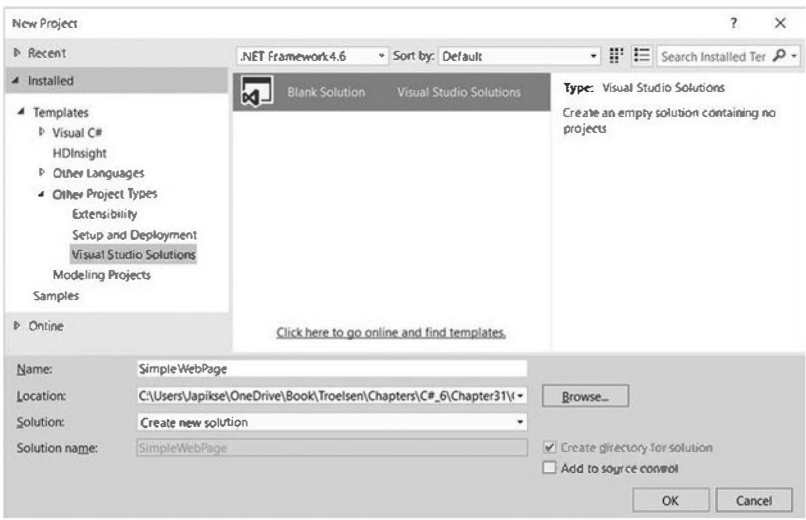


Рис. 31.3. Выбор варианта Blank Solution в диалоговом окне создания проекта

Добавьте пустой файл типа HTML Page (Страница HTML), выбрав пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент) и указав Visual C#/Web в панели слева и HTML Page в панели по центру. Назначьте файлу имя `HtmlPage1.html`. Вы должны увидеть следующую начальную разметку (в зависимости от конфигурации Visual Studio она может отличаться):

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> </title>
    <meta charset="utf-8" />
</head>
<body> </body>
</html>
```

Прежде всего, обратите внимание, что данный файл HTML начинается с инструкции обработки DOCTYPE. В сочетании с открывающим дескриптором <html> это указывает, что содержащиеся в файле дескрипторы HTML должны проверяться на соответствие стандарту HTML 5.0. Стандарт HTML 5.0 представляет собой спецификацию W3C, которая добавляет к базовому языку разметки много новых возможностей.

На заметку! По умолчанию Visual Studio проверяет все документы HTML на соответствие схеме HTML 5.0, чтобы обеспечить согласованность разметки со стандартом HTML 5. Если нужно указать альтернативную схему проверки достоверности, то откройте диалоговое окно Options (Параметры) за счет выбора пункта меню Tools⇨Options (Сервис⇨Параметры), разверните узел Text Editor (Текстовый редактор), затем узел HTML (Web Forms) и выберите узел Validation (Проверка достоверности). Если вы не хотите видеть предупреждения проверки, то просто снимите отметку с расположенного там же флажка Show Errors (Показывать ошибки).

Чтобы немного приукрасить начальную страницу, изменим ее заголовок:

```
<head>
  <title>This is my simple web page</title>
</head>
```

Дескрипторы <title> вполне ожидаемо применяются для указания текстовой строки, которая должна быть помещена в область заголовка окна веб-браузера.

Роль форм HTML

Форма HTML — это просто именованная группа связанных элементов пользовательского интерфейса, обычно используемых для сбора пользовательского ввода. Не путайте форму HTML с полной областью отображения заданного браузера. В действительности форма HTML — это в большей степени *логическая группа* графических элементов управления, помещенных между дескрипторами <form> и </form>, например:

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head>
   <title>This is my simple web page</title>
</head>
<body>
  <form id="defaultPage">
    <!-- Вставить сюда содержимое пользовательского веб-интерфейса -->
  </form>
</body>
</html>
```

Этой форме был назначен идентификатор defaultPage. Обычно открывающий дескриптор <form> содержит атрибут action, который указывает URL-адрес, применяемый для отправки данных формы, а также атрибут method, определяющий метод передачи данных (POST или GET). Вы узнаете о них в следующем разделе. А пока давайте взглянем на виды элементов, которые могут быть помещены в форму HTML (помимо обычного текста).

Инструменты визуального конструктора разметки HTML в Visual Studio

В панели инструментов (доступной через пункт меню View⇨Toolbox (Вид⇨Панель инструментов)) среды Visual Studio есть вкладка HTML, которая позволяет выбрать

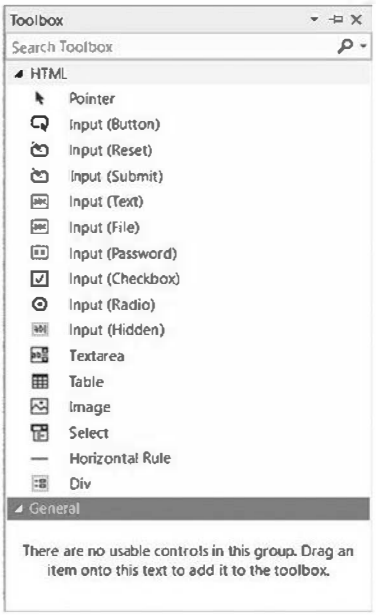


Рис. 31.4. Вкладка HTML панели инструментов

элемент управления HTML и поместить его на поверхность визуального конструктора разметки HTML (рис. 31.4). Аналогично процессу построения приложения WPF элементы управления можно перетаскивать на поверхность конструктора или прямо в разметку страницы.

На заметку! При построении веб-страниц ASP.NET с использованием модели программирования Web Forms такие элементы управления HTML обычно не применяются для создания пользовательского интерфейса. Вместо них используются элементы управления Web Forms, которые будут преобразованы в корректную разметку HTML без вашего участия. Роль веб-элементов управления обсуждается далее в главе.

Редактор HTML не имеет поверхности визуального конструктора. Для работы с визуальным конструктором (или применения режима разделения) понадобится использовать редактор HTML из Web Forms. Чтобы сделать это, закройте окно редактора для `HtmlPage1.html` и в окне Solution Explorer щелкните правой кнопкой мыши на имени файла, выберите в контекстном меню пункт Open With (Открыть с помощью), в результате чего откроется диалоговое окно, которое позволяет указать необходимый редактор (рис. 31.5). Если щелкнуть на кнопке Set as Default (Установить по умолчанию), то больше не придется создавать решение, как описано здесь, потому что всегда будет применяться редактор HTML из Web Forms.

Щелчок на кнопке Split (Разделить) внизу окна редактора HTML приводит к тому, что нижняя панель будет отображать визуальную компоновку HTML, а верхняя — связанную разметку. Еще одно преимущество этого редактора заключается в том, что при выборе разметки или элемента HTML пользовательского интерфейса соответствующее представление подсвечивается. На рис. 31.6 показан пример режима разделения в действии.

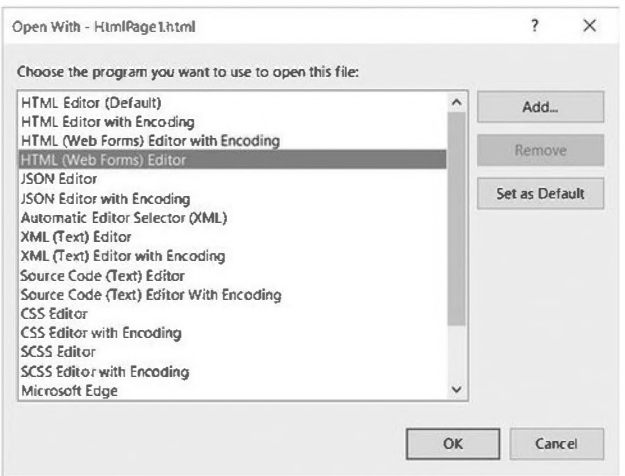


Рис. 31.5. Выбор редактора HTML из Web Forms

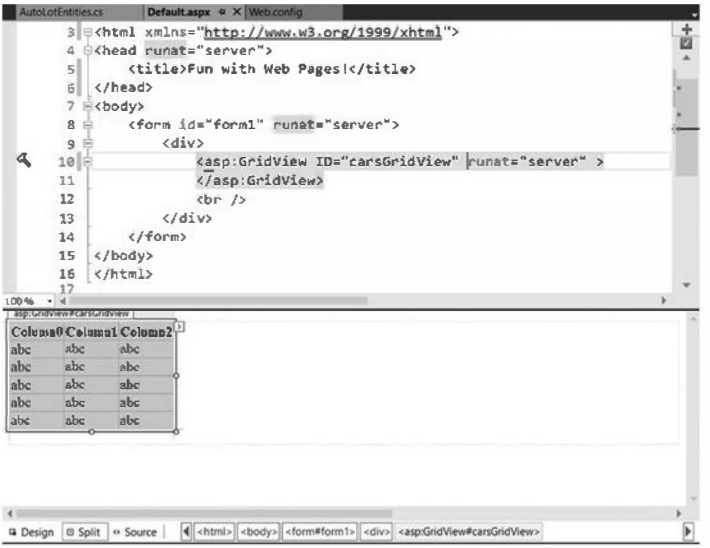


Рис. 31.6. Редактор HTML из Web Forms в Visual Studio

Среда Visual Studio также позволяет редактировать общий внешний вид и поведение страницы `*.html` или отдельного элемента управления в `<form>` с использованием окна Properties (Свойства). Например, выбрав в раскрывающемся списке окна Properties элемент `DOCUMENT`, можно конфигурировать разнообразные аспекты страницы HTML (рис. 31.7).

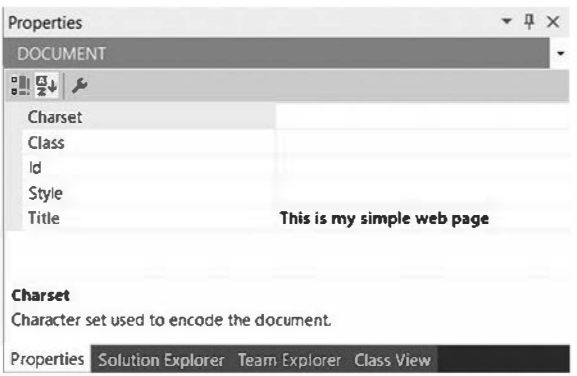


Рис. 31.7. Окно Properties в Visual Studio может применяться для конфигурирования разметки HTML

По мере использования окна Properties для конфигурирования какого-то аспекта веб-страницы IDE-среда будет соответствующим образом изменять разметку HTML. При проработке примеров в следующих главах для упрощения редактирования страниц HTML также можно применять IDE-среду.

Построение формы HTML

Модифицируйте раздел `<body>` первоначального файла, чтобы отображать литеральный текст, который приглашает пользователя ввести сообщение. Имейте в виду, что литеральное текстовое содержимое можно вводить и форматировать, набирая прямо в визуальном конструкторе разметки HTML. Здесь дескриптор `<h1>` используется для установки веса заголовка, `<p>` — для блока абзаца и `<i>` — для курсивного текста:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>This is my simple web page</title>
</head>
<body>
  <!-- Пригласить пользователя ввести текст -->
  <h1>Simple HTML Page</h1>
  <p>
    <br/>
    <i>Please enter a message</i>.
  </p>
  <form id="defaultPage">
  </form>
</body>
</html>
```

Теперь давайте создадим область ввода формы. В общем случае каждый элемент управления HTML описывается с применением атрибута `id` (для идентификации элемента в коде) и атрибута `type` (для указания того, какой элемент управления вводом должен быть помещен в объявление `<form>`).

Создаваемый пользовательский интерфейс будет содержать одно текстовое поле и две кнопки. Первая кнопка будет использоваться для запуска сценария клиентской стороны, а вторая — для сброса полей ввода формы в стандартные значения. Измените форму HTML следующим образом:

```
<!-- Построить форму для получения информации о пользователе -->
<form id="defaultPage">
  <p>
    Your Message:
    <input id="txtUserMessage" type="text"/></p>
  <p>
    <input id="btnShow" type="button" value="Show!"/>
    <input id="btnReset" type="reset" value="Reset"/>
  </p>
</form>
```

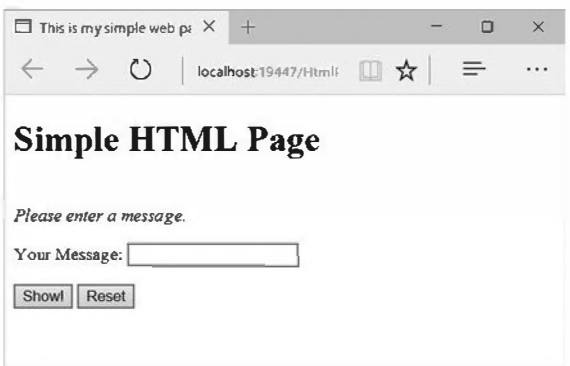


Рис. 31.8. Простая страница HTML

Обратите внимание, что каждому элементу управления в атрибуте `id` назначается подходящий идентификатор (`txtUserMessage`, `btnShow` и `btnReset`). Кроме того, каждый элемент ввода имеет дополнительный атрибут `type`, который указывает, что элемент автоматически сбрасывает все поля в их начальные значения (`type="reset"`), принимает текстовый ввод (`type="text"`) или функционирует как простая кнопка клиентской стороны, ничего не отправляющая веб-серверу (`type="button"`).

Сохраните файл, щелкните правой кнопкой мыши на поверхности визуального конструктора и выберите в контекстном меню пункт View in Browser (Просмотреть в браузере). На рис. 31.8 показана текущая страница в новом браузере Microsoft Edge.

На заметку! В случае выбора пункта View in Browser для файла HTML среда Visual Studio автоматически запустит веб-сервер IIS Express для размещения содержимого.

Роль сценариев клиентской стороны

В дополнение к элементам графического пользовательского интерфейса файл `*.html` может содержать блоки кода сценариев, которые будут обрабатываться запрашивающим браузером. Существуют две главные причины использования сценариев на стороне клиента:

- проверка достоверности пользовательского ввода перед обратной отправкой веб-серверу;
- взаимодействие с объектной моделью документа (Document Object Model — DOM) браузера.

Относительно первой причины следует понимать, что врожденным изъяном веб-приложения является потребность выполнять частые полные обмены (называемые *обратным отправками*) с машиной сервера для обновления разметки HTML, визуализируемой в браузере. Хотя обратные отправки неизбежны, вы всегда должны стремиться к минимизации передачи по сети. Один из приемов, которые экономят количество обратных отправок, связан с применением сценариев клиентской стороны для проверки достоверности пользовательского ввода перед отправкой данных формы веб-серверу. В случае выявления ошибки, такой как отсутствие данных в обязательном поле, пользователя об этом можно уведомить без накладных расходов на обратную отправку веб-серверу. (В конце концов, нет ничего более раздражающего пользователей, чем отправка данных по медленному подключению лишь для того, чтобы получить инструкции по исправлению ошибок ввода!)

На заметку! Имейте в виду, что даже при выполнении проверки достоверности на стороне клиента (в целях сокращения времени реакции) проверка достоверности *также* должна происходить на самом веб-сервере. Это позволит гарантировать, что данные не были искажены после того, как клиент отправил их по сети. Элементы управления проверкой достоверности ASP.NET автоматически выполняют проверку достоверности на клиентской и серверной стороне (более подробно об этом пойдет речь в главе 32).

Сценарии клиентской стороны могут также использоваться для взаимодействия с лежащей в основе объектной моделью (DOM) самого браузера. Большинство коммерческих браузеров открывают доступ к набору объектов, которые можно задействовать для управления поведением браузера.

Когда браузер производит синтаксический анализ страницы HTML, он строит в памяти дерево объектов, представляющее все содержимое веб-страницы (формы, элементы управления вводом и т.д.). Браузеры предоставляют API-интерфейс под названием DOM, который открывает доступ к дереву объектов и позволяет программно модифицировать его содержимое. Например, можно написать код JavaScript, который выполняется в браузере для получения значений из определенных элементов управления, изменения цвета элемента, динамического добавления новых элементов управления к странице и т.п.

К большому сожалению разные браузеры обычно предлагают сходные, *но не идентичные* объектные модели. Таким образом, блок сценария клиентской стороны, который взаимодействует с DOM, в разных браузерах может работать неодинаково (и потому обязательно требует тестирования).

В ASP.NET имеется свойство `HttpRequest.Browser`, которое во время выполнения позволяет определять функциональные возможности браузера и устройства, откуда поступил текущий запрос. Данная информация позволяет максимально оптимизировать формируемый ответ HTTP. Но потребность в ней возникает редко, если только вы не реализуете специальные элементы управления, поскольку все стандартные веб-элементы управления в ASP.NET умеют соответствующим образом визуализировать себя в зависимости от типа браузера. Эта ценная способность известна как *адаптивная визуализация* и предоставляется в готовом виде для всех стандартных элементов управления ASP.NET.

Для написания сценариев клиентской стороны доступны разнообразные языки программирования, среди которых наиболее популярным является JavaScript. Важно помнить, что JavaScript ни в каких аспектах нельзя считать языком Java. Наряду с тем, что JavaScript и Java имеют кое в чем похожий синтаксис, язык JavaScript менее мощный по сравнению с Java. Но самое главное: все современные веб-браузеры поддерживают язык JavaScript, делая его естественным кандидатом для написания логики сценариев клиентской стороны.

Пример сценария клиентской стороны

Чтобы проиллюстрировать роль сценариев клиентской стороны, давайте сначала выясним, как перехватывать события, отправляемые виджетами графического пользовательского интерфейса клиентской стороны. Для перехвата события щелчка на кнопке Show! (Показать) добавьте в определение элемента управления `btnShow` атрибут `onclick` и укажите в нем метод JavaScript по имени `btnShow_onclick()`:

```
<input id="btnShow" type="button" value="Show!"
       onclick="return btnShow_onclick()" />
```

Теперь поместите сразу после открывающего элемента `<head>` код функции JavaScript, которая вызывается, когда пользователь щелкает на кнопке. С помощью метода `alert()` отобразите на стороне клиента окно сообщения со значением из текстового поля, которое доступно через свойство `value`:

```
<script type="text/javascript">
// <![CDATA[
  function btnShow_onclick() {
    alert(window.txtUserMessage.value);
  }
// ]]>
</script>
```

Обратите внимание, что блок сценария помещен внутрь раздела `CDATA`. Причина проста: если страница попадет в браузер, который не поддерживает JavaScript, то код сценария будет воспринят как блок комментария и проигнорирован. Разумеется, страница может стать менее функциональной, но зато не вызовет ошибку во время визуализации в браузере. Теперь при просмотре страницы в браузере должна быть возможность вводить сообщение и видеть его в окне сообщения клиентской стороны (рис. 31.9).

Щелчок на кнопке Reset (Сброс) приводит к очистке текстового поля, потому что данная кнопка была определена с атрибутом `type="reset"`.

Обратная отправка веб-серверу

Эта простая страница HTML выполняет всю свою функциональность внутри браузера. Тем не менее, *реальная* веб-страница нуждается в обратной отправке по адресу ресурса на веб-сервере, одновременно передавая все введенные данные. После того, как ресурс серверной стороны получит эти данные, он может применять их для динамического построения подходящего HTTP-ответа.

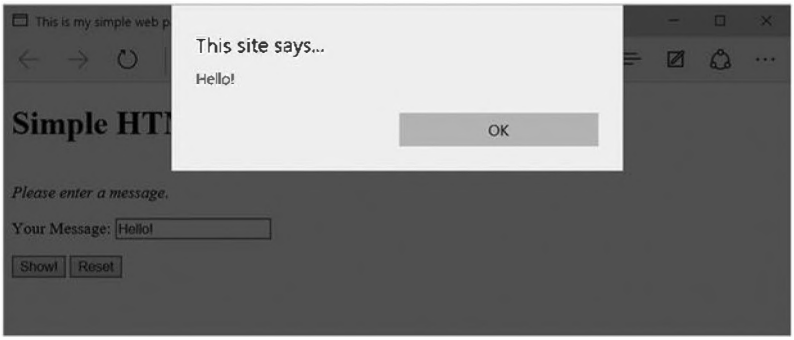


Рис. 31.9. Вызов функции JavaScript клиентской стороны

Атрибут `action` в открывающем дескрипторе `<form>` указывает получателя входных данных формы. В число возможных получателей входят почтовые серверы, другие файлы HTML на веб-сервере, веб-службы REST, страницы Web Forms и т.д.

Помимо атрибута `action`, скорее всего, будет также определена кнопка отправки (`submit`), щелчок на которой приведет к передаче данных формы веб-приложению через HTTP-запрос. В текущем примере в этом нет необходимости; однако ниже приведена разметка, где в открывающем дескрипторе `<form>` указан следующий атрибут:

```
<form id="defaultPage"
      action="http://localhost/Cars/MyAspNetPage.aspx" method="GET">
  <input id="btnPostBack" type="submit" value="Post to Server!"/>
  ...
</form>
```

Когда пользователь щелкает на кнопке отправки, данные формы отправляются файлу `MyAspNetPage.aspx`, расположенному по указанному URL. Если в качестве режима передачи задано `method="GET"`, то данные формы присоединяются к строке запроса в виде набора пар "имя-значение", разделенных амперсандами. Вы наверняка уже видели данные подобного рода в браузере, которые выглядят так:

```
http://www.google.com/search?hl=en&source=hp&q=vikings&cts=1264370773666
&aq=f&aql=&aqi=g1g-z1g1g-z1g1g-z1g4&oq=
```

Другой метод передачи данных формы веб-серверу указывается как `method="POST"`:

```
<form id="defaultPage"
      action="http://localhost/Cars/MyAspNetPage.aspx" method="POST">
  ...
</form>
```

В таком случае данные формы не присоединяются к строке запроса. При использовании метода `POST` данные формы не будут напрямую видны извне. Что более важно, данные `POST` не имеют ограничения по количеству символов, в то время как многие браузеры ограничивают длину запросов `GET`.

Обратные отправки в Web Forms

При создании веб-сайтов на основе Web Forms инфраструктура самостоятельно позаботится о механизме отправки данных. Одним из многих преимуществ построения веб-сайтов с применением ASP.NET Web Forms является то, что модель программирования находится поверх стандартного цикла "запрос/ответ" HTTP, который относится к системе, управляемой событиями. Таким образом, вместо ручной установки атрибута `action` и определения кнопки отправки HTML можно просто обрабатывать события элементов управления Web Forms, используя стандартный синтаксис C#.

Применяя такую управляемую событиями модель, можно очень легко выполнять обратную отправку веб-серверу с помощью множества элементов управления. При необходимости это можно делать, когда пользователь щелкает на переключателе, на элементе в окне со списком, на каком-нибудь дне в элементе управления типа календаря и т.д. В каждом случае вы просто обрабатываете соответствующее событие, а исполняющая среда ASP.NET автоматически выдаст корректные данные HTML для отправки.

Исходный код. Веб-сайт `SimpleWebPage` доступен в подкаталоге `Chapter_31`.

Обзор API-интерфейса Web Forms

К настоящему моменту краткий обзор разработки классических веб-приложений завершен, так что вы готовы углубиться в исследования Web Forms. Как и можно было ожидать, в каждой новой версии платформы .NET к API-интерфейсам программирования веб-приложений добавлялась дополнительная функциональность, и это справедливо также в отношении .NET 4.6. Независимо от версии .NET, с которой вы имеете дело, перечисленные ниже возможности доступны всем веб-приложениям, основанным на ASP.NET Web Forms.

- Инфраструктура ASP.NET предоставляет модель *отделенного кода*, которая позволяет отделять логику презентации (разметка HTML) от бизнес-логики (код C#).
- Страницы ASP.NET кодируются с использованием языков программирования .NET, а не языков сценариев серверной стороны. Файлы кода компилируются в допустимые .NET-сборки `*.dll` (которые транслируются в намного более быстродействующий исполняемый код).
- Элементы управления Web Forms могут применяться для построения пользовательского веб-интерфейса в манере, похожей на построение настольных оконных приложений.
- Приложения Web Forms могут использовать любые сборки из библиотек базовых классов .NET и конструируются с применением объектно-ориентированных технологий, рассматриваемых в данной книге (классы, интерфейсы, структуры, перечисления и делегаты).
- Приложения Web Forms можно легко конфигурировать с помощью конфигурационного файла веб-приложения (`Web.config`).

Прежде всего, здесь необходимо отметить, что пользовательский интерфейс веб-страницы Web Forms может быть сконструирован с использованием разнообразных *веб-элементов управления*. В отличие от типичного элемента управления HTML веб-элементы управления функционируют на веб-сервере и выдают в HTTP-ответе корректные дескрипторы HTML. Одно лишь это является огромным преимуществом Web Forms, потому что значительно сокращает объем разметки HTML, которая должна быть написана вручную. В качестве небольшого примера предположим, что внутри веб-страницы Web Forms определен следующий элемент управления Web Forms:

```
<asp:Button ID="btnMyButton" runat="server" Text="Button" BorderColor="Blue"
            BorderStyle="Solid" BorderWidth="5px" />
```

Вы узнаете детали объявления элементов управления Web Forms чуть позже, а пока обратите внимание, что многие атрибуты элемента управления `<asp:Button>` выглядят очень похожими на свойства, которые встречались в примерах WPF. То же самое верно для всех элементов управления Web Forms, поскольку когда в Microsoft создавали инструментальный набор веб-элементов управления, эти виджеты намеренно были

спроектированы, чтобы иметь внешний вид и поведение, подобные своим настольным аналогам.

Если теперь браузер обратится к файлу `.aspx`, содержащему указанный элемент управления, то этот элемент выдаст в выходной поток следующее объявление HTML:

```
<input type="submit" name="btnMyButton" value="Button" id="btnMyButton"
       style="border-color:Blue;border-width:5px;border-style:Solid;" />
```

Обратите внимание, что веб-элемент управления выдает стандартную разметку HTML, которая может быть визуализирована в любом браузере. С учетом этого важно понимать, что применение элементов управления Web Forms никак не привязывает к семейству операционных систем Microsoft или к браузеру Microsoft Internet Explorer/Edge. Страницу Web Forms можно просматривать в среде любой операционной системы и в любом браузере (включая портативные устройства, такие как Apple iPhone, Android и Windows Phone).

В предшествующем списке возможностей было указано, что приложение Web Forms будет компилироваться в сборку .NET. Таким образом, веб-проекты ничем не отличаются от любой .NET-сборки `.dll`, которая строилась в примерах, рассмотренных в книге. Скомпилированное веб-приложение будет состоять из кода CIL, манифеста сборки и метаданных типов. Это дает несколько крупных преимуществ, наиболее заметными из которых являются выигрыш в производительности, строгая типизация и возможность микроуправления со стороны CLR (например, сборка мусора и т.д.).

Наконец, приложения Web Forms предоставляют программную модель, посредством которой разметку страницы можно отделять от связанной кодовой базы C# с применением *файлов кода*. В случае использования файлов кода введенная разметка будет отображаться на полноценную объектную модель, которая объединяется с файлом кода C# через объявления частичных классов.

Основные возможности Web Forms 2.0 и последующих версий

Версия ASP.NET 1.0 была значительным шагом в правильном направлении, а ASP.NET 2.0 предоставила множество дополнительных средств, которые помогли перейти от построения динамических *веб-страниц* к построению многофункциональных *веб-сайтов*. Ниже приведен частичный список основных средств.

- Появление веб-сервера разработки ASP.NET (означающего, что разработчики больше не нуждаются в наличии полной версии IIS, установленной на машинах разработки). Теперь он заменен IIS Express.
- Большое количество новых веб-элементов управления, которые обеспечили поддержку во многих сложных ситуациях (элементы управления навигацией, элементы управления безопасностью, новые элементы управления привязкой данных и т.д.).
- Появление *мастер-страниц*, которые позволяют разработчикам присоединять общие области пользовательского интерфейса к набору связанных страниц.
- Поддержка *тем*, которые предлагают декларативный способ изменения внешнего вида и поведения всего веб-приложения на веб-сервере.
- Поддержка *веб-частей* (Web Parts), которые позволяют конечным пользователям настраивать внешний вид и поведение веб-страницы и сохранять эти настройки для последующего применения (подобно порталам).
- Появление веб-ориентированной утилиты конфигурации и управления, которая обслуживает разнообразные файлы `Web.config`.

Помимо веб-сервера разработки ASP.NET одним из самых крупных нововведений ASP.NET 2.0 было появление *мастер-страниц*. Как известно, большинство веб-сайтов поддерживают согласованный внешний вид и поведение для всех страниц сайта. Взгляните на коммерческий веб-сайт, подобный `www.amazon.com`. Каждая страница содержит одни и те же элементы, такие как общий заголовок, общий нижний колонтитул, общее меню навигации и т.п.

С использованием мастер-страниц можно моделировать общую функциональность и определять *места заполнения* для подключения других файлов `.aspx`. Это существенно облегчает изменение общего вида сайта (перемещение панели навигации в другое место, смену логотипа в заголовке и т.д.) за счет простой модификации мастер-страницы, оставляя другие файлы `.aspx` незатронутыми.

На заметку! Мастер-страницы настолько удобны, что в Visual Studio 2010 и последующих версиях все новые веб-проекты Web Forms по умолчанию включают мастер-страницу.

В ASP.NET 2.0 также было добавлено много новых веб-элементов управления, в том числе элементы управления, которые автоматически поддерживают общие средства безопасности (вход на сайт, восстановление пароля и т.д.), элементы управления, позволяющие укладывать навигационную структуру поверх набора связанных файлов `.aspx`, и другие элементы управления, выполняющие сложные операции привязки данных, где необходимые SQL-запросы могут генерироваться с применением набора элементов управления Web Forms.

Основные возможности Web Forms 3.5 (и .NET 3.5 SP1) и последующих версий

Следует отметить, что в версии .NET 3.5 приложения Web Forms получили возможность задействовать модель программирования LINQ (также появившуюся в .NET 3.5) и перечисленные далее веб-ориентированные средства.

- Поддержка привязки данных для классов ADO.NET Entity Framework (см. главу 23).
- Поддержка ASP.NET Dynamic Data. Это инфраструктура, подобная Ruby on Rails, которая может использоваться для построения веб-приложений, управляемых данными. Она открывает доступ к таблицам базы данных за счет их кодирования в URI веб-службы ASP.NET, а для данных в таблицах автоматически генерируется разметка HTML.
- Интегрированная поддержка разработки в стиле Ajax, которая по существу позволяет выполнять обратные микро-отправки для как можно более быстрого обновления части веб-страницы.

Шаблоны проектов ASP.NET Dynamic Data, появившиеся в .NET 3.5 Service Pack 1, предлагают новую модель для построения сайтов, которые в высокой степени управляются реляционной базой данных. Конечно, большинство веб-сайтов в какой-то мере будут нуждаться во взаимодействии с базами данных, но проекты ASP.NET Dynamic Data тесно связаны с ADO.NET Entity Framework и непосредственно ориентированы на быструю разработку сайтов, управляемых данными (аналогичных сайтам, которые строятся с помощью Ruby).

Основные возможности Web Forms 4.0

В версии .NET 4.0 к платформе веб-разработки Microsoft были добавлены дополнительные возможности. Вот список некоторых наиболее заметных веб-ориентированных средств.

- Возможность сжатия данных "состояния представления" с применением стандарта GZIP.
- Библиотека JQuery включена для инфраструктур Web Forms и MVC.
- Обновленные определения браузеров для обеспечения корректной визуализации страниц ASP.NET в новых браузерах и устройствах (Google Chrome, Apple iPhone, Windows Phone, Android и т.д.).
- Возможность настройки вывода из элементов управления проверкой достоверности с помощью каскадных таблиц стилей (CSS).
- Включение в библиотеку ASP.NET элемента управления `Chart`, который позволяет создавать страницы ASP.NET с наглядными диаграммами для сложного статистического или финансового анализа.
- Поддержка шаблонов проектов ASP.NET Model View Controller, которая уменьшает зависимость между уровнями приложения за счет использования шаблона MVC (Model-View-Controller — модель-представление-контроллер). Это совершенно другой подход к разработке веб-сайтов, который немного напоминает модель программирования с веб-формами, рассматриваемую в настоящей книге.

Несмотря на то что список, несомненно, впечатляет (и он включает лишь подмножество новых средств), работа, которую проделали специалисты из Microsoft в отношении Web Forms для ASP.NET 4.5 заставила многих разработчиков возвратиться к Web Forms из ASP.NET MVC (рассматривается в главе 34).

Основные возможности Web Forms 4.5 и Web Forms 4.6

Двумя главными областями, которым уделялось внимание в версии .NET 4.5, были улучшения производительности и перенос многих возможностей ASP.NET MVC в инфраструктуру Web Forms. Ниже представлен частичный список нововведений в Web Forms 4.5 и Web Forms 4.6.

Возможности, добавленные в Web Forms 4.5

- Многочисленные обновления для поддержки HTML 5.0.
- Интеграция с новыми асинхронными языковыми возможностями C# и VB.
- Можно объявлять, к какому типу данных планируется привязывать элемент управления, с применением нового свойства `ItemType`, что открывает возможность для строго типизированных элементов управления, поддержки IntelliSense и многого другого.
- Привязка модели, которая означает возможность отображения данных из страницы прямо на параметры типа метода.
- Проверка достоверности клиентской стороны теперь интегрирована с JQuery, позволяя писать более ясный код проверки.
- Стали доступны дополнительные возможности проверки достоверности посредством аннотаций данных (Data Annotations), которые являются атрибутами в классах моделей.
- Дополнительная защита от атак межсайтовыми сценариями с включением (по умолчанию) библиотеки AntiXSS.
- Сокращение размеров файлов (JavaScript и CSS) с помощью минификации (уменьшающей размеры файлов за счет сжатия текста внутри файлов).

- Сокращенное количество обращений браузера за счет объединения файлов в один посредством упаковки.
- Возможность откладывания проверки достоверности запросов, что позволяет отправлять *потенциально* небезопасное содержимое (используйте с осторожностью).
- Возможность применения более одного серверного кода при компиляции приложений Web Forms.

Возможности, добавленные в Web Forms 4.6

- Поддержка нового высокоскоростного протокола HTTP2 (в настоящий момент он доступен только для защищенных приложений на IIS).
- Включение новых средств C# 6 с использованием компилятора Roslyn.
- Возможность применения `async/await` в функциях привязки модели.

Вы наверняка согласитесь с тем, что набор средств Web Forms довольно широк (к тому же этот API-интерфейс содержит намного больше возможностей, чем было здесь перечислено). По правде говоря, если попытаться раскрыть все средства Web Forms, то объем книги легко увеличился бы вдвое (а то и втрое). Поскольку это нереально, мы рассмотрим только базовые средства ASP.NET, которые, скорее всего, будут использоваться вами ежедневно. Описание средств, которые в книге не раскрыты, находится в документации .NET Framework 4.6 SDK.

На заметку! Если нужно исчерпывающее руководство по разработке веб-приложений в ASP.NET, то рекомендуется почитать книгу *ASP.NET 4.5 с примерами на C# 5.0 для профессионалов, 5-е изд.* (ИД "Вильямс", 2014 г.).

Построение однофайловой веб-страницы Web Forms

Страница Web Forms может быть сконструирована с применением двух основных подходов, первый из которых предусматривает построение единственного файла `.aspx`, содержащего смесь кода серверной стороны и разметки HTML. В случае подхода с *моделью однофайловой страницы* код серверной стороны помещается внутрь области `<script>`, но сам код *не* является сценарием (например, на VBScript или JavaScript). Взамен код в блоке `<script>` пишется на выбранном языке .NET (C#, Visual Basic и т.д.).

При построении веб-страницы, которая содержит очень мало кода (но значительный объем статической разметки HTML), модель однофайловой страницы может оказаться проще в работе, потому что она позволяет видеть код и разметку в едином файле `.aspx`. Кроме того, помещение процедурного кода и разметки HTML в единственный файл `.aspx` обеспечивает ряд других преимуществ.

- Страницы, написанные с использованием однофайловой модели, несколько легче развертывать или отправлять другим разработчикам.
- Поскольку нет никаких зависимостей между многочисленными файлами, однофайловую страницу проще переименовывать.
- Немного облегчается манипулирование файлами в системе управления исходным кодом, т.к. все действия производятся с единственным файлом.

Недостаток модели однофайловой страницы связан с тем, что она приводит к появлению излишне сложных файлов, поскольку разметка пользовательского интерфейса и

программная логика находятся в одном месте. Тем не менее, путешествие по Web Forms мы начнем с исследования модели однофайловой страницы.

Наша цель заключается в построении файла `.aspx`, который отображает содержимое таблицы `Inventory` базы данных `AutoLot` (созданной в главе 21) с применением Entity Framework. Запустите Visual Studio, выберите пункт меню File⇨New Project (Файл⇨Создать проект), после чего откроется диалоговое окно New Project (Новый проект). В узле Visual C# древовидного представления слева выберите элемент Web, в центральной панели укажите в качестве типа проекта ASP.NET Web Application (Веб-приложение ASP.NET) и введите `SinglePageModel` в поле Name (Имя), как показано на рис. 31.10.

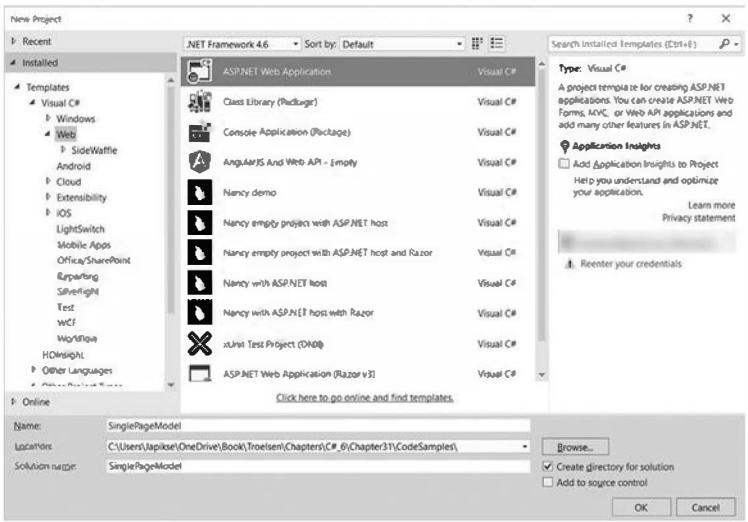


Рис. 31.10. Выбор типа проекта ASP.NET Web Application в диалоговом окне New Project

После щелчка на кнопке OK вы увидите обновленное диалоговое окно New ASP.NET Project (Новый проект ASP.NET). Убедитесь, что в области ASP.NET 4.6 Templates (Шаблоны ASP.NET 4.6) выбран шаблон Empty (Пустой). Оставьте неотмеченными флажки Web Forms, MVC и Web API в области Add folders and core references for (Добавить папки и основные ссылки для), также оставьте неотмеченным флажок Add unit tests (Добавить модульные тесты) и снимите отметку с флажка Host in the cloud (Разместить в облаке), если он отмечен, в области Microsoft Azure (рис. 31.11).

На заметку! Текущей полной версией платформы является .NET 4.6 Framework (раскрываемая в этой книге). Версия ASP.NET 5 встроена в платформу .NET Core, которая представляет собой подмножество .NET 4.6. Обратите внимание, что с точки зрения разработки веб-приложений .NET Core ориентируется на межплатформенные веб-сайты и поддерживает только инфраструктуры ASP.NET MVC и Web API.

Далее добавим в проект новую веб-форму, выбрав пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент). В древовидном представлении слева должен быть выбран узел Web, а в нем Web Forms. Назначьте файлу имя `Default.aspx`.

Добавление ссылки на сборку `AutoLotDAL.dll`

С помощью проводника Windows скопируйте папку `AutoLotDAL` из подкаталога `Chapter_23` (либо из папки `SinglePageModel` в подкаталоге `Chapter_31`). Добавьте

проект к решению, щелкнув правой кнопкой мыши на имени решения, выбрав в контекстном меню пункт Add⇨Existing Project (Добавить⇨Существующий проект) и указав проект AutoLotDAL. Затем добавьте ссылку на проект AutoLotDAL, для чего щелкните правой кнопкой мыши на узле References (Ссылки) в проекте SinglePageModel и выберите AutoLotDAL.

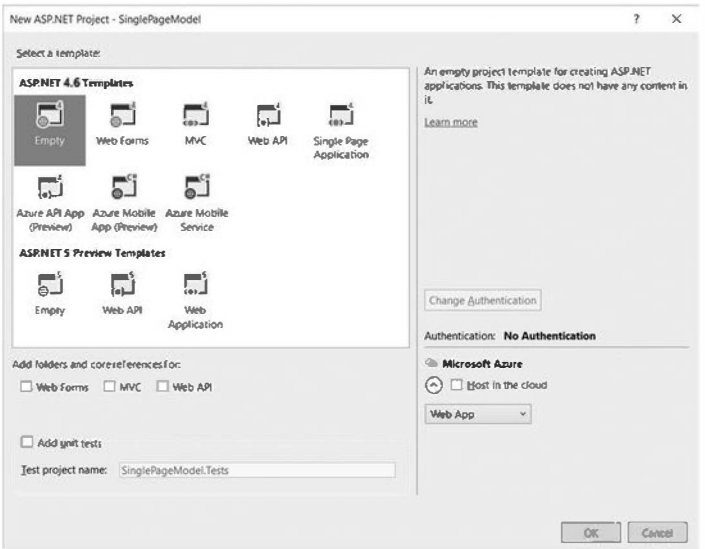


Рис. 31.11. Выбор шаблона проекта Empty

В проект AutoLotDAL понадобится внести небольшое изменение, связанное с пространством имен System.Web. При вызове конструктора класса DatabaseLogger не указан каталог, что приведет к отказу в работе кода на веб-сайте из-за нехватки разрешений. Ситуацию необходимо исправить, указав физический каталог веб-сайта. В пространстве имен System.Web имеется серверная переменная по имени HttPRuntime.AppDomainAppPath, которая содержит физический каталог веб-сайта. Откройте файл AutoLotEntities.cs из папки EF, добавьте оператор using для System.Web и измените инициализатор DatabaseLogger следующим образом:

```
static readonly DatabaseLogger DatabaseLogger =
  new DatabaseLogger($"{HttpRuntime.AppDomainAppPath}/sqllog.txt");
```

Такое изменение гарантирует, что журнальный файл будет создаваться в том же самом каталоге, что и веб-сайт, устраняя проблему с разрешениями.

Добавьте к веб-проекту инфраструктуру Entity Framework, щелкнув правой кнопкой мыши на имени решения в окне Solution Explorer, выбрав в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet) и указав Entity Framework. Понадобится также модифицировать файл Web.config (который аналогичен файлам App.config, с которыми приходилось работать в предшествующих главах), как показано ниже (в строке подключения должно быть указано соответствующее имя экземпляра SQL Server):

```
<configuration>
  <configSections>
    <!-- Дополнительные сведения о конфигурации Entity Framework доступны
         по адресу http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
```

```
    type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c56
1934e089"
    requirePermission="false"/>
  </configSections>
  <system.web>
    <compilation debug="true" targetFramework="4.6"/>
    <httpRuntime targetFramework="4.6"/>
  </system.web>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.
LocalDbConnectionFactory, EntityFramework">
      <parameters>
        <parameter value="mssqllocaldb"/>
      </parameters>
    </defaultConnectionFactory>
    <providers>
      <provider invariantName="System.Data.SqlClient" type="System.Data.
Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer"/>
    </providers>
  </entityFramework>
  <connectionStrings>
    <add name="AutoLotConnection"
      connectionString="data source=.\SQLEXPRESS2014;initial
catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;
App=EntityFramework"
      providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

Проектирование пользовательского интерфейса

Откройте файл `Default.aspx`, щелкните на вкладке Design (Схема), выберите в панели инструментов Visual Studio вкладку Data (Данные) и перетащите на поверхность визуального конструктора страницы элемент управления `GridView`, поместив его между открывающим и закрывающим элементами `form`. Визуальный конструктор заполняет `GridView` произвольными данными, чтобы дать представление о том, как будет выглядеть страница. С помощью окна Properties установите разнообразные визуальные свойства. Отыщите в разметке страницы раздел `<form>`. Обратите внимание, что веб-элемент управления определен с использованием дескриптора `<asp:>`. После префикса `asp` указано имя элемента управления Web Forms (`GridView`). Перед закрывающим дескриптором элемента находится последовательность пар "имя-значение", которые соответствуют настройкам, доступным в окне Properties:

```
<form id="form1" runat="server">
<div>
  <asp:GridView ID="carsGridView" runat="server">
  </asp:GridView>
</div>
</form>
```

Элементы управления Web Forms (а также атрибут `runat="server"`) будут подробно рассматриваться в главе 32. Пока просто запомните, что веб-элементы управления — это объекты, обрабатываемые на веб-сервере, который автоматически выпускает их представление HTML в исходящем HTTP-ответе. Помимо такого важного преимущества элементы управления Web Forms имитируют модель программирования настольных

приложений в том, что имена свойств, методов и событий обычно воспроизводят их аналоги из Windows Forms/WPF.

Добавление логики доступа к данным

Теперь добавьте к дескриптору `asp:GridView` атрибут `ItemType` со значением `"AutoLotDAL.Models.Inventory"`. Эта новая возможность, появившаяся в .NET 4.5, обеспечивает поддержку строго типизированных списковых элементов управления в ASP.NET Web Forms и позволяет средству IntelliSense распознавать классы, доступные в решении.

Добавьте атрибут `SelectMethod` со значением `"GetData"`. Этот атрибут также был введен в версии .NET 4.5 и устанавливает метод, который будет выполняться, когда элемент управления визуализируется для получения данных, заполняющих списковый элемент управления. Ниже показана обновленная разметка:

```
<asp:GridView ID="carsGridView" runat="server"
  ItemType="AutoLotDAL.Models.Inventory"
  SelectMethod="GetData" >
</asp:GridView>
```

Создайте метод `GetData()` внутри дескриптора `<script>` страницы. В этом методе вызовите метод `InventoryRepo.GetAll()`. С помощью директив `<%@ Import ... %>` импортируйте пространства имен `AutoLotDAL.Models` и `AutoLotDAL.Repos`. Код должен выглядеть следующим образом:

```
<!-- Находится в начале файла, после директивы -->
<%@ Import Namespace="AutoLotDAL.Models" %>
<%@ Import Namespace="AutoLotDAL.Repos" %>

<!-- Находится где-то в файле, перед определением элемента управления
GridView -->
<script runat="server">
  public IEnumerable<Inventory> GetData()
  {
    return new InventoryRepo().GetAll();
  }
</script>
```

На заметку! При построении страницы с применением однофайловой модели необходимо использовать только директиву `<%@ Import %>`. В случае стандартного подхода с файлом кода пространства имен включаются в этот файл с помощью ключевого слова `using` языка C#. То же самое касается директивы `<%@ Assembly %>`.

Прежде чем приступить к исследованию формата файла `*.aspx` давайте произведем тестовый запуск. Сохраните файл `.aspx`. Щелкните на значке Run (Запустить) или нажмите <F5>, что приведет к запуску веб-сервера IIS Express, на котором разместится страница.

Во время обслуживания страницы выполняется метод, указанный в атрибуте `SelectMethod`, который загружает данные в элемент управления `GridView`. Результат приведен на рис. 31.12.

Пока что пользовательский интерфейс довольно скромен. Чтобы улучшить его, выберите элемент `GridView` на поверхности визуального конструктора и в контекстном меню (открывающемся по щелчку на небольшой стрелке в правом верхнем углу элемента) выберите пункт Auto Format (Автоформат), как показано на рис. 31.13.

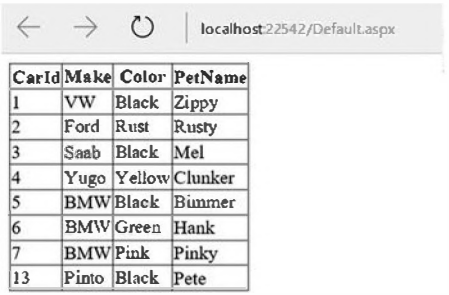


Рис. 31.12. ASP.NET предоставляет декларативную модель привязки данных

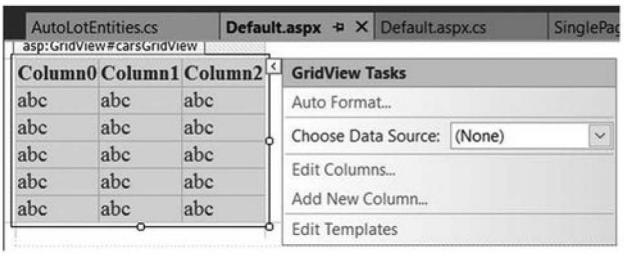


Рис. 31.13. Конфигурирование элемента управления `GridView`

В открывшемся диалоговом окне выберите подходящий шаблон (скажем, Slate). После щелчка на кнопке OK просмотрите сгенерированное объявление элемента управления, которое стало более развитым, чем ранее:

```
<asp:GridView ID="carsGridView" runat="server"
  ItemType="AutoLotDAL.Models.Inventory"
  SelectMethod="GetData" BackColor="White" BorderColor="#E7E7FF"
  BorderStyle="None" BorderWidth="1px" CellPadding="3" GridLines="Horizontal" >
 <AlternatingRowStyle BackColor="#F7F7F7" />
 <FooterStyle BackColor="#B5C7DE" ForeColor="#4A3C8C" />
 <HeaderStyle BackColor="#4A3C8C" Font-Bold="True" ForeColor="#F7F7F7" />
 <PagerStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" HorizontalAlign="Right" />
 <RowStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" />
 <SelectedRowStyle BackColor="#738A9C" Font-Bold="True" ForeColor="#F7F7F7" />
 <SortedAscendingCellStyle BackColor="#F4F4FD" />
 <SortedAscendingHeaderStyle BackColor="#5A4C9D" />
 <SortedDescendingCellStyle
   BackColor="#D8D8F0" />
 <SortedDescendingHeaderStyle
   BackColor="#3E3277" />
</asp:GridView>
```

Снова запустив приложение и щелкнув на кнопке, вы увидите более интересный пользовательский интерфейс (рис. 31.14).

Просто, не правда ли? Разумеется, как всегда, сложности кроются в деталях, поэтому давайте немного углубимся в состав этого файла `.aspx`, начав с выяснения роли директивы `<%@ Page ... %>`. Имейте в виду, что рассмотренные здесь темы

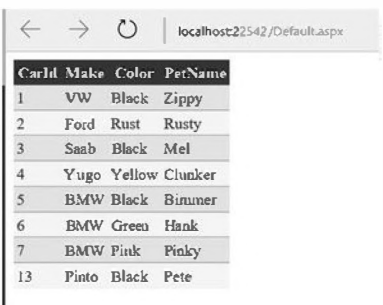


Рис. 31.14. Тестовая страница с более насыщенным оформлением

будут применимы к рекомендуемой модели с *файлом кода*, которая описана далее в главе.

Роль директив ASP.NET

Файл `.aspx` обычно начинается с набора *директив*. Директивы ASP.NET всегда помечаются маркерами `<%@ ... %>` и могут содержать разнообразные атрибуты, которые информируют исполняющую среду ASP.NET о том, как следует обрабатывать конкретную директиву.

Каждый файл `.aspx` должен содержать минимум директиву `<%@ Page %>`, которая используется для определения управляемого языка, применяемого внутри страницы (посредством атрибута `language`). Кроме того, директива `<%@ Page %>` может определять имя связанного файла отделенного кода (рассматривается ниже) и т.д. Наиболее интересные атрибуты директивы `<%@ Page %>` кратко описаны в табл. 31.1.

Таблица 31.1. Избранные атрибуты директивы `<%@ Page %>`

Атрибут	Описание
`CodePage`	Указывает имя связанного файла отделенного кода
`EnableTheming`	Указывает, поддерживают ли элементы управления на странице `.aspx` темы ASP.NET
`EnableViewState`	Указывает, поддерживается ли состояние представления между запросами страницы (более подробно это свойство рассматривается в главе 33)
`Inherits`	Определяет класс в файле отделенного кода, от которого наследуется страница; может быть любым классом, производным от `System.Web.UI.Page`
`MasterPageFile`	Устанавливает мастер-страницу, используемую в сочетании с текущей страницей `.aspx`
`Trace`	Указывает, включена ли трассировка

В дополнение к директиве `<%@ Page %>` файл `*.aspx` может содержать различные директивы `<%@ Import %>` для явного указания пространств имен, требуемых текущей страницей, и директивы `<%@ Assembly %>` для описания внешних библиотек кода, которые используются сайтом (и обычно расположены в папке `\bin` веб-сайта).

В этом примере было заявлено применение типов из пространств имен `Models` и `Repos` сборки `AutoLotDAL.dll`. Если необходимо использовать дополнительные пространства имен .NET, то нужно просто указать несколько директив `<%@ Import %>`/`<%@ Assembly %>`. Помимо директив `<%@ Page %>`, `<%@ Import %>` и `<%@ Assembly %>` в ASP.NET определено несколько других директив, которые могут появляться в файле `.aspx`; они будут обсуждаться позже. Примеры применения других директив вы найдете в оставшихся главах.

Анализ блока `<script>`

В однофайловой модели страницы файл `.aspx` может содержать логику сценариев серверной стороны, которая выполняется на веб-сервере. В таком случае *критически важно*, чтобы все блоки кода серверной стороны были определены для выполнения на сервере с использованием атрибута `runat="server"`. Если атрибут `runat="server"` не указан, то исполняющая среда считает, что был написан блок сценария *клиентской стороны*, который предназначен для встраивания в исходящий HTTP-ответ, и генерирует исключение. Ниже показан правильный блок `<script>` серверной стороны:

```
<script runat="server">
  public IEnumerable<Inventory> GetData()
  {
    return new InventoryRepo().GetAll();
  }
</script>
```

На заметку! Все элементы управления Web Forms должны иметь в открывающем дескрипторе атрибут `runat="server"`. В противном случае они не будут генерировать свою разметку HTML в исходящий HTTP-ответ.

Анализ объявлений элементов управления ASP.NET

Последний интересный момент в первом примере связан с объявлением веб-элемента управления `GridView`. Подобно классическому коду ASP и низкоуровневой разметке HTML виджеты Web Forms помещаются внутрь элементов `<form>`. Однако на этот раз открывающий дескриптор `<form>` помечен атрибутом `runat="server"`, а элементы управления снабжены дескрипторным префиксом `asp:`. Любой элемент с таким префиксом является членом библиотеки элементов управления ASP.NET и имеет соответствующее представление в виде класса C# внутри некоторого пространства имен .NET библиотеки базовых классов .NET. Вот как выглядит разметка:

```
<form id="form1" runat="server">
  <div>
    <asp:GridView ID="carsGridView" runat="server"
        ItemType="AutoLotDAL.Models.Inventory"
        SelectMethod="GetData" >
    </asp:GridView>
  </div>
</form>
```

Пространство имен `System.Web.UI.WebControls` сборки `System.Web.dll` содержит большинство элементов управления Web Forms. Открыв браузер объектов Visual Studio, можно обнаружить, к примеру, элемент управления `DataGrid` (рис. 31.15).

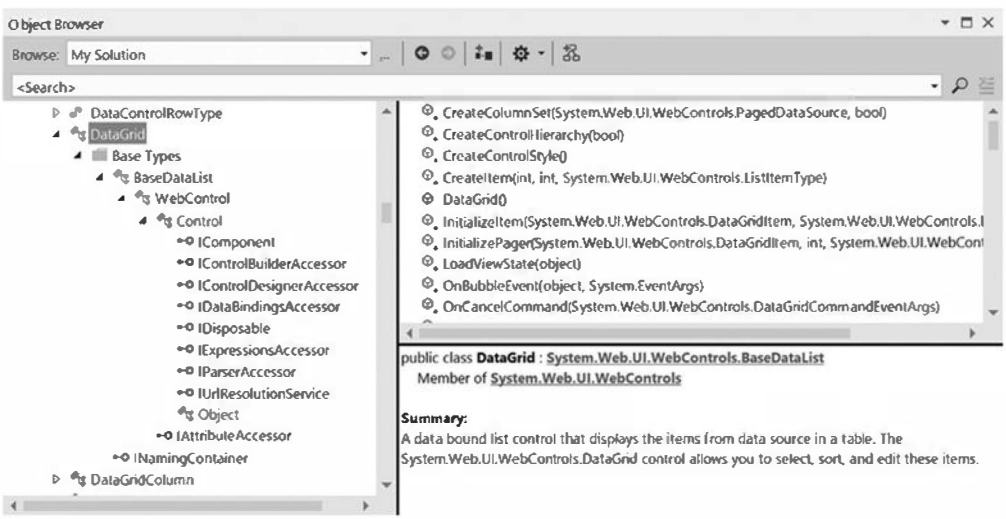


Рис. 31.15. Все объявления элементов управления ASP.NET отображаются на типы классов .NET

Как видите, на самой вершине цепочки наследования для любого элемента управления Web Forms находится класс `System.Object`. Родительский класс `WebControl` — это общий базовый класс для всех элементов управления ASP.NET, который определяет все общие свойства пользовательского интерфейса (`BackColor`, `Height` и т.д.). Класс `Control` также очень распространен в инфраструктуре, но в нем определены члены, больше ориентированные на инфраструктуру (привязка данных, состояние представления и т.п.), а не на внешний вид и поведение дочерних элементов. Дополнительные сведения об этих классах вы узнаете в главе 33.

Исходный код. Веб-сайт `SinglePageModel` доступен в подкаталоге `Chapter_31`.

Построение веб-страницы ASP.NET с использованием файлов кода

Хотя однофайловая модель временами бывает полезной, стандартный подход, принятый в Visual Studio (при создании нового веб-проекта), предусматривает применение подхода, который называется *отделенным кодом* и позволяет разносить программный код серверной стороны и логику презентации HTML в два разных файла. Эта модель работает довольно хорошо, когда страницы содержат существенный объем кода или когда созданием одного веб-сайта занимается множество разработчиков. Модель отделенного кода обеспечивает и другие преимущества.

- Поскольку страницы с отделенным кодом обеспечивают четкое разделение разметки HTML и кода, можно организовать параллельную работу дизайнеров над разметкой и программистов над кодом C#.
- Код не виден дизайнерам страниц и прочему персоналу, который имеет дело только с разметкой страницы (как и можно было догадаться, специалистам по разметке HTML не всегда интересны детали кода C#).
- Файлы кода можно использовать во множестве файлов `.aspx`.

Независимо от выбранного подхода *никакой* разницы в производительности не будет. На самом деле многие приложения Web Forms выигрывают, когда задействованы оба подхода. Для демонстрации модели с отделенным кодом давайте заново создадим предыдущий пример с применением шаблона пустого веб-сайта в Visual Studio. Выберите пункт меню File⇨New Project (Файл⇨Создать проект), затем ASP.NET Web Application (Веб-приложение ASP.NET) и, наконец, шаблон Empty (Пустой) в группе ASP.NET 4.6 Templates (Шаблоны ASP.NET 4.6).

Теперь с использованием пункта меню Project⇨Add New Item (Проект⇨Добавить новый элемент) вставьте новый элемент Web Form (Веб-форма) по имени `Default.aspx`. Создайте в визуальном конструкторе пользовательский интерфейс, состоящий из одного элемента управления `GridView`, и настройте его в окне Properties по собственному усмотрению. При желании можете просто скопировать в новый файл `.aspx` объявления элементов управления из примера `SinglePageModel`. Учитывая, что разметка здесь в точности та же самая, повторно она не приводится (понадобится только поместить объявления элементов управления между дескрипторами `<form>` и `</form>`).

Обратите внимание, что директива `<%@ Page %>`, применяемая в модели файла кода, имеет несколько атрибутов:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeBehind="Default.aspx.cs" Inherits="CodeBehindPageModel.Default" %>
```

Атрибут `CodeFile` используется для указания связанного внешнего файла, который содержит код логики для этой страницы. По умолчанию такие файлы отделенного кода именуются за счет добавления суффикса `.cs` к имени файла `.aspx` (в рассматриваемом примере получается `Default.aspx.cs`). Заглянув в окно Solution Explorer, вы увидите, что файл отделенного кода находится в подузле под значком веб-формы (рис. 31.16). Открыв файл отделенного кода, вы обнаружите в нем частичный класс, производный от `System.Web.UI.Page`, с поддержкой обработки события `Load`. Обратите внимание, что полностью заданное имя этого класса (`CodeBehindPageModel.Default`) идентично имени, указанному в атрибуте `Inherits` директивы `<%@ Page %>`:

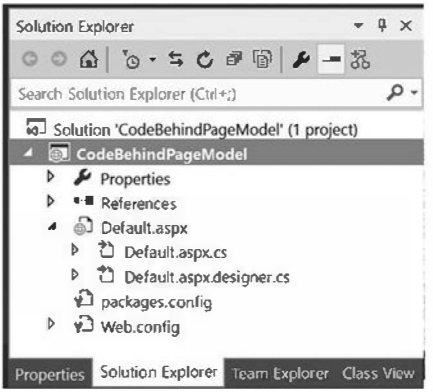


Рис. 31.16. Файл отделенного кода, связанный с заданным файлом `*.aspx`

```
public partial class Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }
}
```

Ссылка на проект AutoLotDAL

Вам потребуется модифицировать проект `AutoLotDAL` (или задействовать скомпилированную сборку `AutoLotDAL.dll`) из предыдущего примера. Если вы добавили проект `AutoLotDAL` к решению, то придется добавить в свой веб-проект ссылку на `AutoLotDAL`. Если вы собираетесь ссылаться на скомпилированную сборку `AutoLotDAL.dll`, то должны добавить этот файл в папку `\bin` в окне Solution Explorer, как показано на рис. 31.17 (может понадобиться щелчок на кнопке Show All Files (Показать все файлы)).

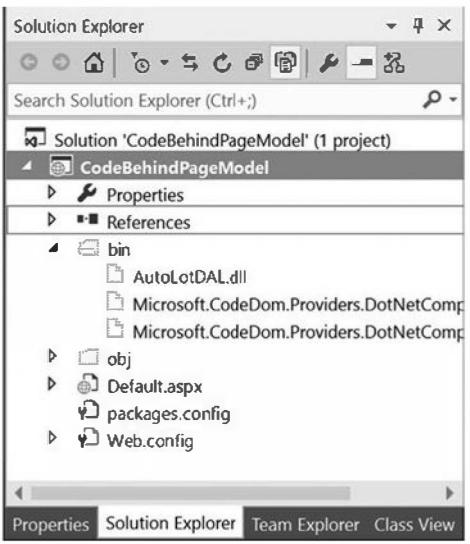


Рис. 31.17. В веб-проектах Visual Studio используются специальные папки ASP.NET

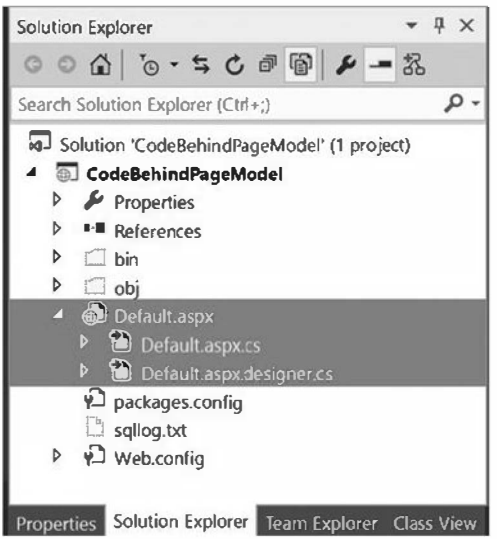


Рис. 31.18. Каждая веб-страница состоит из трех файлов

Как и в предыдущем примере, добавьте в веб-проект инфраструктуру Entity Framework, щелкнув правой кнопкой мыши на имени проекта, выбрав в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet) и установив EF. Наконец, скопируйте узел `<connectionStrings>` в файл `Web.config`.

Обновление файла кода

Заглянув внутрь файла `Default.aspx` из предыдущего примера, вы увидите, что каждая страница Web Forms состоит из трех файлов: файла `*.aspx` (для разметки), файла `*.designer.cs` (для кода C#, генерируемого визуальным конструктором) и главного файла кода C# (для ваших обработчиков событий, специальных методов и т.п.), как иллюстрируется на рис. 31.18. После копирования разметки из предыдущего примера останется лишь создать метод `GetData()` в файле отделенного кода `Default.aspx.cs`. Для начала добавьте операторы `using` для пространств имен `AutoLotDAL.Models` и `AutoLotDAL.Repos`. Затем добавьте метод `GetData()` со следующим кодом:

```
using AutoLotDAL.Models;
using AutoLotDAL.Repos;

namespace CodeBehindPageModel
{
  public partial class Default : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    public IEnumerable<Inventory> GetData()
    {
      return new InventoryRepo().GetAll();
    }
  }
}
```

В этот момент веб-приложение можно запустить, нажав комбинацию клавиш <Ctrl+F5>. Снова запустится веб-сервер IIS Express и обслужит страницу для запрашивающего браузера.

Отладка и трассировка страниц ASP.NET

Для отладки приложений Web Forms сайт должен содержать правильно сконфигурированный файл `Web.config`. Когда вы запускаете сеанс отладки, и IDE-среда запрашивает о необходимости изменения файла `Web.config` для включения отладки, ответьте утвердительно. Это значит, что в файле `Web.config` отсутствует приведенная ниже разметка (наиболее важен здесь атрибут `debug="true"`):

```
<compilation debug="true" targetFramework="4.6"/>
```

Кроме того, можно включить *поддержку трассировки* для файла `.aspx`, установив атрибут `Trace` в `true` внутри директивы `<%@ Page %>` (также возможно включить трассировку для всего сайта, модифицировав файл `Web.config`):

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeBehind="Default.aspx.cs" Inherits="CodeBehindPageModel.Default"
    Trace="true" %>
```

На заметку! Страницы веб-приложения унаследованы от класса, указанного с помощью полностью заданного имени, в этом случае `CodeBehindPageModel.Default`. Страницы веб-сайта унаследованы от класса с именем страницы, предваренным символом подчеркивания, таким как `_Default`.

В результате выпущенная разметка HTML будет содержать многочисленные детали, касающиеся предыдущего цикла "запрос/ответ" HTTP (серверные переменные, переменные сеанса и приложения, запрос/ответ и т.д.). Для вставки собственных сообщений трассировки можно применять свойство `Trace`, унаследованное от класса `System.Web.UI.Page`. Всякий раз, когда нужно занести в журнал специальное сообщение (из блока сценария или файла исходного кода C#), просто вызывайте статический метод `Trace.Write()`. Первый аргумент представляет имя специальной категории, а второй — сообщение трассировки. В целях иллюстрации измените код метода `GetData()`, как показано ниже:

```
public IEnumerable<Inventory> GetData()
{
  Trace.Write("Default.aspx","Getting Data");
  return new InventoryRepo().GetAll();
}
```

Запустите проект снова. В журнале появится специальная категория и специальное сообщение. На рис. 31.19 обратите внимание на выделенное сообщение с информацией трассировки.

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin PreInit		
aspx.page	End PreInit	0.000054	0.000054
aspx.page	Begin Init	0.000083	0.000029
aspx.page	End Init	0.035438	0.035356
aspx.page	Begin InitComplete	0.035471	0.000032
aspx.page	End InitComplete	0.035483	0.000012
aspx.page	Begin PreLoad	0.035494	0.000011
aspx.page	End PreLoad	0.035515	0.000021
aspx.page	Begin Load	0.035525	0.000010
aspx.page	End Load	0.035875	0.000350
aspx.page	Begin LoadComplete	0.035890	0.000015
aspx.page	End LoadComplete	0.041524	0.005634
aspx.page	Begin PreRender	0.041583	0.000059
Default.aspx	Getting Data	0.071696	0.030112
aspx.page	End PreRender	0.732762	0.661067
aspx.page	Begin PreRenderComplete	0.732802	0.000040
aspx.page	End PreRenderComplete	0.732812	0.000010
aspx.page	Begin SaveState	0.735502	0.002690
aspx.page	End SaveState	0.748051	0.012549
aspx.page	Begin SaveStateComplete	0.748073	0.000022
aspx.page	End SaveStateComplete	0.821923	0.073850
aspx.page	Begin Render	0.821966	0.000043
aspx.page	End Render	0.826076	0.004110

Рис. 31.19. Занесение в журнал специальных сообщений трассировки

Теперь вы знаете, как строить страницу Web Forms с использованием однофайлового подхода и подхода с отделенным кодом. Оставшийся материал этой главы посвящен анализу состава проекта Web Forms, а также способам взаимодействия с запросами/ответами HTTP и жизненному циклу класса, производного от `Page`.

Исходный код. Веб-сайт `CodeBehindPageModel` доступен в подкаталоге `Chapter_31`.

Сравнение веб-сайтов и веб-приложений ASP.NET

Перед построением нового проекта Web Forms необходимо выбрать один из двух форматов проектов: *веб-сайт ASP.NET* или *веб-приложение ASP.NET*. Такой выбор будет определять то, каким образом Visual Studio организует и обрабатывает стартовые файлы

веб-приложения, тип создаваемых начальных файлов проекта и степень контроля над результирующим составом скомпилированной сборки .NET.

Когда инфраструктура ASP.NET впервые появилась в составе платформы .NET 1.0, единственным вариантом было построение того, что сейчас называется *веб-приложением*. В рамках этой модели вы имеете непосредственный контроль над именем и местоположением скомпилированной выходной сборки.

Веб-приложения удобны, когда нужно переносить старые веб-сайты .NET 1.1 в проекты .NET 2.0 и последующих версий. Они также полезны, если требуется создать единственное решение Visual Studio, которое может содержать несколько проектов (например, веб-приложение и любые связанные библиотеки кода .NET). В предыдущих двух примерах в качестве отправной точки применялись веб-приложения ASP.NET.

На заметку! Из-за того, что шаблоны проектов ASP.NET в Visual Studio могут генерировать значительный объем стартового кода (мастер-страницы, страницы содержимого, библиотеки сценариев, страница входа и т.д.), в книге будет использоваться только шаблон пустого веб-сайта. Тем не менее, когда вы закончите чтение глав, посвященных ASP.NET, обязательно создайте новый проект веб-сайта ASP.NET и первым делом просмотрите стартовый код.

По разительному контрасту шаблоны проектов веб-сайтов ASP.NET в Visual Studio (доступные через пункт меню File⇨New Web Site (Файл⇨Создать веб-сайт)) скрывают файл `*.designer.cs` в пользу находящегося в памяти частичного класса. Более того, проекты веб-сайтов ASP.NET поддерживают несколько папок со специальными именами, такими как `App_Code`. В эту папку можно помещать любые файлы кода C# (или VB), которые напрямую не отображаются на веб-страницы, и компилятор времени выполнения по мере необходимости будет их динамически компилировать. В итоге получается значительное упрощение обычного процесса построения выделенной библиотеки кода .NET и ссылки на нее в новых проектах.

К слову, проект веб-сайта можно перенести в неизменном виде на производственный веб-сервер без предварительной компиляции сайта, которую требуется делать в случае веб-приложения ASP.NET.

В книге будут применяться типы проектов веб-сайтов ASP.NET, поскольку они упрощают процесс построения веб-приложений на платформе .NET. Однако независимо от выбранного подхода вы будете иметь доступ к той же самой модели программирования.

Включение поддержки C# 6 для веб-сайтов ASP.NET

По умолчанию поддержка C# 6 для веб-сайтов ASP.NET не включена (для проектов Web Forms она включена в стандартном шаблоне проекта). Для включения поддержки новых языковых средств C# 6 понадобится установить NuGet-пакет CodeDOM Providers for .NET Compiler Platform ("Roslyn") (Поставщики CodeDOM для платформы компилятора .NET ("Roslyn")). Чтобы установить его для веб-сайтов, щелкните правой кнопкой мыши на имени сайта в окне Solution Explorer, выберите в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet) и поищите `CodeDom`. В результате отобразится пакет `Microsoft.CodeDom.Providers.DotNetCompilerPlatform`. Щелкните на кнопке Install (Установить).

Структура каталогов веб-сайта ASP.NET

Когда создается новый проект веб-сайта ASP.NET, он может содержать несколько специфически именованных подкаталогов, каждый из которых имеет специальное значение для исполняющей среды ASP.NET. Эти специальные подкаталоги кратко описаны в табл. 31.2.

Таблица 31.2. Специальные подкаталоги ASP.NET

Подкаталог	Описание
App_Browsers	Папка для файлов определений браузеров, используемых для идентификации индивидуальных браузеров и выяснения их возможностей
App_Code	Папка для исходного кода компонентов или классов, которые должны компилироваться как составные части приложения. Исполняющая среда ASP.NET компилирует код в этой папке при запросе страниц. Код в папке App_Code автоматически доступен приложению
App_Data	Папка для хранения файлов Access (*.mdb), файлов SQL Express (*.mdf), файлов XML или других хранилищ данных
App_GlobalResources	Папка для файлов *.resx, доступных программно из кода приложения
App_LocalResources	Папка для файлов *.resx, привязанных к определенной странице
App_Themes	Папка, содержащая коллекцию файлов, которые определяют внешний вид страниц и элементов управления Web Forms
App_WebReferences	Папка для прокси-классов, схем и других файлов, связанных с использованием веб-служб в приложении
Bin	Папка для скомпилированных закрытых сборок (файлов .dll). Сборки из папки Bin автоматически доступны приложению

Любой из перечисленных в табл. 31.2 известных подкаталогов можно явно добавить в текущее веб-приложение, выбрав пункт меню Website⇨Add ASP.NET Folder (Веб-сайт⇨Добавить папку ASP.NET). Тем не менее, во многих случаях IDE-среда будет делать это автоматически при естественном добавлении связанных файлов к сайту. Например, вставка нового файла класса в проект автоматически добавит в структуру каталогов папку App_Code, если она еще не существует.

Ссылка на сборки

Несмотря на то что шаблоны веб-сайтов генерируют файл .sln для загрузки файлов .aspx в IDE-среду, файла *.csproj больше нет. Однако проекты веб-приложений ASP.NET записывают все внешние сборки в файл *.csproj. Тогда где записываются внешние сборки в ASP.NET?

Вы уже видели, что при ссылке на закрытую сборку Visual Studio автоматически создает внутри структуры каталогов подкаталог \bin для хранения локальной копии этой двоичной сборки. Когда кодовая база задействует типы из библиотеки кода, она автоматически загружается по требованию.

Если производится ссылка на разделяемую сборку, находящуюся в глобальном кеше сборок (Global Assembly Cache — GAC), то среда Visual Studio автоматически вставляет в текущее веб-решение файл Web.config (при его отсутствии) и записывает внешнюю ссылку в элемент <assemblies>. Например, если выбрать пункт меню Website⇨Add Reference (Веб-сайт⇨Добавить ссылку) и указать разделяемую сборку (такую как System.Security.dll), то файл Web.config будет обновлен следующим образом:

```
<assemblies>
  <add assembly="System.Security, Version=4.0.0.0,
       Culture=neutral, PublicKeyToken=B03F5F7F11D50A3A"/>
</assemblies>
```

Итак, каждая сборка описывается с применением одной и той же информации, требуемой для динамической загрузки посредством метода Assembly.Load() (см. главу 15).

Роль папки App_Code

Папка App_Code используется для хранения файлов кода, которые напрямую не привязаны к конкретной веб-странице (подобно файлу отделенного кода), но должны быть скомпилированы для потребления веб-сайтом. Код внутри папки App_Code будет автоматически компилироваться на лету по мере необходимости. После этого сборка доступна любому другому коду в рамках веб-сайта. В данном отношении папка App_Code очень похожа на папку Bin, но только в ней можно хранить исходный код, а не скомпилированный. Основное преимущество такого подхода в том, что появляется возможность определять специальные типы для веб-приложения, не компилируя их отдельно.

Единственная папка App_Code может содержать код на разных языках программирования. Во время выполнения для генерации сборки вызывается подходящий компилятор. Тем не менее, если предпочтительнее разбить код на части, то можно определить несколько подкаталогов для хранения любого количества файлов управляемого кода (*.vb, *.cs и т.д.). Например, предположим, что вы добавили в корневой каталог приложения веб-сайта папку App_Code с двумя подпапками, MyCSharpCode и MyVbNetCode, которые содержат файлы, специфичные для языка программирования. Затем можете обновить файл Web.config, чтобы указать эти подкаталоги с использованием элемента <codeSubDirectories>, вложенного внутрь элемента <configuration>:

```
<compilation debug="true" strict="false" explicit="true">
  <codeSubDirectories>
    <add directoryName="MyCSharpCode" />
    <add directoryName="MyVbNetCode" />
  </codeSubDirectories>
</compilation>
```

На заметку! В папке App_Code также будут храниться файлы, которые не являются языковыми, но необходимы для других целей (*.xsd, *.wsdl и т.д.).

Помимо Bin и App_Code существуют еще два дополнительных специальных подкаталога, App_Data и App_Themes, с которыми вы должны быть знакомы; оба они рассматриваются в последующих главах. Как всегда, за подробной информацией относительно оставшихся подкаталогов ASP.NET обращайтесь в документацию .NET Framework 4.6 SDK.

Цепочка наследования типа Page

Все веб-страницы .NET в конечном итоге унаследованы от класса System.Web.UI.Page. Подобно любому базовому классу этот тип предоставляет полиморфный интерфейс для всех производных типов. Однако тип Page — не единственный член иерархии наследования. Если найти в браузере объектов Visual Studio класс System.Web.UI.Page (внутри сборки System.Web.dll), то можно обнаружить, что Page "является" классом TemplateControl, который в свою очередь "является" классом Control, а тот — классом Object (рис. 31.20).

Каждый из этих базовых классов привносит приличную часть функциональности во все файлы *.aspx. В большинстве проектов будут применяться члены, определенные внутри родительских классов Page и Control. Функциональность, полученная от класса System.Web.UI.TemplateControl, интересна только при построении специальных элементов управления Web Forms или при взаимодействии с процессом визуализации.

Первый родительский класс, который мы рассмотрим — собственно Page. Он содержит многочисленные свойства, которые позволяют взаимодействовать с разнообразными веб-примитивами, такими как переменные приложения и сеанса, поддержка цикла "запрос/ответ" HTTP и т.д.

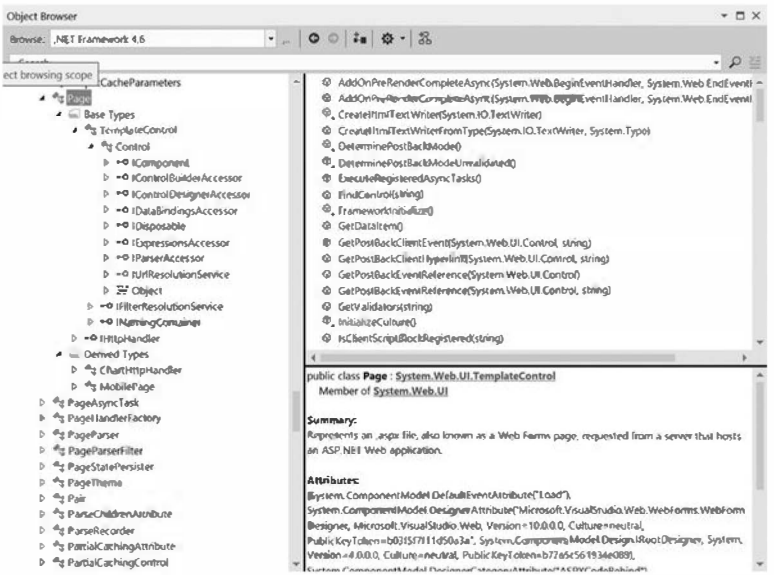


Рис. 31.20. Цепочка наследования класса `Page`

В табл. 31.3 кратко описаны некоторые (но, конечно же, не все) основные свойства.

Таблица 31.3. Избранные свойства типа `Page`

Свойство	Описание
`Application`	Позволяет взаимодействовать с данными, которые доступны по всему веб-сайту всем пользователям
`Cache`	Позволяет взаимодействовать с объектом кеша для текущего веб-сайта
`ClientTarget`	Позволяет указать, как данная страница должна визуализироваться в запрашивающем браузере
`IsPostBack`	Получает значение, которое указывает, загружается страница в ответ на клиентскую обратную отправку или при обращении к ней в первый раз
`MasterPageFile`	Устанавливает мастер-страницу для текущей страницы
`Request`	Предоставляет доступ к текущему HTTP-запросу
`Response`	Позволяет взаимодействовать с исходящим HTTP-ответом
`Server`	Предоставляет доступ к объекту `HttpServerUtility`, который содержит различные вспомогательные функции серверной стороны
`Session`	Позволяет взаимодействовать с данными сеанса для текущего вызывающего компонента
`Theme`	Получает или устанавливает имя темы, используемой для текущей страницы
`Trace`	Предоставляет доступ к объекту `TraceContext`, который позволяет заносить в журнал специальные сообщения во время сеансов отладки

Взаимодействие с входящим HTTP-запросом

Ранее в главе вы узнали, что базовый поток веб-приложения начинается с запроса веб-страницы клиентом, возможного ввода пользователем информации и щелчка на

"кнопке отправки" для обратной отправки данных формы HTML указанной веб-странице на обработку. В большинстве случаев в открывающем дескрипторе <form> присутствуют атрибуты action и method, указывающие файл на веб-сервере, которому будут отправлены данные из разнообразных виджетов HTML, и метод отправки этих данных (GET или POST):

```
<form name="defaultPage" id="defaultPage"
      action="http://localhost/Cars/ClassicAspPage.asp" method = "GET">
  ...
</form>
```

Все страницы ASP.NET поддерживают свойство System.Web.UI.Page.Request, которое обеспечивает доступ к экземпляру класса HttpRequest (основные его члены перечислены в табл. 31.4).

Таблица 31.4. Члены класса HttpRequest

Член	Описание
ApplicationPath	Получает виртуальный корневой путь к приложению ASP.NET на сервере
Browser	Предоставляет информацию о возможностях клиентского браузера
Cookies	Получает коллекцию cookie-наборов, отправленную клиентским браузером
FilePath	Указывает виртуальный путь текущего запроса
Form	Получает коллекцию переменных формы HTTP
Headers	Получает коллекцию заголовков HTTP
HttpMethod	Указывает метод передачи данных HTTP, применяемый клиентом (GET, POST или HEAD)
IsSecureConnection	Указывает, является ли подключение HTTP защищенным (т.е. HTTPS)
QueryString	Получает коллекцию переменных строки HTTP-запроса
RawUrl	Получает низкоуровневый URL текущего запроса
RequestType	Указывает метод передачи данных HTTP, используемый клиентом (GET или POST)
ServerVariables	Получает коллекцию переменных веб-сервера
UserHostAddress	Получает IP-адрес хоста удаленного клиента
UserHostName	Получает имя DNS удаленного клиента

В дополнение к указанным в табл. 31.4 членам класс HttpRequest содержит несколько полезных методов, включая описанные ниже.

- MapPath(). Отображает виртуальный путь в запрошенном URL на физический путь на сервере для текущего запроса.
- SaveAs(). Сохраняет информацию о текущем HTTP-запросе в файле на веб-сервере, что может оказаться полезным для целей отладки.
- ValidateInput(). Если включено средство проверки достоверности посредством атрибута Validate директивы Page, то этот метод можно вызывать для проверки всех введенных пользователем данных (включая cookie-наборы) по заранее определенному списку потенциально опасных входных данных.

Получение статистики о браузере

Первый интересный аспект типа `HttpRequest` связан со свойством `Browser`, которое предоставляет доступ к лежащему в основе объекту `HttpBrowserCapabilities`. В свою очередь объект `HttpBrowserCapabilities` содержит многочисленные члены, позволяющие программно исследовать статистику относительно браузера, который отправил входящий HTTP-запрос.

Создайте новый проект пустого веб-сайта ASP.NET (по имени `FunWithPageMembers`), выбрав пункт меню File⇨New Web Site (Файл⇨Создать веб-сайт). Диалоговое окно New Web Site (Новый веб-сайт) будет выглядеть так, как показано на рис. 31.21.

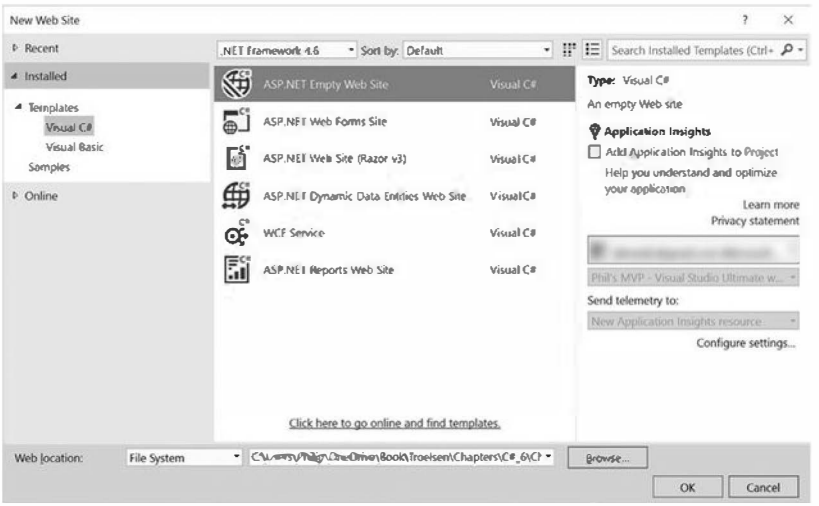


Рис. 31.21. Создание нового пустого веб-сайта

На рис. 31.21 обратите внимание, что есть возможность задать местоположение для нового веб-сайта. Если в раскрывающемся списке выбрать вариант File System (Файловая система), то файлы содержимого будут помещены в локальный каталог, а страницы будут обслуживаться веб-сервером IIS Express. В случае выбора варианта FTP или HTTP сайт будет размещен в новом виртуальном каталоге, поддерживаемом IIS.

В рассматриваемом примере не имеет значения, какой вариант будет выбран, но для простоты выберите File System.

Когда выбирается каталог, в котором уже имеется какой-то веб-сайт (или любые файлы, если уж на то пошло), в открывшемся диалоговом окне, показанном на рис. 31.22, будет предложено ввести новое имя (что приведет к созданию нового каталога).

Создав пустой веб-сайт, добавьте в проект новый файл Web Forms через пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент). Выберите Visual C# в древовидном представлении слева и назначьте файлу имя `Default.aspx`. Первая задача заключается в построении пользовательского интерфейса, который позволит пользователям щелкать на веб-элементе управления `Button` (по имени `btnGetBrowserStats`) для просмотра разнообразных статистических данных о вызывающем браузере. Статистические сведения будут генерироваться динамическим образом и затем присоединяться к элементу управления `Label` (с именем `lblOutput`). Поместите упомянутые два элемента управления в любое место графического конструктора веб-страницы. После этого обработайте событие `Click` для кнопки, добавив к ее дескриптору атрибут `OnClick` и указав метод `btnGetBrowserStats()`.

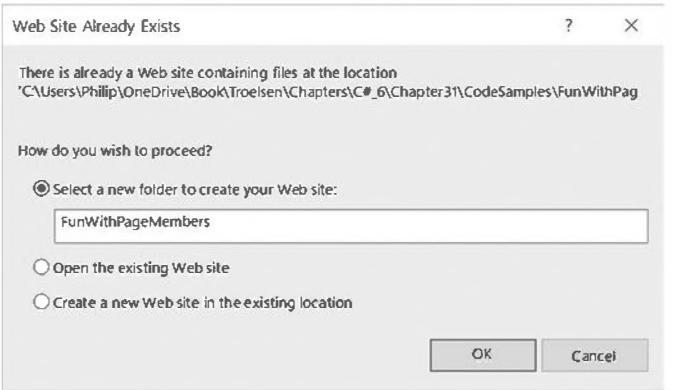


Рис. 31.22. Указание имени для нового пустого веб-сайта

Вот разметка:

```
<strong style="font-weight: 700">Basic Request / Response Info<br />
<br />
<asp:Button ID="btnGetBrowserStats" runat="server"
     OnClick="btnGetBrowserStats_Click" Text="Get Stats" />
<br />
<br />
<asp:Label ID="lblOutput" runat="server"></asp:Label>
</strong>
```

В файле отделенного кода для страницы Web Forms реализуйте обработчик, как показано далее (обратите внимание на применение интерполяции строк, которая объяснялась в главе 3):

```
protected void btnGetBrowserStats_Click(object sender, EventArgs e)
{
  string theInfo = "";
  // Клиент AOL?
  theInfo += $"<li>Is the client AOL? {Request.Browser.AOL}</li>";
  // Поддерживает ли ActiveX?
  theInfo += $"<li>Does the client support ActiveX?
{Request.Browser.ActiveXControls}</li>";
  // Бета-версия?
  theInfo += $"<li>Is the client a Beta? {Request.Browser.Beta}</li>";
  // Поддерживает ли Java-аплеты?
  theInfo += $"<li>Does the client support Java Applets?
{Request.Browser.JavaApplets}</li>";
  // Поддерживает ли cookie-наборы?
  theInfo += $"<li>Does the client support Cookies?
{Request.Browser.Cookies}</li>";
  // Поддерживает ли VBScript?
  theInfo += $"<li>Does the client support VBScript?
{Request.Browser.VBScript}</li>";
  lblOutput.Text = theInfo;
}
```

Здесь производится проверка ряда возможностей браузера. Как и можно было предположить, выяснять наличие поддержки браузером элементов ActiveX, Java-аплетов и кода VBScript клиентской стороны очень полезно. Если вызывающий браузер не поддерживает какую-то веб-технологию, то страница `.aspx` сможет предпринять альтернативные действия.

Доступ к входным данным формы

В классе HttpResponse также определены свойства Form и QueryString. Они позволяют просматривать входные данные формы в виде пар "имя-значение". Хотя свойства HttpRequest.Form и HttpRequest.QueryString можно было бы использовать для доступа к данным формы, предоставленным клиентом, на веб-сервере, ASP.NET предлагает более элегантный объектно-ориентированный подход. Учитывая, что ASP.NET снабжает вас веб-элементами управления серверной стороны, элементы пользовательского интерфейса HTML можно трактовать как настоящие объекты. Следовательно, вместо того чтобы получать значение из текстового поля как в приведенном далее коде:

```
protected void btnGetFormData_Click(object sender, System.EventArgs e)
{
  // Получить значение из виджета с идентификатором txtFirstName.
  string firstName = Request.Form("txtFirstName");
  // Использовать полученное значение на странице...
}
```

можно просто обратиться к виджету серверной стороны напрямую через свойство Text:

```
protected void btnGetFormData_Click(object sender, System.EventArgs e)
{
  // Получить значение из виджета с идентификатором txtFirstName.
  string firstName = txtFirstName.Text;
  // Использовать полученное значение на странице...
}
```

В целях иллюстрации добавьте к форме элементы управления TextBox и Button. Установите атрибут Id для TextBox в txtFirstName, а Id для Button — в btnGetFormData. Добавьте атрибут OnClick и установите его в btnGetFormData_OnClick:

```
<br/>
<label>First Name</label>
<asp:TextBox runat="server" Id="txtFirstName"/>
<asp:Button runat="server" Id="btnGetFormData"
     OnClick="btnGetFormData_Click" Text="Get First Name"/>
```

Реализуйте обработчик события btnGetFormData_Click(), как было показано выше. Запустите приложение, введите свое имя в элементе TextBox и щелкните на кнопке Get First Name (Получить имя). Введенное имя отобразится в элементе Label.

Такой подход не только соответствует основополагающим принципам объектно-ориентированного программирования, но также позволяет не беспокоиться о способе отправки данных формы (GET или POST) перед получением значений. Более того, работа с виджетом напрямую намного безопаснее в отношении типов, поскольку ошибки обнаруживаются на этапе компиляции, а не во время выполнения. Конечно, речь не идет о том, что вы *никогда* не будете работать со свойствами Form и QueryString в ASP.NET; просто потребность в них значительно снижена, поэтому применять их необязательно.

Свойство IsPostBack

Еще одним очень важным членом класса Page является свойство IsPostBack. Вспомните, что понятием "обратная отправка" обозначается ситуация, когда веб-страница отправляет данные обратно по тому же самому URL на веб-сервере. С учетом такого определения запомните, что свойство IsPostBack возвратит true, если текущий HTTP-запрос был отправлен пользователем текущего сеанса, и false, если происходит первое взаимодействие пользователя со страницей.

Необходимость в выяснении, является ли текущий HTTP-запрос на самом деле обратной отправкой, обычно возникает, когда некоторый блок кода должен выполняться только в случае, если пользователь впервые обращается к заданной странице. Например, когда пользователь получает доступ к файлу .aspx в первый раз, можно заполнить данными объект DataSet из ADO.NET и кешировать его для последующего использования. Тогда при возвращении пользователя на страницу удастся избежать излишнего обращения к базе данных (разумеется, некоторые страницы могут требовать обновления объекта DataSet в каждом запросе, но это совсем другая проблема). Предполагая, что страница .aspx обрабатывает событие Load (объясняется далее в главе), запрограммировать проверку условия обратной отправки можно следующим образом:

```
protected void Page_Load(object sender, EventArgs e)
{
  // Заполнить DataSet только при самом первом
  // входе пользователя на страницу.
  if (!IsPostBack)
  {
    // Заполнить объект DataSet и кешировать его.
  }
  // Использовать кешированный объект DataSet.
}
```

Взаимодействие с исходящим HTTP-ответом

Теперь, когда вы лучше понимаете, как тип Page позволяет взаимодействовать с входящим HTTP-запросом, необходимо научиться взаимодействовать с исходящим HTTP-ответом. В ASP.NET свойство Response класса Page предоставляет доступ к экземпляру типа HttpResponse. В типе HttpResponse определен набор свойств, которые позволяют форматировать HTTP-ответ, отправляемый обратно клиентскому браузеру. Основные свойства HttpResponse перечислены в табл. 31.5.

Таблица 31.5. Свойства типа HttpResponse

Свойство	Описание
Cache	Возвращает семантику кеширования веб-страницы (см. главу 33)
ContentEncoding	Получает или устанавливает набор символов HTTP для выходного потока
ContentType	Получает или устанавливает MIME-тип HTTP для выходного потока
Cookies	Получает коллекцию HttpCookie, которая будет возвращена браузеру
Output	Позволяет осуществлять текстовый вывод в тело исходящего HTTP-ответа
OutputStream	Позволяет осуществлять двоичный вывод в тело исходящего HTTP-ответа
StatusCode	Получает или устанавливает код состояния HTTP выходных данных, возвращаемых клиенту
StatusDescription	Получает или устанавливает строку состояния HTTP выходных данных, возвращаемых клиенту
SuppressContent	Получает или устанавливает значение, которое указывает, что HTTP-ответ не будет отправлен клиенту

В табл. 31.6 приведен частичный список методов, поддерживаемых типом HttpResponse.

Таблица 31.6. Методы типа `HttpResponse`

Метод	Описание
`Clear()`	Очищает все заголовки и выходное содержимое из буфера потока
`End()`	Отправляет весь буферизованный вывод клиенту, после чего закрывает сокетное подключение
`Flush()`	Отправляет весь текущий буферизованный вывод клиенту
`Redirect()`	Перенаправляет клиента на новый URL
`Write()`	Записывает значения в выходной поток содержимого HTTP
`WriteFile()`	Записывает файл непосредственно в выходной поток содержимого HTTP

Выпуск содержимого HTML

Вероятно, самым известным аспектом типа `HttpResponse` является способность записывать содержимое прямо в выходной поток HTTP. Метод `HttpResponse.Write()` позволяет передавать в поток любые дескрипторы HTML и/или текстовые литералы. Метод `HttpResponse.WriteFile()` еще больше развивает эту функциональность, позволяя указывать имя физического файла на веб-сервере, содержимое которого должно быть помещено в выходной поток (это очень удобно для быстрого выпуска содержимого существующего файла `.html`).

В целях демонстрации добавьте в текущий файл `.aspx` еще один элемент управления `Button`:

```
<br/>
<asp:Button runat="server" Id="btnHttpResponse"
     OnClick="btnHttpResponse_Click" Text="Get First Name"/>
```

Реализуйте обработчик события `Click` серверной стороны:

```
protected void btnHttpResponse_Click(object sender, EventArgs e)
{
  Response.Write("<b>My name is:</b><br>");
  Response.Write(this.ToString());
}
```

Роль этого вспомогательного метода (который, как и можно было предположить, вызывается некоторыми обработчиками событий на стороне сервера) довольно проста. Хотя всегда можно прибегнуть к старому подходу и генерировать дескрипторы HTML и содержимое с применением метода `Write()`, в ASP.NET такой способ встречается намного реже, чем в классическом ASP. Причина (опять-таки) связана с появлением веб-элементов управления серверной стороны. Таким образом, чтобы визуализировать блок текстовых данных в браузере, достаточно присвоить нужную строку свойству `Text` виджета `Label`.

Перенаправление пользователей

Еще одним аспектом класса `HttpResponse` является способность перенаправления пользователей на новый URL, например:

```
protected void btnWasteTime_Click(object sender, EventArgs e)
{
  Response.Redirect("http://www.facebook.com");
}
```

Если этот обработчик событий вызывается через обратную отправку клиентской стороны, то пользователь будет автоматически перенаправлен на указанный URL.

На заметку! Метод `HttpResponse.Redirect()` всегда влечет за собой взаимодействие с клиентским браузером. Если нужно просто передать управление файлу `.aspx` из того же самого виртуального каталога, то более эффективно использовать метод `HttpServerUtility.Transfer()`, доступный через унаследованное свойство `Server`.

Представленного материала вполне достаточно для оценки функциональности класса `System.Web.UI.Page`. Роль базового класса `System.Web.UI.Control` будет исследоваться в следующей главе. А теперь давайте рассмотрим жизненный цикл объекта, производного от `Page`.

Исходный код. Веб-сайт `FunWithPageMembers` доступен в подкаталоге `Chapter_31`.

Жизненный цикл страницы Web Forms

Каждая страница Web Forms имеет фиксированный жизненный цикл. Когда исполняющая среда ASP.NET принимает входящий запрос некоторого файла `.aspx`, в памяти размещается соответствующий объект производного от `System.Web.UI.Page` типа с применением стандартного конструктора этого типа. Затем инфраструктура автоматически запускает последовательность событий. По умолчанию принимается во внимание событие `Load`, в тело обработчика которого можно поместить свой специальный код:

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    Response.Write("Load event fired!");
  }
}
```

Кроме события `Load` заданный объект `Page` способен перехватывать любые основные события из табл. 31.7, которые приведены в порядке их появления (подробные сведения о событиях, возникающих на протяжении времени жизни страницы, ищите в документации .NET Framework 4.6 SDK).

Таблица 31.7. Избранные события типа `Page`

Событие	Описание
`PreInit`	Инфраструктура использует это событие для размещения в памяти всех веб-элементов управления, применения тем, установки мастер-страницы и настройки пользовательских профилей. Данное событие можно перехватить, чтобы вмешаться в процесс
`Init`	Инфраструктура использует это событие для установки свойств веб-элементов управления в их предыдущие значения посредством обратной отправки или данных состояния представления
`Load`	Когда возникает это событие, страница и ее элементы управления полностью инициализированы, а их предыдущие значения восстановлены. В данный момент можно безопасно взаимодействовать с каждым веб-виджетом
"Событие, которое инициировало обратную отправку"	Конечно, события с таким именем нет. Подобным образом просто обозначается любое событие, из-за которого браузер выполнил обратную отправку веб-серверу (вроде щелчка на кнопке)

Окончание табл. 31.7

Событие	Описание
PreRender	Привязка данных для элементов управления и конфигурирование пользовательского интерфейса выполнены, а элементы управления готовы визуализировать свои данные в исходящий HTTP-ответ
Unload	Страница и ее элементы управления завершили процесс визуализации, и объект страницы готов к уничтожению. В данный момент попытка взаимодействия с исходящим HTTP-ответом приведет к ошибке времени выполнения. Тем не менее, это событие можно перехватить для проведения любой очистки на уровне страницы (закрыть файл или подключение к базе данных, выполнить действие регистрации в журнале, освободить память, занимаемую объектами, и т.д.)

Как ни странно, IDE-среда не поддерживает обработку других событий помимо Load. В таком случае требуется вручную писать в файле кода метод с именем Page_*ИмяСобытия*. Например, вот каким образом можно обработать событие Unload:

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    Response.Write("Load event fired!");
  }
  protected void Page_Unload(object sender, EventArgs e)
  {
    // Больше невозможно помещать данные в HTTP-ответ,
    // поэтому будем записывать их в локальный файл.
    System.IO.File.WriteAllText(@"C:\MyLog.txt", "Page unloading!");
  }
}
```

На заметку! Каждое событие, определенное типом Page, работает в сочетании с делегатом System.EventHandler; следовательно, обработчики этих событий всегда принимают Object в первом параметре и EventArgs — во втором.

Роль атрибута AutoEventWireup

Чтобы организовать обработку событий для своей страницы, необходимо добавить подходящий обработчик в блок <script> или файл отделенного кода. Однако, исследуя директиву <%@ Page %>, можно заметить специфический атрибут по имени AutoEventWireup, который по умолчанию установлен в true:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

При таком стандартном поведении каждый обработчик события уровня страницы будет автоматически добавляться при вводе метода с соответствующим именем. Если же установить атрибут AutoPageWireup в false:

```
<%@ Page Language="C#" AutoEventWireup="false"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

то события уровня страницы перехватываться не будут. Включение атрибута AutoPageWireup приводит к генерации необходимой оснастки для событий внутри автоматически сгенерированного частичного класса, который был описан ранее в этой

главе. Даже если атрибут AutoEventWireup отключен, события уровня страницы все равно можно будет обрабатывать с применением логики обработки событий C#:

```
public _Default()
{
  // Явно привязаться к событиям Load и Unload.
  this.Load += Page_Load;
  this.Unload += Page_Unload;
}
```

Как и можно было ожидать, атрибут AutoEventWireup обычно оставляется во включенном состоянии.

Событие Error

Во время жизненного цикла страницы может произойти еще одно событие по имени Error. Оно возникает, когда метод унаследованного от Page типа инициирует исключение, которое не было явно обработано. Предположим, что имеется обработчик события Click для заданного элемента управления Button на странице, и внутри этого обработчика (btnGetFile_Click) предпринимается попытка записать в HTTP-ответ содержимое локального файла. Также предположим, что проверка на существование файла посредством стандартной структурированной обработки исключений *отсутствует*. Если поместить в стандартный конструктор обработчик события Error, то появится последний шанс справиться с проблемой на этой странице, прежде чем конечный пользователь получит невнятное сообщение об ошибке. Взгляните на следующий код:

```
public partial class _Default : System.Web.UI.Page
{
  void Page_Error(object sender, EventArgs e)
  {
    Response.Clear();
    Response.Write("I am sorry...I can't find a required file.<br>");
    Response.Write($"The error was: <b>{ Server.GetLastError().Message }</b>");
    Server.ClearError();
  }
  protected void Page_Load(object sender, EventArgs e)
  {
    Response.Write("Load event fired!");
  }
  protected void Page_Unload(object sender, EventArgs e)
  {
    // Больше невозможно помещать данные в HTTP-ответ,
    // поэтому будем записывать их в локальный файл.
    System.IO.File.WriteAllText(@"C:\MyLog.txt", "Page unloading!");
  }
  protected void btnPostback_Click(object sender, EventArgs e)
  {
    // Здесь ничего не происходит. Это нужно только
    // для обеспечения обратной отправки страницы.
  }
  protected void btnTriggerError_Click(object sender, EventArgs e)
  {
    System.IO.File.ReadAllText(@"C:\IDontExist.txt");
  }
}
```

Обратите внимание, что обработчик события Error начинается с очистки любого содержимого, имеющегося в HTTP-ответе, и выдачи обобщенного сообщения об ошибке. Чтобы получить доступ к конкретному объекту System.Exception, можно использовать метод HttpServerUtility.GetLastError(), доступный через унаследованное свойство Server:

```
Exception e = Server.GetLastError();
```

Перед выходом из обобщенного обработчика ошибок явно вызывается метод `HttpServerUtility.ClearError()` через свойство `Server`. Вызов `ClearError()` обязателен, т.к. он информирует исполняющую среду о том, что мы решили проблему вручную, и дальнейшая обработка не требуется. Если вы забудете сделать это, то конечному пользователю отобразится страница с ошибкой времени выполнения.

К настоящему моменту вы должны хорошо понимать структуру типа `Page`. При такой подготовке можно переключать внимание на элементы управления Web Forms, темы и мастер-страницы, рассматриваемые в последующих главах. Тем не менее, чтобы завершить данную главу, мы давайте ознакомимся с ролью файла `Web.config`.

Исходный код. Веб-сайт `PageLifeCycle` доступен в подкаталоге `Chapter_31`.

Роль файла `Web.config`

По умолчанию все приложения Web Forms на C#, созданные с помощью Visual Studio, автоматически снабжаются файлом `Web.config`. Однако если возникает необходимость добавить файл `Web.config` к веб-сайту вручную (например, когда вы работаете с однофайловой моделью и не создавали веб-решение), то это делается с применением пункта меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент). В файл `Web.config` можно добавлять настройки, которые управляют поведением веб-приложения во время выполнения.

При рассмотрении сборок .NET (в главе 14) вы узнали, что клиентские приложения могут использовать конфигурационный файл XML для инструктирования среды CLR относительно того, как она должна обрабатывать запросы привязки, проводить зондирование сборок и поддерживать другие детали времени выполнения. То же самое остается справедливым и для приложений Web Forms, но с заметным отличием в том, что веб-ориентированные конфигурационные файлы всегда именуются `Web.config` (в отличие от конфигурационных файлов `*.exe`, которые называются по имени связанной клиентской исполняемой сборки).

Полная структура файла `Web.config` довольно многословна. Тем не менее, в табл. 31.8 приведен обзор наиболее интересных элементов, которые можно обнаружить в `Web.config`.

Таблица 31.8. Избранные элементы файла `Web.config`

Элемент	Описание
`<appSettings>`	Этот элемент служит для установления специальных пар "имя-значение", которые с помощью типа `ConfigurationManager` можно программно прочитать в память для использования страницами
`<authentication>`	Этот элемент, относящийся к безопасности, служит для определения режима аутентификации для данного веб-приложения
`<authorization>`	Этот элемент, также связанный с безопасностью, применяется для определения того, какие пользователи имеют доступ к тем или иным ресурсам на веб-сервере
`<connectionStrings>`	Этот элемент используется для хранения строк внешних подключений, которые применяются внутри веб-сайта
`<customErrors>`	Этот элемент используется для указания исполняющей среде точного способа сообщения об ошибках, возникающих во время функционирования веб-приложения

Окончание табл. 31.8

Элемент	Описание
`<globalization>`	Этот элемент применяется для конфигурирования настроек глобализации для данного веб-приложения
`<namespaces>`	Этот элемент содержит список всех пространств имен, подлежащих включению, если веб-приложение было заранее скомпилировано с использованием нового инструмента командной строки `aspnet_compiler.exe`
`<sessionState>`	Этот элемент применяется для управления тем, как и где исполняющая среда .NET будет хранить данные состояния сеанса
`<trace>`	Этот элемент используется для включения (или отключения) поддержки трассировки для данного веб-приложения

Кроме набора, представленного в табл. 31.8, файл `Web.config` может содержать дополнительные элементы. Подавляющее большинство этих элементов связано с безопасностью, в то время как остальные полезны только в расширенных сценариях работы с ASP.NET, таких как создание специальных заголовков HTTP или специальных модулей HTTP (темы, которые здесь не раскрываются).

Утилита администрирования веб-сайтов ASP.NET

Несмотря на то что содержимое файла `Web.config` всегда можно модифицировать прямо в Visual Studio, для веб-проектов Web Forms доступен удобный веб-ориентированный редактор, который позволяет графически редактировать многочисленные элементы и атрибуты файла `Web.config`. Чтобы запустить этот инструмент, выберите пункт меню Website⇨ASP.NET Configuration (Веб-сайт⇨Конфигурация ASP.NET).

Просмотрев содержимое вкладок в верхней части страницы, вы легко заметите, что большая часть функциональности инструмента предназначена главным образом для установления настроек безопасности веб-сайта. Однако данный инструмент также делает возможным добавление настроек к элементу `<appSettings>`, определение параметров отладки и трассировки, а также установку стандартной страницы с сообщением об ошибке.

Вы увидите этот инструмент в действии, когда он понадобится, а сейчас имейте в виду, что данная утилита *не* позволяет добавлять в файл `Web.config` абсолютно все возможные настройки. Почти наверняка будут возникать ситуации, когда файл `Web.config` придется изменять вручную с применением обычного текстового редактора.

Резюме

По сравнению с созданием традиционных настольных приложений построение веб-приложений требует иного образа мышления. В настоящей главе вы ознакомились с рядом основных тем, включая HTML, HTTP, роль сценариев клиентской стороны и роль сценариев серверной стороны, в которых используется классический ASP. Материал главы по большей части был посвящен исследованию архитектуры страницы ASP.NET. Вы видели, что каждый файл `*.aspx` в проекте имеет ассоциированный с ним класс, производный от `System.Web.UI.Page`. С применением такого объектно-ориентированного подхода инфраструктура ASP.NET позволяет создавать многократно используемые объектно-ориентированные системы.

После рассмотрения основной функциональности, предлагаемой цепочкой наследования страницы, в главе было показано, как страницы в конечном итоге компилируются в допустимую сборку .NET. Глава завершилась обсуждением роли файла `Web.config` и кратким обзором инструмента администрирования веб-сайтов ASP.NET.

ГЛАВА 32

Веб-элементы управления, мастер-страницы и темы ASP.NET

Основное внимание в предыдущей главе было сосредоточено на общей структуре страницы Web Forms и роли класса `Page`. В этой главе мы погрузимся в детали *веб-элементов управления*, которые составляют пользовательский интерфейс страницы. После исследования общей природы элемента управления Web Forms вы узнаете, как использовать многие элементы пользовательского интерфейса, включая элементы управления проверкой достоверности и разнообразные приемы привязки данных.

Значительная порция настоящей главы будет посвящена обсуждению роли *мастер-страниц*, которые обеспечивают упрощенный способ установления общего скелета пользовательского интерфейса, повторяющегося на страницах веб-сайта. С темой мастер-страниц тесно связано применение элементов управления навигацией по сайту (а также файла `*.sitemap`), которые позволяют определять навигационную структуру многостраничного сайта посредством файла XML серверной стороны.

В завершение вы ознакомитесь с ролью тем Web Forms. Концептуально темы служат той же цели, что и каскадные таблицы стилей (CSS); однако темы Web Forms применяются на веб-сервере (в противоположность клиентскому браузеру) и по этой причине имеют доступ к ресурсам серверной стороны.

Природа веб-элементов управления

Главным преимуществом инфраструктуры Web Forms является возможность собирать пользовательский интерфейс из страниц с использованием типов, определенных в пространстве имен `System.Web.UI.WebControls`. Как было показано, такие элементы управления (называемые *серверными элементами управления*, *веб-элементами управления* или *элементами управления Web Forms*) исключительно полезны в том, что автоматически генерируют необходимую разметку HTML для запрашивающего браузера и открывают доступ к набору событий, которые могут быть обработаны на веб-сервере. Более того, поскольку каждый элемент управления Web Forms имеет соответствующий класс в пространстве имен `System.Web.UI.WebControls`, им можно манипулировать в объектно-ориентированной манере.

При конфигурировании свойств веб-элемента управления в окне Properties (Свойства) среды Visual Studio изменения фиксируются в открывающем дескрипторе данного элемента внутри файла *.aspx как последовательность пар "имя-значение". Таким образом, если добавить новый элемент TextBox в визуальном конструкторе и изменить его свойства ID, BorderStyle, BorderWidth, BackColor и Text, то открывающий дескриптор <asp:TextBox> будет соответствующим образом модифицирован (тем не менее, обратите внимание, что значение Text становится внутренним текстом в области TextBox):

```
<asp:TextBox ID="txtNameTextBox" runat="server" BackColor="#C0FFC0"
    BorderStyle="Dotted" BorderWidth="3px">Enter Your Name</asp:TextBox>
```

С учетом того, что объявление веб-элемента управления в итоге становится переменной-членом типа из пространства имен System.Web.UI.WebControls (посредством цикла динамической компиляции, упомянутого в главе 31), с членами этого типа можно взаимодействовать внутри блока <script> серверной стороны или более распространенным способом — через файл отделенного кода страницы. Следовательно, если добавить в файл *.aspx новый элемент управления Button, то можно написать обработчик события Click серверной стороны, в котором будет изменяться цвет фона элемента TextBox:

```
partial class _Default : System.Web.UI.Page
{
  protected void btnChangeTextBoxColor_Click(object sender, EventArgs e)
  {
    // Изменить цвет фона объекта TextBox в коде.
    this.txtNameTextBox.BackColor = System.Drawing.Color.DarkBlue;
  }
}
```

Все элементы управления Web Forms в конечном счете являются производными от общего базового класса System.Web.UI.WenControls.WebControl, который в свою очередь унаследован от System.Web.UI.Control (а тот — от System.Object). Классы Control и WebControl определяют несколько свойств, общих для всех элементов управления серверной стороны. Прежде чем изучать унаследованную функциональность, давайте формализуем понятие обработки события серверной стороны.

Обработка события серверной стороны

Учитывая текущее состояние World Wide Web, обойтись без понимания фундаментальной природы взаимодействия браузеров и веб-серверов невозможно. Всякий раз, когда эти две сущности взаимодействуют, происходит лишенный состояния цикл "запрос/ответ" HTTP. Хотя серверные элементы управления Web Forms выполняют немалую работу, изолируя вас от низкоуровневых деталей протокола HTTP, всегда помните о том, что восприятие World Wide Web как управляемой событиями сущности — всего лишь маскировка, которую обеспечивает платформа .NET. Здесь нет ничего общего с управляемой событиями моделью, которая поддерживается основанной на Windows инфраструктурой для построения графических пользовательских интерфейсов, такой как WPF.

Например, несмотря на то, что в пространстве имен System.Windows.Controls из WPF и в пространстве имен System.Web.UI.WebControls из Web Forms определены классы с совпадающими простыми именами (Button, TextBox, Label и т.д.), они не предлагают одинаковые наборы свойств, методов или событий. Скажем, обработать событие MouseMove серверной стороны, когда пользователь перемещает курсор над элементом управления Button из Web Forms, не получится.

Суть в том, что заданный элемент управления Web Forms будет открывать доступ к ограниченному набору событий, которые в итоге приводят к обратной отправке веб-серверу. Любая необходимая обработка событий клиентской стороны потребует написания фрагментов сценарного кода JavaScript/VBScript *клиентской стороны*, которые будут обрабатываться механизмом сценариев запрашивающего браузера. Поскольку Web Forms является главным образом технологией серверной стороны, тема написания сценариев клиентской стороны здесь не рассматривается.

На заметку! Обработка события для заданного веб-элемента управления с применением Visual Studio может делаться в такой же манере, как для элемента управления графического пользовательского интерфейса Windows. Просто выберите виджет на поверхности визуального конструктора и щелкните на значке с изображением молнии в окне Properties.

Свойство AutoPostBack

Полезно также упомянуть, что многие элементы управления Web Forms поддерживают свойство по имени `AutoPostBack` (в особенности элементы управления `CheckBox`, `RadioButton` и `TextBox`, а также любые элементы, производные от абстрактного класса `ListControl`). По умолчанию свойство `AutoPostBack` установлено в `false`, что отключает немедленную обратную отправку серверу (даже при наличии обработчика события в файле отделенного кода). В большинстве случаев именно это поведение и требуется, поскольку таким элементам пользовательского интерфейса, как флажки, обычно не нужна функциональность обратной отправки. Другими словами, выполнять обратную отправку немедленно после отметки или снятия отметки с флажка нежелательно, т.к. объект страницы может получить состояние виджета внутри более естественного обработчика события `Click` для `Button`.

Однако если необходимо заставить любой из этих виджетов немедленно выполнять обратную отправку обработчику события серверной стороны, то следует установить свойство `AutoPostBack` в `true`. Такой прием удобен, когда состояние одного виджета должно автоматически заполнять значение внутри другого виджета на той же самой странице. В целях иллюстрации предположим, что есть веб-страница, которая содержит элемент `TextBox` (по имени `txtAutoPostback`) и элемент `ListBox` (с именем `lstTextBoxData`).

Ниже показана соответствующая разметка:

```
<form id="form1" runat="server">
  <asp:TextBox ID="txtAutoPostback" runat="server"></asp:TextBox>
  <br/>
  <asp:ListBox ID="lstTextBoxData" runat="server"></asp:ListBox>
</form>
```

Если обработано событие `TextChanged` элемента `TextBox`, то в обработчике события серверной стороны можно было бы попытаться заполнить элемент управления `ListBox` текущим значением `TextBox`:

```
partial class _Default : System.Web.UI.Page
{
  protected void txtAutoPostback_TextChanged(object sender, EventArgs e)
  {
    lstTextBoxData.Items.Add(txtAutoPostback.Text);
  }
}
```

После запуска приложения в том виде, как есть, при вводе текста в TextBox выяснится, что ничего не происходит. Более того, если ввести что-нибудь в TextBox и перейти к следующему элементу управления с помощью клавиши <Tab>, также ничего не происходит. Причина в том, что по умолчанию свойство AutoPostBack элемента TextBox установлено в false. Тем не менее, если установить это свойство в true:

```
<asp:TextBox ID="txtAutoPostback" runat="server"
             AutoPostBack="true" ... >
</asp:TextBox>
```

то при покидании TextBox по нажатию <Tab> или <Enter> элемент управления ListBox автоматически заполнится текущим значением TextBox. Конечно, помимо необходимости заполнения элементов одного виджета на основе значения другого виджета изменять состояние свойства AutoPostBack виджета обычно не придется (даже в такой ситуации задачу можно решить внутри клиентского сценария, устраняя потребность во взаимодействии с сервером).

Базовые классы Control и WebControl

Базовый класс System.Web.UI.Control определяет разнообразные свойства, методы и события, которые предоставляют возможность взаимодействия с основными (обычно не имеющими отношения к графическому пользовательскому интерфейсу) аспектами веб-элемента управления. В табл. 32.1 документированы некоторые члены, представляющие интерес.

Таблица 32.1. Избранные члены System.Web.UI.Control

Член	Описание
Controls	Это свойство получает объект ControlCollection, который представляет дочерние элементы управления внутри текущего элемента
DataBind()	Этот метод привязывает источник данных к вызывающему серверному элементу управления и всем его дочерним элементам
EnableTheming	Это свойство устанавливает, поддерживает ли элемент функциональность тем (стандартное значение — true)
HasControls()	Этот метод определяет, содержит ли серверный элемент управления какие-то дочерние элементы
ID	Это свойство получает и устанавливает программный идентификатор серверного элемента управления
Page	Это свойство получает ссылку на экземпляр Page, который содержит данный серверный элемент управления
Parent	Это свойство получает ссылку на родительский элемент данного серверного элемента управления в иерархии элементов управления страницы
SkinID	Это свойство получает или устанавливает обложку для применения к элементу управления, позволяя определять внешний вид и поведение с использованием ресурсов серверной стороны
Visible	Это свойство получает или устанавливает значение, указывающее на то, будет ли серверный элемент управления визуализироваться как элемент пользовательского интерфейса на странице

Перечисление содержащихся элементов управления

Первым исследуемым аспектом класса `System.Web.UI.Control` будет тот факт, что все веб-элементы управления (включая `Page`) наследуют коллекцию специальных элементов управления (доступную через свойство `Controls`). Во многом подобно приложениям Windows Forms свойство `Controls` предоставляет доступ к строго типизованной коллекции объектов производных от `WebControl` типов. Как и любая коллекция .NET, она позволяет динамически добавлять, вставлять и удалять элементы во время выполнения.

Наряду с тем, что формально возможно добавлять веб-элементы управления прямо к объекту производного от `Page` типа, проще (и надежнее) применять элемент управления `Panel`. Класс `Panel` представляет контейнер виджетов, которые могут быть или не быть видимыми конечному пользователю (в зависимости от значений их свойств `Visible` и `BorderStyle`).

Для примера создайте новый проект пустого веб-сайта по имени `DynamicCtrls` и добавьте в него веб-форму. С помощью визуального конструктора страниц Visual Studio добавьте элемент типа `Panel` (с именем `myPanel`), который содержит в себе виджеты `TextBox`, `Button` и `HyperLink` с произвольными именами (имейте в виду, что визуальный конструктор требует перетаскивания внутренних элементов на поверхность пользовательского интерфейса элемента управления `Panel`). За пределами `Panel` разместите виджет `Label` (по имени `lblControlInfo`), который будет содержать визуализированный вывод. Вот одна из возможных разметок HTML:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Dynamic Control Test</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
    <hr />
    <h1>Dynamic Controls</h1>
    <asp:Label ID="lblTextBoxText" runat="server"></asp:Label>
    <hr />
    </div>
    <!-- Элемент Panel содержит три элемента управления -->
    <asp:Panel ID="myPanel" runat="server" Width="200px"
         BorderColor="Black" BorderStyle="Solid" >
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox><br/>
      <asp:Button ID="Button1" runat="server" Text="Button"/><br/>
      <asp:HyperLink ID="HyperLink1" runat="server">HyperLink
      </asp:HyperLink>
    </asp:Panel>
    <br />
    <br />
    <asp:Label ID="lblControlInfo" runat="server"></asp:Label>
  </form>
</body>
</html>
```

При такой разметке поверхность визуального конструктора страницы будет выглядеть примерно так, как показано на рис. 32.1.

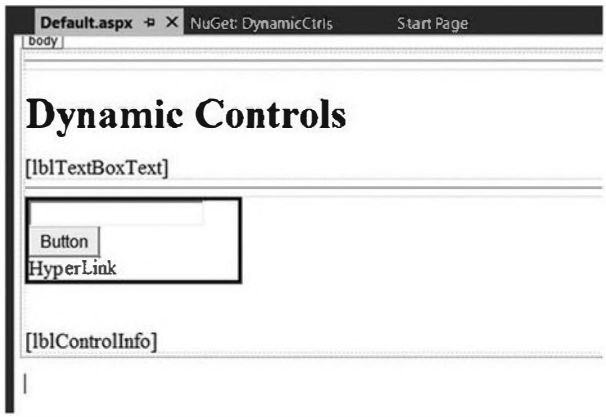


Рис. 32.1. Пользовательский интерфейс веб-страницы в проекте `DynamicCtrls`

Предположим, что в обработчике события `Page_Load()` необходимо получить детальную информацию об элементах управления, содержащихся внутри `Panel`, и присвоить результат элементу управления `Label` (`lblControlInfo`). Взгляните на следующий код C#:

```
public partial class _Default : System.Web.UI.Page
{
 protected void Page_Load(object sender, System.EventArgs e)
 {
   ListControlsInPanel();
 }
 private void ListControlsInPanel()
 {
   var theInfo = "";
   theInfo = $"<b>Does the panel have controls? {myPanel.HasControls()} </b><br/>";
   // Получить все элементы управления в панели.
   foreach (Control c in myPanel.Controls)
   {
    if (!object.ReferenceEquals(c.GetType(), typeof(System.Web.UI.LiteralControl)))
    {
      theInfo += "**************************<br/>";
      theInfo += $"Control Name? {c} <br/>";     // Имя элемента управления
      theInfo += $"ID? {c.ID} <br>";     // Идентификатор элемента управления
      theInfo += $"Control Visible? {c.Visible} <br/>";
                 // Является ли элемент управления видимым
      theInfo += $"ViewState? {c.EnableViewState} <br/>";
                // Включено ли состояние представления
    }
  }
  lblControlInfo.Text = theInfo;
 }
}
```

**Здесь осуществляется проход по всем объектам `WebControl`, которые обслуживает элемент управления `Panel`, с проверкой их принадлежности к типу `System.Web.UI.LiteralControl`; объекты этого типа пропускаются. Класс `LiteralControl` используется для представления литеральных дескрипторов HTML и содержимого (
, текстовых литералов и т.д.). Если не предпринять такой проверки, то внутри `Panel` будет обнаружено намного больше элементов управления (для приведенного выше объявления `*.aspx`). Предполагая, что элемент управления не является литеральным содержимым HTML, для него выводятся разнообразные статистические сведения. Результат показан на рис. 32.2.**

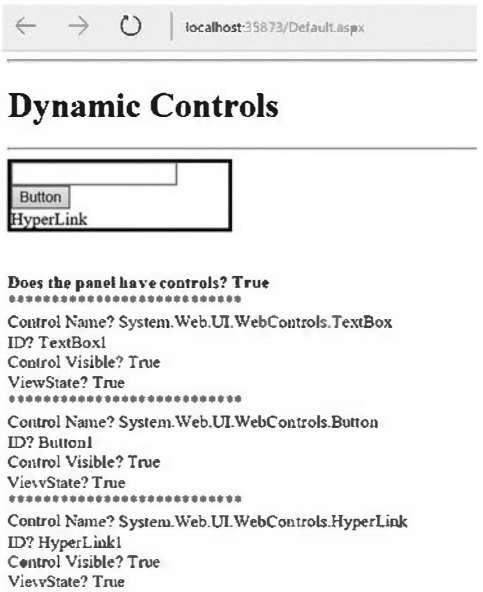


Рис. 32.2. Перечисление элементов управления во время выполнения

Динамическое добавление и удаление элементов управления

А что, если требуется изменять содержимое `Panel` во время выполнения? Давайте поместим на текущую страницу элемент `Button` (по имени `bntAddWidgets`), который будет динамически добавлять к `Panel` три новых элемента управления `TextBox`, и еще один элемент `Button` (с именем `btnRemovePanelItems`), который очистит `Panel` от всех вложенных элементов управления. Ниже представлены обработчики события `Click` для обеих кнопок:

```
protected void btnClearPanel_Click(object sender, System.EventArgs e)
{
  // Очистить содержимое панели, затем заново перечислить элементы.
  myPanel.Controls.Clear();
  ListControlsInPanel();
}

protected void btnAddWidgets_Click(object sender, System.EventArgs e)
{
  for (int i = 0; i < 3; i++)
  {
    // Присвоить идентификатор, чтобы позже можно было получить
    // текстовое значение с использованием входных данных формы.
    TextBox t = new TextBox {ID = $"newTextBox{i}"};
    myPanel.Controls.Add(t);
    ListControlsInPanel();
  }
}
```

Обратите внимание, что каждому элементу управления `TextBox` назначается уникальный идентификатор (`newTextBox0`, `newTextBox1` и т. д.). Запустив приложение, можно добавлять новые элементы к элементу управления `Panel` и полностью очищать содержимое `Panel`.

Взаимодействие с динамически созданными элементами управления

Получать значения из динамически сгенерированных элементов управления TextBox можно разными способами. Добавьте к пользовательскому интерфейсу еще один элемент управления Button (по имени btnGetTextData) и последний элемент Label с именем lblTextBoxData, после чего обработайте событие Click для Button.

Существует несколько способов доступа к данным в динамически сгенерированных элементах TextBox. Один из подходов предусматривает проход в цикле по всем элементам, которые содержатся во входных данных формы HTML (доступных через HttpRequest.Form), и накопление текстовой информации в локальной строке System.String. После прохождения всей коллекции результирующую строку необходимо присвоить свойству Text нового элемента Label:

```
protected void btnGetTextData_Click(object sender, System.EventArgs e)
{
  string textBoxValues = "";
  for (int i = 0; i < Request.Form.Count; i++)
  {
    textBoxValues += $"<li>{ Request.Form[i]}</li><br/>";
  }
  lblTextBoxData.Text = textBoxValues;
}
```

Запустив приложение, вы увидите, что содержимое каждого текстового поля можно просматривать в виде довольно длинной (нечитабельной) строки. Эта строка содержит *состояние представления* каждого элемента управления на странице. Роль состояния представления обсуждается в главе 33.

Для получения более ясного вывода текстовые данные можно разнести по уникально именованным элементам (newTextBox0, newTextBox1 и newTextBox2). Взгляните на следующую модификацию:

```
protected void btnGetTextData_Click(object sender, System.EventArgs e)
{
  // Получить текстовые поля по их именам.
  string lableData = $"<li>{Request.Form.Get("newTextBox0")}</li><br/>";
  lableData += $"<li>{Request.Form.Get("newTextBox1")}</li><br/>";
  lableData += $"<li>{Request.Form.Get("newTextBox2")}</li><br/>";
  lblTextBoxData.Text = lableData;
}
```

Применяя любой из подходов, вы заметите, что как только запрос обработан, текстовые поля исчезают. И снова причина кроется в не поддерживающей состояние природе HTML. Чтобы сохранять эти динамически созданные элементы управления TextBox между обратными отправками, нужно использовать приемы программирования с состоянием Web Forms (см. главу 33).

Исходный код. Веб-сайт DynamicCtrls доступен в подкаталоге Chapter_32.

Функциональность базового класса WebControl

Как уже известно, класс Control обладает несколькими линиями поведения, которые не связаны с графическим пользовательским интерфейсом (коллекция элементов управления, поддержка автоматической обратной отправки и т.д.). С другой стороны, базовый класс WebControl предоставляет графический полиморфный интерфейс всем веб-виджетам (табл. 32.2).

Таблица 32.2. Избранные свойства базового класса `WebControl`

Свойство	Описание
`BackColor`	Получает или устанавливает цвет фона веб-элемента управления
`BorderColor`	Получает или устанавливает цвет контура веб-элемента управления
`BorderStyle`	Получает или устанавливает стиль контура веб-элемента управления
`BorderWidth`	Получает или устанавливает ширину контура веб-элемента управления
`Enabled`	Получает или устанавливает значение, которое указывает, доступен ли веб-элемент управления
`CssClass`	Позволяет назначить веб-элементу управления класс, определенный в каскадной таблице стилей
`Font`	Получает информацию о шрифте веб-элемента управления
`ForeColor`	Получает или устанавливает цвет переднего плана (обычно цвет текста) веб-элемента управления
`Height, Width`	Получает или устанавливает высоту и ширину веб-элемента управления
`TabIndex`	Получает или устанавливает индекс обхода по клавише <Tab> веб-элемента управления
`ToolTip`	Получает или устанавливает всплывающую подсказку веб-элемента управления, появляющуюся при наведении на него курсора мыши

Почти все эти свойства самоочевидны, поэтому вместо того, чтобы демонстрировать их применение по одиночке, давайте посмотрим на множество элементов управления Web Forms в действии.

Основные категории элементов управления Web Forms

Библиотека элементов управления Web Forms может быть разбита на ряд обширных категорий, которые видны в панели инструментов Visual Studio (при условии, что в визуальном конструкторе открыта страница `*.aspx`), как показано на рис. 32.3.

В области Standard (Стандартные) панели инструментов находятся самые часто используемые элементы управления, включая `Button`, `Label`, `TextBox` и `ListBox`. Помимо этих простых элементов пользовательского интерфейса в области Standard также доступны и более экзотические веб-элементы управления, такие как `Calendar`, `Wizard` и `AdRotator` (рис. 32.4).

Область Data (Данные) содержит набор элементов управления, применяемых для операций привязки данных, в том числе элемент управления Web Forms под названием `Chart`, позволяющий визуализировать график (круговую диаграмму, гистограмму и т.д.), который обычно является результатом операции привязки данных (рис. 32.5).

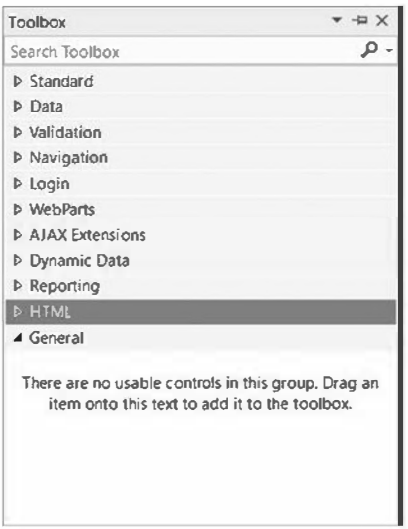


Рис. 32.3. Категории элементов управления Web Forms

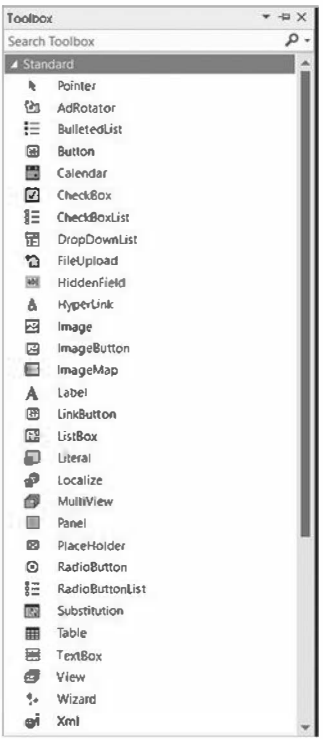


Рис. 32.4. Стандартные элементы управления Web Forms

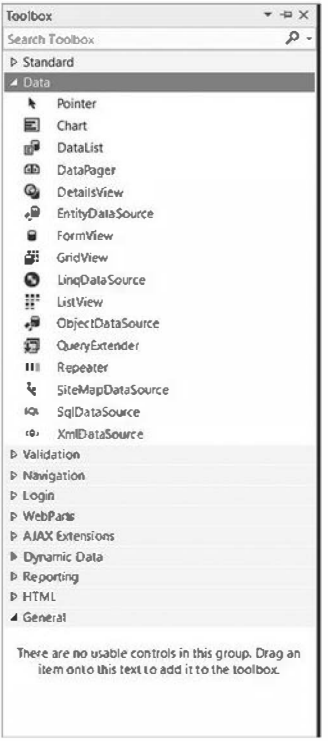


Рис. 32.5. Элементы управления Web Forms, ориентированные на данные

Элементы управления проверкой достоверности Web Forms (находящиеся в области Validation (Проверка достоверности) панели инструментов) очень интересны тем, что их можно конфигурировать для выпуска блоков кода JavaScript клиентской стороны, которые будут проверять допустимость данных в полях ввода. Если происходит ошибка проверки достоверности, то пользователь увидит сообщение об ошибке и не сможет выполнить обратную отправку на сервер до тех пор, пока не устранит ошибку.

В области Navigation (Навигация) панели инструментов располагается небольшой набор элементов управления (`Menu`, `SiteMapPath` и `TreeView`), которые обычно работают в сочетании с файлом `*.sitemap`. Как уже кратко упоминалось ранее в главе, элементы управления навигацией позволяют описывать структуру многостраничного сайта, используя дескрипторы XML.

По-настоящему экзотическим набором элементов управления Web Forms можно назвать элементы в области Login (Вход), показанные на рис. 32.6.

Эти элементы управления могут радикально упростить внедрение в веб-приложения базовых средств безопасности (восстановление пароля, экраны входа и т.п.). В действительности они настолько мощные, что даже будут динамически создавать выделенную базу данных для хранения регистрационных данных (в папке `App_Data` веб-сайта), если специфическая база данных для целей безопасности отсутствует.

На заметку! Оставшиеся категории веб-элементов управления, отображаемые в панели инструментов Visual Studio (WebParts (Веб-части), AJAX Extensions (Расширения AJAX) и Dynamic Data (Динамические данные)), предназначены для решения более специализированных задач программирования и здесь не рассматриваются.

Несколько слов о пространстве имен System.Web.UI.HtmlControls

По правде говоря, с инфраструктурой Web Forms поставляются два отдельных инструментальных набора веб-элементов управления. В дополнение к элементам управления Web Forms (внутри пространства имен `System.Web.UI.WebControls`) библиотеки базовых классов также предоставляют элементы управления HTML в пространстве имен `System.Web.UI.HtmlControls`.

Элементы управления HTML — это коллекция типов, которые позволяют применять традиционные элементы управления HTML на странице Web Forms. Однако в отличие от простых дескрипторов HTML элементы управления HTML являются объектно-ориентированными сущностями, которые могут быть сконфигурированы для запуска на сервере и, соответственно, поддерживать обработку событий серверной стороны. В отличие от элементов управления Web Forms элементы управления HTML довольно просты по своей природе и предлагают лишь небольшую функциональность сверх той, что обеспечивают стандартные дескрипторы HTML (`HtmlButton`, `HtmlInputControl`, `HtmlTable` и т.д.).

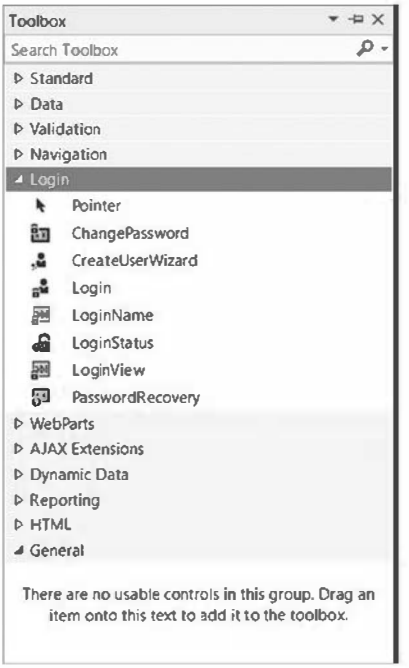


Рис. 32.6. Элементы управления Web Forms, связанные с безопасностью

Элементы управления HTML могут быть полезны, если команда четко разделена на тех, кто занимается построением пользовательских интерфейсов HTML, и разработчиков .NET. Специалисты по HTML могут использовать свой веб-редактор, имея дело со знакомыми дескрипторами разметки, и передавать готовые файлы HTML команде разработки. Затем разработчики .NET могут сконфигурировать элементы управления HTML для выполнения в качестве серверных элементов управления (посредством контекстного меню, открываемого по щелчку правой кнопкой мыши на элементе управления HTML в Visual Studio). Такой подход позволит разработчикам обрабатывать события серверной стороны и программно взаимодействовать с виджетами HTML.

Элементы управления HTML предоставляют открытый интерфейс, который имитирует стандартные атрибуты HTML. Например, для получения информации из области ввода применяется свойство `Value` вместо свойства `Text`, принятого у веб-элементов управления. Поскольку элементы управления HTML не настолько многофункциональны, как элементы управления Web Forms, больше в книге они упоминаться не будут.

Документация по веб-элементам управления

В оставшемся материале книги у вас еще будет шанс поработать с элементами управления Web Forms; тем не менее, вы определенно должны уделить время ознакомлению с описанием пространства имен `System.Web.UI.WebControls` в документации .NET Framework 4.6 SDK. Там вы найдете объяснения и примеры кода для каждого члена этого пространства имен (рис. 32.7).

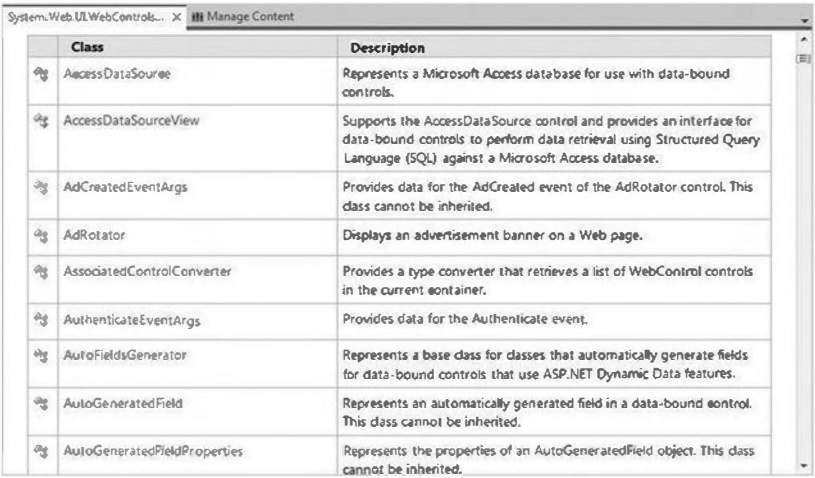


Рис. 32.7. Все элементы управления Web Forms описаны в документации .NET Framework 4.6 SDK

Построение веб-сайта Web Forms для работы с автомобилями

Учитывая, что многие "простые" элементы управления выглядят и ведут себя близко к своим аналогам из графического пользовательского интерфейса Windows, детали базовых виджетов (`Button`, `Label`, `TextBox` и т.д.) подробно рассматриваться не будут. Взамен давайте построим веб-сайт, который будет иллюстрировать работу с несколькими более экзотическими элементами управления, а также с моделью мастер-страниц Web Forms и аспектами механизма привязки данных. В частности, в примере будут продемонстрированы следующие приемы:

- работа с мастер-страницами;
- работа с навигацией посредством карты сайта;
- работа с элементом управления `GridView`;
- работа с элементом управления `Wizard`.

Для начала создайте проект пустого веб-сайта по имени `AppNetCarsSite`. Обратите внимание, что пока не создается новый полный проект веб-сайта ASP.NET, т.к. в нем предусмотрено несколько начальных файлов, которые еще не рассматривались. В текущем проекте все необходимое будет добавляться вручную.

Работа с мастер-страницами Web Forms

Многие веб-сайты обеспечивают согласованный внешний вид и поведение, которые распространяются на целый набор страниц (общая система навигации с помощью меню, общее содержимое заголовков и нижних колонтитулов, логотип компании и т.п.). Мастер-страница является всего лишь страницей Web Forms, которая имеет файловое расширение `*.master`. Сами по себе мастер-страницы не являются просматриваемыми в браузере клиентской стороны (на самом деле исполняющая среда ASP.NET не обслуживает веб-содержимое такого рода). Вместо этого мастер-страницы определяют общую компоновку пользовательского интерфейса, разделяемую всеми страницами (либо их подмножеством) на сайте.

Кроме того, страница `*.master` будет определять разнообразные области-заполнители содержимого, которые образуют раздел пользовательского интерфейса, куда могут подключаться другие файлы `*.aspx`. Вы увидите, что файлы `*.aspx`, которые включают свое содержимое в мастер-страницу, выглядят и ведут себя немного иначе, чем те файлы `*.aspx`, с которыми вы сталкивались до сих пор. В частности, такая разновидность файла `*.aspx` называется *страницей содержимого*. Страницы содержимого представляют собой файлы `*.aspx`, в которых не определен HTML-элемент `<form>` (это относится к работе мастер-страницы).

Однако с точки зрения конечного пользователя запрос производится к заданному файлу `*.aspx`. На веб-сервере соответствующий файл `*.master` и любые связанные страницы содержимого `*.aspx` смешиваются вместе в единственное объявление страницы HTML. Для демонстрации использования мастер-страниц и страниц содержимого вставьте в разрабатываемый веб-сайт новую мастер-страницу посредством пункта меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент); результирующее диалоговое окно приведено на рис. 32.8.

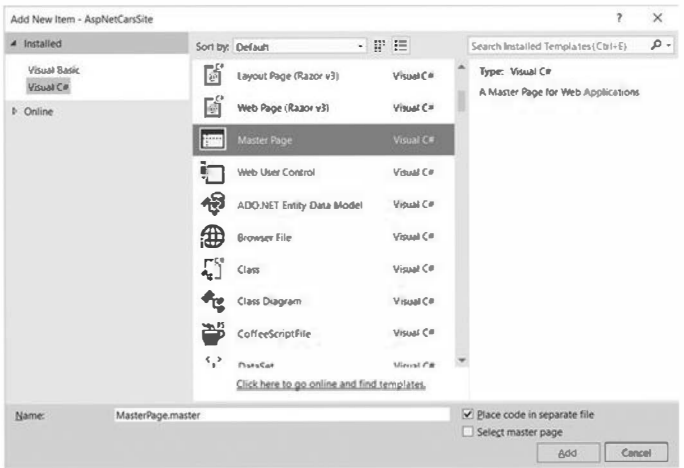


Рис. 32.8. Вставка нового файла `*.master`

Начальная разметка файла `MasterPage.master` выглядит следующим образом:

```
<%@ Master Language="C#" AutoEventWireup="true"
    CodeFile="MasterPage.master.cs" Inherits="MasterPage" %>

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
  <asp:ContentPlaceHolder id="head" runat="server">
  </asp:ContentPlaceHolder>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
      </asp:ContentPlaceHolder>
    </div>
  </form>
</body>
</html>
```

Первый интересный момент связан с новой директивой <%@ Master %>. Большей частью эта директива поддерживает те же самые атрибуты, что и директива <%@ Page %>, описанная в главе 31. Подобно типам Page мастер-страница является производной от специфичного базового класса, в данном случае MasterPage. Открыв соответствующий файл кода, вы обнаружите такое определение класса:

```
public partial class MasterPage : System.Web.UI.MasterPage
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }
}
```

Другой интересный момент в разметке мастер-страницы касается определения <asp:ContentPlaceHolder>. В эту область мастер-страницы могут подключаться виджеты пользовательского интерфейса из связанного файла содержимого *.aspx, а не содержимое, которое определено самой мастер-страницей.

Если вы намерены подключить файл *.aspx к этому разделу, то область внутри дескрипторов <asp:ContentPlaceHolder> и </asp:ContentPlaceHolder> обычно оставляется пустой. Тем не менее, данный раздел можно заполнить разнообразными элементами управления Web Forms, которые функционируют как стандартный пользовательский интерфейс для применения в случае, если заданный файл *.aspx на сайте не предоставит специфическое содержимое. В рассматриваемом примере предположим, что каждая страница *.aspx сайта действительно будет предоставлять специальное содержимое, и потому элементы <asp:ContentPlaceHolder> будут пустыми.

На заметку! Внутри страницы *.master можно определять произвольное количество заполнителей содержимого. Кроме того, страница *.master может иметь вложенные страницы *.master.

Общий пользовательский интерфейс файла *.master можно строить с помощью тех же визуальных конструкторов Visual Studio, которые используются для создания файлов *.aspx. Добавьте к сайту описательный элемент Label (служащий общим приветственным сообщением), элемент управления AdRotator (который отображает случайно выбранное одно из двух изображений) и элемент управления TreeView (позволяющий пользователю выполнять навигацию на другие области сайта). Вот как выглядит возможная разметка мастер-страницы:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title> </title>
  <asp:ContentPlaceHolder id="head" runat="server">
  </asp:ContentPlaceHolder>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <hr />
      <asp:Label ID="Label1" runat="server" Font-Size="XX-Large"
        Text="Welcome to the ASP.NET Cars Super Site!"></asp:Label>
      <asp:AdRotator ID="myAdRotator" runat="server"/>
       <br />
      <br />
      <asp:TreeView ID="navigationTree" runat="server">
      </asp:TreeView>
      <hr />
    </div>
```

```
    <div>
      <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
      </asp:ContentPlaceHolder>
    </div>
  </form>
</body>
</html>
```

На рис. 32.9 показано представление текущей мастер-страницы во время проектирования (обратите внимание, что область отображения элемента управления `AdRotator` на данный момент пуста).

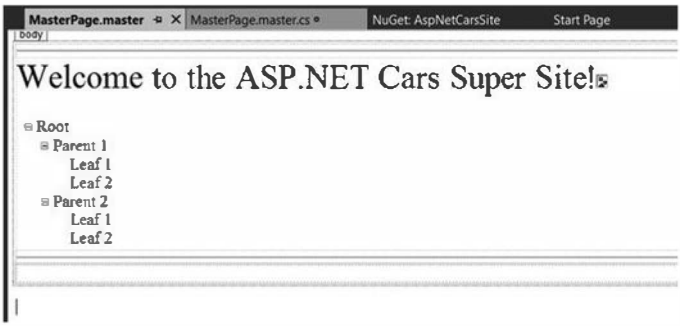


Рис. 32.9. Разделяемый пользовательский интерфейс в файле `*.master`

Можете улучшить внешний вид элемента управления `TreeView` с применением встроенного редактора элемента управления и выбора ссылки Auto Format (Автоформат). Кроме того, можете настроить отображение остальных элементов управления в окне Properties. После получения удовлетворительных результатов переходите к следующему разделу.

Конфигурирование логики навигации по сайту с помощью элемента управления `TreeView`

Инфраструктура Web Forms поставляется с несколькими веб-элементами управления, которые позволяют поддерживать навигацию по сайту: `SiteMapPath`, `TreeView` и `Menu`. Как и можно было предположить, упомянутые веб-виджеты можно конфигурировать многими способами. Например, каждый из этих элементов управления может динамически генерировать свои узлы через внешний файл XML (или файл `*.sitemap`, основанный на XML), программно в коде или посредством разметки с использованием визуальных конструкторов среды Visual Studio.

Создаваемая система навигации будет динамически наполняться с применением файла `*.sitemap`. Преимущество такого подхода в том, что можно определить общую структуру веб-сайта во внешнем файле и затем привязывать его к элементу управления `TreeView` (или `Menu`) на лету. Таким образом, если навигационная структура веб-сайта изменится, то достаточно будет просто модифицировать файл `*.sitemap` и перегрузить страницу. Вставьте в проект новый файл `Web.sitemap`, выбрав пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент), в результате чего откроется диалоговое окно, показанное на рис. 32.10.

Как видите, в начальном содержимом файла `Web.sitemap` определен элемент самого верхнего уровня с двумя подузлами:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="" title="" description="">
    <siteMapNode url="" title="" description="" />
```

```
    <siteMapNode url="" title="" description="" />
  </siteMapNode>
</siteMap>
```

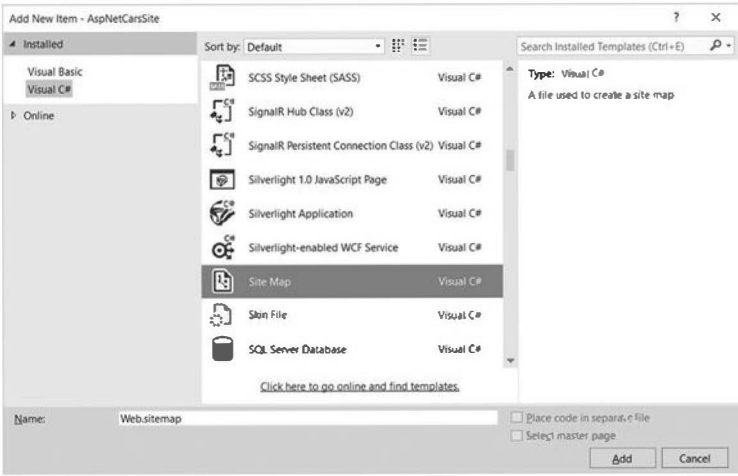


Рис. 32.10. Вставка нового файла `Web.sitemap`

После привязки такой структуры к элементу управления `Menu` должен отобразиться элемент верхнего уровня с двумя подэлементами. Следовательно, для создания подэлементов понадобится просто определить новые элементы `<siteMapNode>` внутри существующего элемента `<siteMapNode>`. В любом случае цель связана с определением общей структуры веб-сайта в файлt `Web.sitemap` с использованием разнообразных элементов `<siteMapNode>`. Каждый такой элемент может определять заголовок и атрибут URL. Атрибут URL представляет файл `*.aspx`, к которому будет выполнен переход, когда пользователь щелкнет на заданном элементе (или на узле `TreeView`). Создаваемая карта сайта будет содержать три узла (расположенные ниже узла верхнего уровня):

- Home (Домой): `Default.aspx`
- Build a Car (Собрать автомобиль): `BuildCar.aspx`
- View Inventory (Просмотреть склад): `Inventory.aspx`

Вскоре вы добавите в проект три указанных новых страницы Web Forms. А пока нужно просто сконфигурировать файл карты сайта.

Система навигации имеет единственный элемент верхнего уровня Welcome (Добро пожаловать) с тремя подэлементами. Модифицируйте файл `Web.sitemap` следующим образом (имейте в виду, что каждое значение `url` должно быть уникальным, иначе возникнет ошибка времени выполнения):

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="" title="Welcome!" description="">
    <siteMapNode url="~/Default.aspx" title="Home"
      description="The Home Page" />
    <siteMapNode url="~/BuildCar.aspx" title="Build a car"
      description="Create your dream car" />
    <siteMapNode url="~/Inventory.aspx" title="View Inventory"
      description="See what is in stock" />
    </siteMapNode>
</siteMap>
```

На заметку! Префикс ~/ перед каждой страницей в атрибуте `url` обозначает корень веб-сайта.

Вопреки тому, что вы могли подумать, файл `Web.sitemap` не ассоциируется непосредственно с элементом управления `Menu` или `TreeView` с применением какого-то свойства. Взамен файл `*.master` или `*.aspx`, содержащий виджет пользовательского интерфейса, который отобразит файл `Web.sitemap`, должен содержать компонент `SiteMapDataSource`. Этот компонент будет автоматически загружать файл `Web.sitemap` в свою объектную модель, когда страница запрашивается. Затем объекты `Menu` и `TreeView` установят свои свойства `DataSourceID` так, чтобы указывать на данный экземпляр `SiteMapDataSource`.

Чтобы добавить новый компонент `SiteMapDataSource` в файл `*.master` и автоматически установить свойство `DataSourceID`, можно задействовать визуальный конструктор Visual Studio. Активизируйте встроенный редактор элемента управления `TreeView` (щелкнув на небольшой стрелке в правом верхнем углу `TreeView`), раскройте список Choose Data Source (Выберите источник данных) и выберите пункт <New Data Source> (<Новый источник данных>), как показано на рис. 32.11.

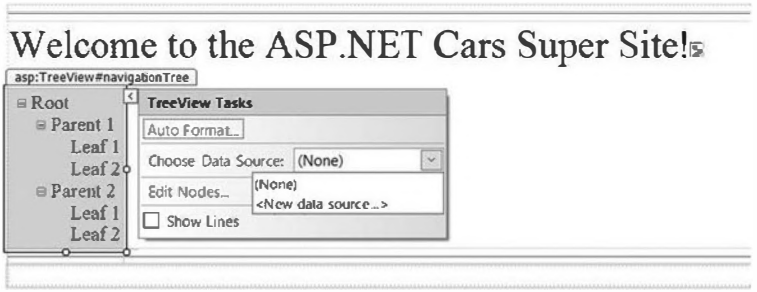


Рис. 32.11. Добавление нового компонента `SiteMapDataSource`

В открывшемся диалоговом окне выберите значок SiteMap (Карта сайта). В результате будет установлено свойство `DataSourceID` элемента управления `Menu` или `TreeView`, а к странице добавится компонент `SiteMapDataSource`. Вот и все, что потребовалось сделать, чтобы сконфигурировать элемент управления `TreeView` для перехода на дополнительные страницы внутри сайта. Если необходимо выполнить дополнительную обработку, когда пользователь выбивает пункт меню, то придется обработать событие `SelectedNodeChanged` элемента управления `TreeView`. В рассматриваемом примере в этом нет нужды, но имейте в виду, что выяснить, какой пункт меню был выбран, можно с использованием входных аргументов события.

Установка навигационных цепочек с помощью элемента `SiteMapPath`

Прежде чем переходить к элементу управления `AddRotator`, добавьте элемент `SiteMapPath` (расположенный на вкладке Navigation (Навигация) панели инструментов) в файл `*.master`. Этот виджет будет автоматически настраивать свое содержимое на основе текущего выбора в системе меню. В итоге конечный пользователь получает полезную визуальную подсказку (формально данный аспект пользовательского интерфейса называется *навигационными цепочками* или *"хлебными крошками"*). По завершении вы заметите, что при выборе пункта меню Welcome⇨Build a Car (Добро пожаловать⇨Укомплектовать автомобиль) виджет `SiteMapPath` автоматически обновится должным образом.

Конфигурирование элемента управления AdRotator

Роль виджета AdRotator из Web Forms заключается в показе случайно выбранного изображения в определенной позиции внутри браузера. В настоящий момент виджет AdRotator отображает пустой заполнитель. Этот элемент управления не может выполнять свою работу до тех пор, пока в его свойстве AdvertisementFile не будет указан исходный файл, описывающий каждое изображение. В рассматриваемом примере источником данных будет простой файл XML по имени Ads.xml.

Чтобы добавить файл XML к веб-сайту, выберите пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент) и укажите вариант XML file (XML-файл). Назовите файл Ads.xml и предусмотрите уникальный элемент <Ad> для каждого изображения, которое планируется отображать. Как минимум в элементе <Ad> должно быть указано графическое изображение для показа (ImageUrl), URL для навигации, если изображение выбрано (TargetUrl), текст, появляющийся при наведении курсора мыши (AlternateText), и вес показа (Impressions):

```
<Advertisements>
  <Ad>
    <ImageUrl>SlugBug.jpg</ImageUrl>
    <TargetUrl>http://www.Cars.com</TargetUrl>
    <AlternateText>Your new Car?</AlternateText>
    <Impressions>80</Impressions>
  </Ad>
  <Ad>
    <ImageUrl>car.gif</ImageUrl>
    <TargetUrl>http://www.CarSuperSite.com</TargetUrl>
    <AlternateText>Like this Car?</AlternateText>
    <Impressions>80</Impressions>
  </Ad>
</Advertisements>
```

Здесь указаны два файла с изображениями (SlugBug.jpg и car.gif). В результате понадобится обеспечить наличие этих файлов в корневом каталоге веб-сайта (они включены в состав загружаемого кода примеров для книги). Чтобы добавить их в текущий проект, выберите пункт меню Web Site⇨Add Existing Item (Веб-сайт⇨Добавить существующий элемент). Затем файл XML можно ассоциировать с элементом управления AdRotator посредством свойства AdvertisementFile (в окне Properties):

```
<asp:AdRotator ID="myAdRotator" runat="server" AdvertisementFile="~/Ads.xml"/>
```

Когда вы позже запустите приложение и выполните обратную отправку страницы, то увидите случайно выбранный из двух файлов изображения.

Определение стандартной страницы содержимого

Располагая готовой мастер-страницей, можно приступать к проектированию индивидуальных страниц *.aspx, которые будут определять содержимое пользовательского интерфейса для помещения внутрь дескриптора <asp:ContentPlaceHolder> мастер-страницы. Файлы *.aspx, которые объединяются с мастер-страницей, называются *страницами содержимого* и имеют несколько ключевых отличий от нормальной автономной страницы Web Forms. Выражаясь кратко, в файле *.master определяется раздел <form> финальной страницы HTML. Следовательно, существующая область <form> внутри файла *.aspx должна быть заменена областью <asp:Content>. В то время как можно было бы модифицировать разметку в начальном файле *.aspx вручную, лучше вставить в проект новую страницу содержимого. Удалите имеющийся файл Default.aspx, щелкните правой кнопкой мыши в любом месте на поверхности визуального

конструктора файла `*.master` и выберите в контекстном меню пункт Add Content Page (Добавить страницу содержимого), как показано на рис. 32.12.

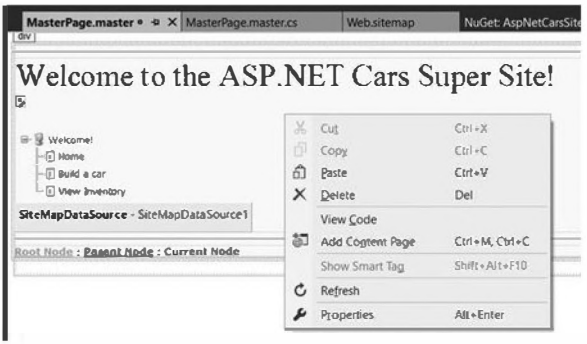


Рис. 32.12. Добавление к мастер-странице новой страницы содержимого

В результате будет сгенерирован новый файл `*.aspx` с приведенной ниже начальной разметкой:

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master"
AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" Title="" %>
<asp:Content ID="Content1"
  ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2"
  ContentPlaceHolderID="ContentPlaceHolder1" Runat="Server">
</asp:Content>
```

Прежде всего, обратите внимание на то, что директива `<%@ Page %>` дополнена новым атрибутом `MasterPageFile`, который указывает на файл `*.master`. Кроме того, вместо элемента `<form>` имеется область `<asp:Content>` (пока пустая), в которой для атрибута `ContentPlaceHolderID` установлено значение, идентичное компоненту `<asp:ContentPlaceHolder>` в файле мастер-страницы.

С учетом этих сопоставлений странице содержимого известно, куда подключается ее содержимое, тогда как содержимое мастер-страницы отображается в стиле только для чтения на странице содержимого. Строить сложный пользовательский интерфейс для области содержимого `Default.aspx` нет нужды. В данном примере просто добавьте некоторый литеральный текст с базовыми инструкциями относительно сайта, как показано на рис. 32.13 (в правом верхнем углу страницы содержимого в визуальном конструкторе имеется ссылка для переключения на связанную мастер-страницу).

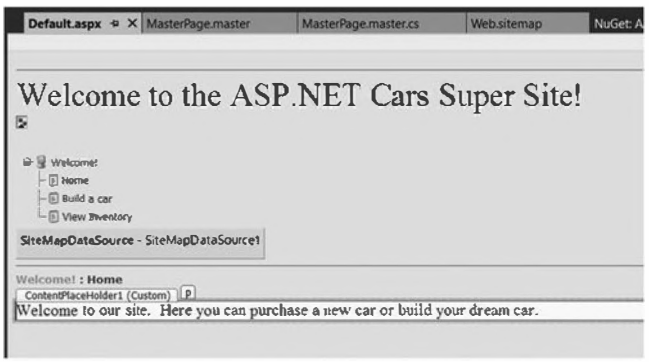


Рис. 32.13. Создание первой страницы содержимого

Теперь после запуска проекта вы обнаружите, что содержимое пользовательского интерфейса из файлов `*.master` и `Default.aspx` объединено в единый поток разметки HTML. Как видно на рис. 32.14, браузер (или конечный пользователь) даже не осведомлен о существовании мастер-страницы (обратите внимание, что браузер просто отображает разметку HTML из `Default.aspx`). Вдобавок, если обновить страницу (нажатием <F5>), то элемент управления `AdRotator` отобразит случайно выбранное одно из двух изображений.

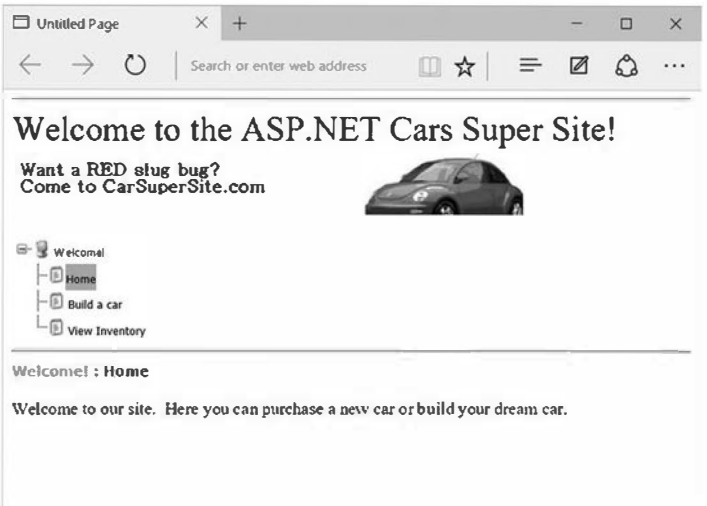


Рис. 32.14. Во время выполнения мастер-страницы и страницы содержимого визуализируются в единую форму

Проектирование страницы содержимого `Inventory.aspx`

Чтобы вставить в текущий проект страницу содержимого `Inventory.aspx`, откройте файл `*.master` в IDE-среде, выберите пункт меню Website⇨Add Content Page (Веб-сайт⇨Добавить страницу содержимого) и в окне Solution Explorer переименуйте этот файл в `Inventory.aspx`. Страница содержимого `Inventory.aspx` будет отображать записи таблицы `Inventory` из базы данных `AutoLot` внутри элемента управления `GridView`. Однако в отличие от предыдущей главы этот элемент управления `GridView` будет сконфигурирован для взаимодействия с базой данных `AutoLot` с применением встроенной поддержки привязки данных и обновленной сборки `AutoLotDAL` из главы 30.

Хотя элемент управления `GridView` из Web Forms обладает способностью представлять данные строки соединения и SQL-операторы `Select`, `Insert`, `Update` и `Delete` (или в качестве альтернативы хранимые процедуры) *в разметке*, предпочтительнее использовать уровень доступа к данным (Data Access Layer — DAL). Такой подход способствует разделению ответственности, а также изолирует хранилище данных от кода пользовательского интерфейса.

С помощью нескольких простых атрибутов и минимального кода (с учетом наличия DAL) элемент управления `GridView` можно сконфигурировать для автоматической выборки, обновления и удаления записей лежащего в основе хранилища данных. Это значительно сокращает объем стереотипного кода и включается через свойства `SelectMethod`, `DeleteMethod` и `UpdateMethod` (а также дополнительные средства для списковых элементов управления), введенные в ASP.NET 4.5.

Добавление сборки AutoLotDAL и инфраструктуры Entity Framework в проект AspNetCarsSite

Скопируйте сборку AutoLotDAL из подкаталога Chapter_31 (либо из подкаталога Chapter_32 в загружаемом коде примеров). Добавьте ссылку на сборку AutoLotDAL, для чего щелкните правой кнопкой мыши на узле References (Ссылки) проекта AspNetCarsSite, выберите в контекстном меню пункт Add Reference (Добавить ссылку), в открывшемся диалоговом окне Add Reference (Добавление ссылки) щелкните на кнопке Browse (Обзор) и укажите файл AutoLotDAL.dll.

Затем добавьте в веб-проект инфраструктуру Entity Framework, щелкнув правой кнопкой мыши на имени проекта и выбрав в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet). Также понадобится добавить в файл Web.config строку подключения. Изменение будет выглядеть следующим образом (строка подключения зависит от установленной версии SQL Server Express):

```
<connectionStrings>
  <add name="AutoLotConnection"
    connectionString="data source=.\SQLEXPRESS2014;initial
catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;
App=EntityFramework"
    providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Наполнение данными элемента управления GridView

Для демонстрации работы с элементом управления GridView в декларативной манере модифицируйте страницу содержимого Inventory.aspx с определением GridView. В табл. 32.3 кратко описаны атрибуты, которые будут добавлены в последующих разделах.

Таблица 32.3. Избранные атрибуты списковых элементов управления

Атрибут	Описание
DataKeyNames	Указывает первичный ключ таблицы
ItemType	Включает строгую типизацию для списковых элементов управления
SelectMethod	Указывает метод в отделенном коде, который будет применяться для наполнения спискового элемента управления. Он вызывается при каждой визуализации элемента управления
DeleteMethod	Указывает метод, который будет использоваться для удаления записи из источника данных для таблицы
UpdateMethod	Указывает метод, который будет применяться для обновления записи, когда отредактированные данные отправляются серверу

Начните с использования атрибутов ItemType и SelectMethod (как делалось в главе 31). Модифицируйте объявление GridView (во втором элементе <asp:Content>), как показано ниже:

```
<asp:GridView ID="GridView2" runat="server" CellPadding="4"
  AutoGenerateColumns="False"
  ItemType="AutoLotDAL.Models.Inventory" SelectMethod="GetData"
  EmptyDataText="There are no data records to display." ForeColor="#333333"
  GridLines="None">
```

Далее добавьте следующие записи в разделе <Columns> (пока что не обращайте внимания на атрибуты SortExpression):

```
<Columns>
  <asp:BoundField DataField="CarID" HeaderText="CarID" ReadOnly="True"
       SortExpression="CarID" />
  <asp:BoundField DataField="Make" HeaderText="Make" SortExpression="Make" />
  <asp:BoundField DataField="Color" HeaderText="Color" SortExpression="Color" />
  <asp:BoundField DataField="PetName" HeaderText="PetName"
       SortExpression="PetName" />
</Columns>
```

Удостоверьтесь в наличии закрывающего дескриптора GridView после закрывающего дескриптора Columns:

```
</asp:GridView>
```

Рис. 32.15. Отображение данных из таблицы Inventory

Откройте файл Inventory.aspx.cs и добавьте метод GetData(), который не принимает параметры и возвращает IEnumerable<Inventory>. Из-за того, что сборка DAL уже создана, код будет тривиален. Добавьте операторы using для пространств имен AutoLotDAL.Models и AutoLotDAL.Repos, после чего реализуйте метод GetData():

```
public IEnumerable<Inventory>
GetData() => new InventoryRepo().GetAll();
```

Теперь приложение можно запустить. Щелкните на пункте меню View Inventory (Просмотреть склад) и просмотрите данные (рис. 32.15). (Внешний вид элемента управления GridView был изменен с помощью встроенного редактора.)

Включение редактирования на месте

Следующей задачей будет включение для элемента управления GridView поддержки действий на месте. Вы начнете применять атрибуты DataKeyNames, DeleteMethod и UpdateMethod. Измените объявление GridView, как показано ниже:

```
<asp:GridView ID="GridView2" runat="server" CellPadding="4"
  AutoGenerateColumns="False"
  DataKeyNames="CarID, Timestamp" ItemType="AutoLotDAL.Models.Inventory"
  SelectMethod="GetData" DeleteMethod="Delete" UpdateMethod="Update"
  EmptyDataText="There are no data records to display." ForeColor="#333333"
  GridLines="None">
```

Поле CarId является первичным ключом, так что ему имеет смысл находиться в атрибуте DataKeyNames. Поле типа Timestamp добавляется в DataKeyNames как ключ данных, поэтому оно будет передаваться методам Update() и Delete(). Добавьте в раздел <Columns> следующую запись CommandField. В итоге к каждой строке будут добавлены ссылки Edit (Редактировать) и Delete (Удалить). Вот как выглядит модифицированная разметка:

```
<Columns>
  <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
  <asp:BoundField DataField="CarID" HeaderText="CarID" ReadOnly="True"
      SortExpression="CarID" />
  <asp:BoundField DataField="Make" HeaderText="Make" SortExpression="Make" />
  <asp:BoundField DataField="Color" HeaderText="Color" SortExpression="Color" />
  <asp:BoundField DataField="PetName" HeaderText="PetName"
      SortExpression="PetName" />
</Columns>
```

Откройте файл `Inventory.aspx.cs` и добавьте в него методы `Delete()` и `Update()`. Метод `Delete()` возвращает `void`, а принимает в качестве параметров `carId` типа `int` и `timeStamp` типа `byte[]`. Оба значения передаются методу, поскольку они указаны внутри атрибута `DataKeyNames` в разметке.

```
public void Delete(int carId, byte[] timeStamp)
{
  new InventoryRepo().Delete(carId, timeStamp);
}
```

Метод `Update()` возвращает `void` и использует привязку модели, поэтому он может принимать параметр типа `Inventory`. Привязка модели представляет собой средство ASP.NET MVC, которое было перенесено в инфраструктуру ASP.NET Web Forms 4.5. Оно берет все пары "имя-значение" формы, строки запроса и других источников и пробует воссоздать объект указанного типа с применением рефлексии. Различают *явную привязку модели* и *неявную привязку модели*. В каждом случае механизм привязки модели пытается присвоить значения из пар "имя-значение" (отправленной формы) соответствующим свойствам объекта желаемого типа. Если присвоить значение одному или нескольким свойствам не удалось (из-за проблем с преобразованием типов данных или ошибок проверки достоверности), то механизм привязки модели устанавливает свойство `ModelState.IsValid` в `false`. Если же всем соответствующим свойствам были успешно присвоены значения, то `ModelState.IsValid` устанавливается в `true`.

Для явной привязки модели необходимо вызвать метод `TryUpdateModel()`, передав ему экземпляр интересующего типа. В случае отказа привязки модели метод `TryUpdateModel()` возвращает `false`. Например, метод `Update()` можно было бы реализовать так:

```
public async void Update(int carID)
{
  var inv = new Inventory() {CarID = carID};
  if (TryUpdateModel(inv))
  {
    await new InventoryRepo().SaveAsync(inventory);
  }
}
```

Для неявной привязки модели нужно передать методу объект желаемого типа в качестве параметра. Вам понадобится сделать это с методом `Update()`. В теле метода сначала проверьте состояние модели на предмет допустимости (более подробно о проверке достоверности речь пойдет позже в главе), а затем вызовите метод `SaveAsync()` на экземпляре `InventoryRepo`. Поскольку при вызове `SaveAsync()` используется `await`, добавьте модификатор `async` к методу `Update()`:

```
public async void Update(Inventory inventory)
{
  if (ModelState.IsValid)
  {
    await new InventoryRepo().SaveAsync(inventory);
  }
}
```

После добавления модификатора `async` к методу `Update()` возникнет ошибка, когда метод `Update()` будет вызван, потому что объект `Page` не помечен как `async`. К счастью, это сводится просто к добавлению атрибута `Async="true"` к директиве `Page`:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/MasterPage.master"
  AutoEventWireup="true" CodeFile="Inventory.aspx.cs" Inherits="InventoryPage"
  Async="true" %>
```

Запустив приложение, вы увидите ссылки Edit (Редактировать) и Delete (Удалить). Щелчок на ссылке Edit включает редактирование на месте (рис. 32.16). В режиме редактирования ссылки изменяются на Update (Обновить) и Cancel (Отменить).

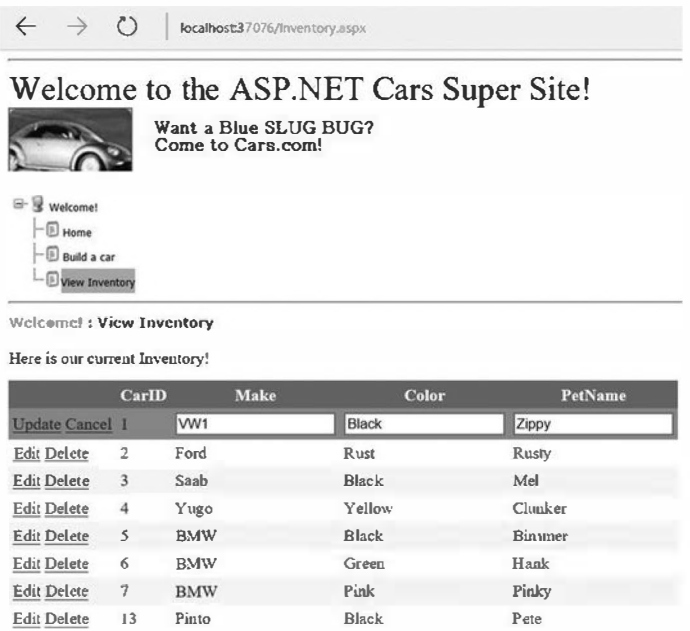


Рис. 32.16. Редактирование на месте и удаление

Включение сортировки и разбиения на страницы

Элемент управления `GridView` можно легко сконфигурировать для выполнения сортировки (через гиперссылки с именами столбцов) и разбиения на страницы (посредством числовых гиперссылок или гиперссылок "следующая/предыдущая"). Чтобы сделать это, модифицируйте разметку для элемента управления `GridView`, добавив атрибуты `AllowPaging`, `PageSize` и `AllowSorting`:

```
<asp:GridView ID="carsGrid" runat="server"
  AllowPaging="True" PageSize="2"
  AllowSorting="True" AutoGenerateColumns="False" CellPadding="4"
  DataKeyNames="CarID" ItemType="AutoLotDAL.Models.Inventory"
  SelectMethod="GetData" DeleteMethod="Delete" UpdateMethod="Update"
  EmptyDataText="There are no data records to display." ForeColor="#333333"
  GridLines="None">
```

Если вы запустите приложение прямо сейчас, то получите следующее сообщение об ошибке:

```
When the DataBoundControl has paging enabled, either the SelectMethod
should return an IQueryable<ItemType> or should have all these mandatory
parameters: int startRowIndex, int maximumRows, out int totalRowCount
```

Когда в элементе DataBoundControl включено разбиение на страницы, метод, указанный в SelectMethod, либо должен возвращать IQueryable<ItemType>, либо должен иметь все обязательные параметры: int startRowIndex, int maximumRows, out int totalRowCount

Ошибка легко устраняется путем добавления вызова метода `AsQueryable()` **после вызова** `GetAll()` **на экземпляре** `InventoryRepo` **и изменения сигнатуры метода для возвращения** `IQueryable<Inventory>`:

```
public IQueryable<Inventory> GetData() =>
  new InventoryRepo().GetAll().AsQueryable();
```

На заметку! Несмотря на то что добавление вызова `AsQueryable()` решает проблему в данном примере, было бы лучше открыть доступ к версии `IQueryable` метода `GetAll()` в самом хранилище.

Снова запустив приложение, вы получите возможность сортировки данных за счет щелчка на именах столбцов и прокрутки данных с помощью страничных ссылок (при условии, что в таблице `Inventory` **присутствует достаточное количество записей), как показано на рис. 32.17.**

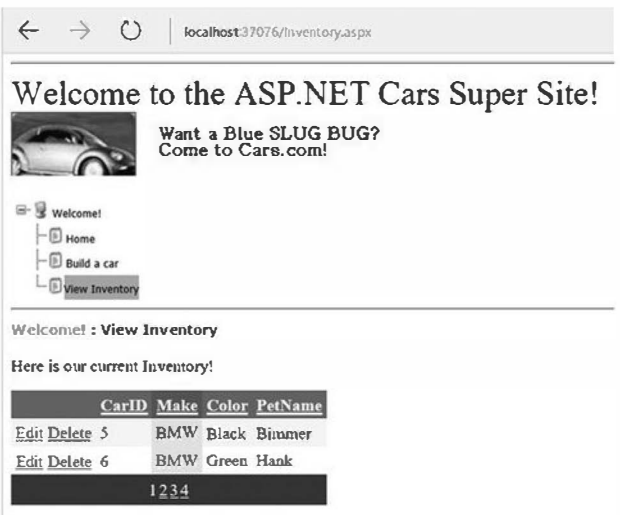


Рис. 32.17. Включение сортировки и разбиения на страницы

Включение фильтрации

Следующая задача заключается в добавлении к элементу управления `GridView` **возможности фильтрации. Опять-таки, благодаря функциональным возможностям, добавленным в ASP.NET Web Forms 4.5, все довольно просто. Начните с добавления элемента управления** `DropDownList`**, который будет привязан к отдельному списку значений** `Make` **в базе данных** `AutoLot`**. Атрибуты** `DataTextField` **(то, что отображается) и** `DataValueField` **(значение в раскрывающемся списке, основанное на выбранном элементе) установлены в** `Make`**. В атрибуте** `SelectMethod` **должен быть указан метод по имени** `GetMakes()`**. Основной момент в том, что в элементе управления** *должна* **при-**

сутствовать настройка runat="server". Кроме того, обратите внимание на дескриптор <asp:ListItem>. Он добавляет вариант выбора (All) (Все), если в списке ничего не выбрано. Ниже приведена разметка:

```
<asp:DropDownList ID="cboMake" SelectMethod="GetMakes"
  AppendDataBoundItems="true" AutoPostBack="true"
  DataTextField="Make" DataValueField="Make" runat="server">
  <asp:ListItem Value="" Text="(All)" />
</asp:DropDownList>
```

Теперь откройте файл Inventory.aspx.cs и создайте метод GetMakes(). В этом методе должен возвращаться список новых анонимных объектов, которые содержат разные значения Make из базы данных. Код метода выглядит так:

```
public IEnumerable GetMakes() =>
  new InventoryRepo().GetAll().Select(x => new {x.Make}).Distinct();
```

Метод GetData() также должен быть обновлен для фильтрации данных в случае передачи значения Make. Параметр помечается атрибутом [Control("cboMake")], в котором указано имя элемента управления. Указывать имя элемента управления необязательно, если оно совпадает с именем параметра, но из-за того, что в этом примере они отличаются, имя элемента управления необходимо предоставить. Параметр будет получать значение из элемента управления при обратной отправке формы (отсюда и требование наличия runat="server") и принимать значение пустой строки, если ничего не выбрано. Вот как это делается:

```
public IQueryable<Inventory> GetData([Control("cboMake")]string make="")
{
  return string.IsNullOrEmpty(make) ?
    new InventoryRepo().GetAll().AsQueryable() :
    new InventoryRepo().GetAll().Where(x => x.Make == make).AsQueryable();
}
```

Запустив приложение, теперь можно выбирать марку автомобиля и фильтровать результирующий набор на основе выбранного значения (рис. 32.18).

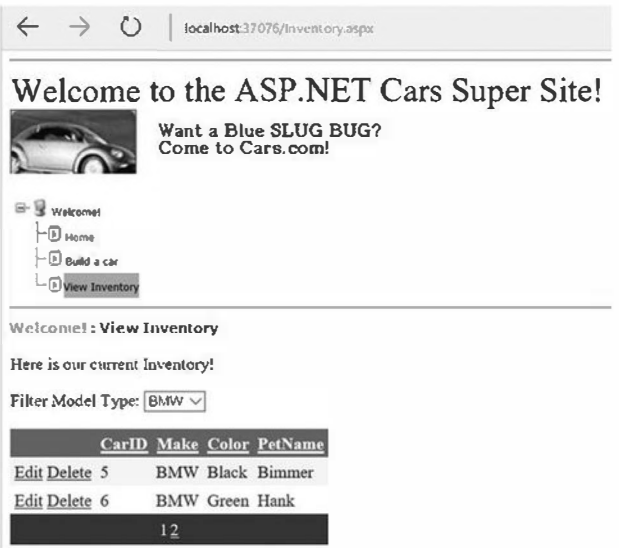


Рис. 32.18. Фильтрация данных на основе значения, выбранного в элементе управления

Проектирование страницы содержимого `BuildCar.aspx`

В рассматриваемом примере осталось еще спроектировать страницу содержимого `BuildCar.aspx`. Удостоверьтесь, что файл `*.master` открыт для редактирования, и вставьте новую страницу содержимого в текущий проект (с помощью пункта меню Website⇨Add Content Page; это альтернатива щелчку правой кнопкой мыши на мастер-странице проекта и выбору подходящего пункта в контекстном меню). В окне Solution Explorer переименуйте новый файл в `BuildCar.aspx`.

На новой странице будет применяться элемент управления `Wizard` из Web Forms, который предоставляет простой способ для проведения конечного пользователя через последовательность связанных шагов. Здесь такие шаги будут эмулировать действие по комплектации автомобиля для покупки.

Поместите в область содержимого элементы управления `Label` и `Wizard`. Затем активизируйте встроенный редактор для `Wizard` и щелкните на ссылке Add/Remove WizardSteps (Добавить/удалить шаги мастера). Добавьте четыре шага, как показано на рис. 32.19.

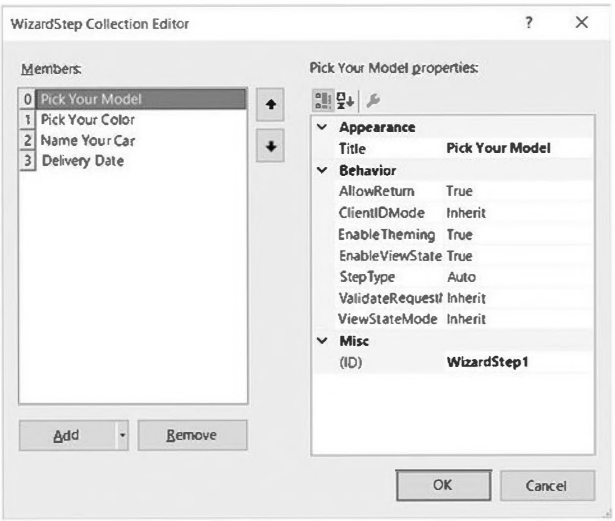


Рис. 32.19. Конфигурирование элемента управления `Wizard`

После определения шагов вы заметите, что `Wizard` предлагает пустую область содержимого, куда можно перетаскивать элементы управления для шага мастера, выбранного в текущий момент. В рассматриваемом примере модифицируйте каждый шаг, добавив следующие элементы пользовательского интерфейса (не забудьте назначить каждому элементу подходящие идентификаторы в окне Properties):

- Pick Your Model (Выбор модели): элемент управления `TextBox`;
- Pick Your Color (Выбор цвета): элемент управления `ListBox`;
- Name Your Car (Название автомобиля): элемент управления `TextBox`;
- Delivery Date (Дата доставки): элемент управления `Calendar`.

Элемент управления `ListBox` — единственный элемент пользовательского интерфейса внутри `Wizard`, который требует выполнения дополнительных шагов. Выберите его в визуальном конструкторе (не забыв сначала выбрать ссылку Pick Your Color) и заполните виджет набором цветов, используя свойство `Items` в окне Properties. После этого вы увидите в области определения `Wizard` разметку, похожую на следующую:

```
<asp:ListBox ID="ListBoxColors" runat="server" Width="237px">
  <asp:ListItem>Purple</asp:ListItem>
  <asp:ListItem>Green</asp:ListItem>
  <asp:ListItem>Red</asp:ListItem>
  <asp:ListItem>Yellow</asp:ListItem>
  <asp:ListItem>Pea Soup Green</asp:ListItem>
  <asp:ListItem>Black</asp:ListItem>
  <asp:ListItem>Lime Green</asp:ListItem>
</asp:ListBox>
```

После определения всех шагов можно обработать событие `FinishButtonClick` для автоматически сгенерированной кнопки Finish (Готово). Тем не менее, имейте в виду, что кнопка Finish не видна до тех пор, пока не будет выбран финальный шаг мастера. Выбрав последний шаг, просто дважды щелкните на кнопке Finish, чтобы создать обработчик события. Внутри обработчика события серверной стороны извлеките выбор из каждого элемента пользовательского интерфейса и постройте строку описания, которую понадобится присвоить свойству `Text` дополнительного элемента `Label` по имени `lblOrder`:

```
public partial class BuildCarPage : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }
  protected void carWizard_FinishButtonClick(object sender,
    WizardNavigationEventArgs e)
  {
    // Получить каждое значение.
    string order = $"{txtCarPetName.Text},
       your { ListBoxColors.SelectedValue } { txtCarModel.Text }
       will arrive on { carCalendar.SelectedDate.ToShortDateString()}";
    // Присвоить результирующую строку метке.
    lblOrder.Text = order;
  }
}
```

Веб-приложение `AspNetCarsSite` готово. На рис. 32.20 показан элемент `Wizard` в действии.

На этом краткий обзор разнообразных элементов управления Web Forms, мастер-страниц, страниц содержимого и навигации с помощью карты сайта завершен. Далее мы займемся исследованием функциональности элементов управления проверкой достоверности Web Forms. Чтобы изолировать тематику настоящей главы, для демонстрации приемов проверки достоверности мы построим новый веб-сайт. Однако элементы управления проверкой достоверности можно добавить и в текущий проект.

Исходный код. Веб-сайт `AspNetCarsSite` доступен в подкаталоге `Chapter_32`.

Роль элементов управления проверкой достоверности

Следующий набор элементов управления Web Forms, который мы рассмотрим, называется *элементами управления проверкой достоверности*. В отличие от других элементов управления Web Forms они не выпускают разметку HTML в целях визуализации, а применяются для выпуска кода JavaScript клиентской стороны, проверяющего достоверность данных формы.

Рис. 32.20. Элемент управления `Wizard` в действии

Как было показано в начале главы, проверка достоверности данных формы клиентской стороны удобна тем, что позволяет на месте проверять данные на предмет соответствия различным ограничениям, прежде чем отправлять их обратно веб-серверу, в итоге сокращая количество затратных процедур обмена с сервером. В табл. 32.4 приведена краткая сводка по элементам управления проверкой достоверности Web Forms.

Таблица 32.4. Элементы управления проверкой достоверности Web Forms

Элемент управления	Описание
`CompareValidator`	Проверяет значение в элементе ввода на равенство заданному значению другого элемента ввода или фиксированной константе
`CustomValidator`	Позволяет строить специальную функцию проверки достоверности, которая проверяет заданный элемент управления
`RangeValidator`	Определяет, находится ли данное значение внутри заранее определенного диапазона
`RegularExpressionValidator`	Проверяет значение в ассоциированном элементе ввода на соответствие шаблону регулярного выражения
`RequiredFieldValidator`	Проверяет заданный элемент ввода на наличие значения (т.е. что он не пуст)
`ValidationSummary`	Отображает итог по всем ошибкам проверки достоверности страницы в формате простого списка, маркированного списка или одиночного абзаца. Ошибки могут отображаться встроенным способом и/или во всплывающем окне сообщения

Все элементы управления проверкой достоверности (кроме `ValidationSummary`) в конечном итоге являются производными от общего базового класса по имени `System.Web.UI.WebControls.BaseValidator` и потому обладают набором общих функциональных возможностей. Основные члены перечислены в табл. 32.5.

Таблица 32.5. Общие члены элементов управления проверкой достоверности Web Forms

Член	Описание
`ControlToValidate`	Получает или устанавливает элемент управления, подлежащий проверке достоверности
`Display`	Получает или устанавливает поведение сообщений об ошибках в элементе управления проверкой достоверности
`EnableClientScript`	Получает или устанавливает значение, которое указывает, включена ли проверка достоверности клиентской стороны
`ErrorMessage`	Получает или устанавливает текст для сообщения об ошибке
`ForeColor`	Получает или устанавливает цвет сообщения, отображаемого при неудачной проверке достоверности

Чтобы проиллюстрировать работу с элементами управления проверкой достоверности, создайте новый проект пустого веб-сайта по имени `ValidatorCtrls` и вставьте в него новую веб-форму по имени `Default.aspx`. Для начала поместите на страницу четыре (именованных) элемента управления `TextBox` (с четырьмя соответствующими описательными элементами `Label`). Затем поместите на страницу рядом с каждым полем элементы управления `RequiredFieldValidator`, `RangeValidator`, `RegularExpressionValidator` и `CompareValidator`. Наконец, добавьте элементы `Button` и `Label`. На рис. 32.21 показан возможный вид компоновки.

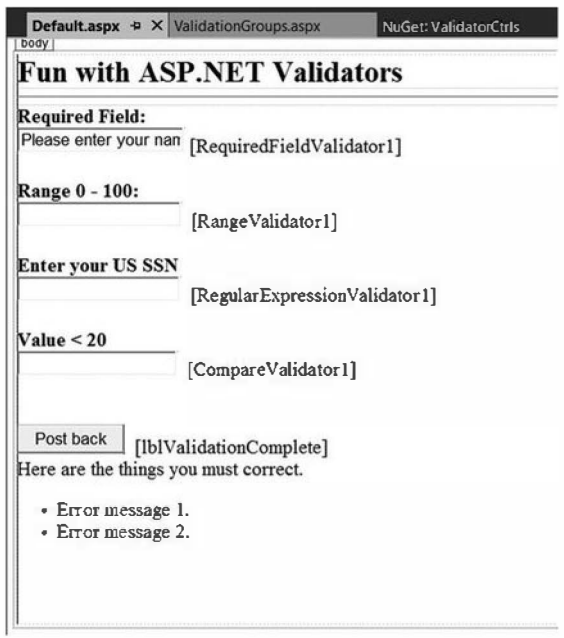


Рис. 32.21. Элементы управления проверкой достоверности Web Forms обеспечат корректность данных формы перед ее обратной отправкой

Располагая начальным пользовательским интерфейсом для проведения экспериментов, давайте пройдемся по процессу конфигурирования каждого элемента управления проверкой достоверности и посмотрим на конечный результат всех действий. Но сначала понадобится модифицировать текущий файл `Web.config`, разрешив обработку клиентской стороны для элементов управления проверкой достоверности.

Включение поддержки проверки достоверности с помощью кода JavaScript клиентской стороны

Начиная с версии ASP.NET 4.5, в Microsoft ввели новую настройку для управления способом реагирования элементов управления проверкой достоверности во время выполнения. Открыв файл `Web.config` после создания веб-приложения ASP.NET, вы обнаружите в нем следующую настройку:

```
<appSettings>
  <add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms" />
</appSettings>
```

Когда эта настройка присутствует в файле `Web.config`, веб-сайт будет проводить проверку достоверности с использованием разнообразных атрибутов данных HTML 5.0, а не отправлять обратно порции кода JavaScript клиентской стороны для обработки веб-браузером. Поскольку HTML 5.0 в книге подробно не рассматривается, после создания проекта приложения Web Forms (вместо веб-сайта) эту строку необходимо закомментировать (или удалить), чтобы текущий пример проверки достоверности работал корректно.

Элемент управления `RequiredFieldValidator`

Конфигурировать элемент управления `RequiredFieldValidator` легко. Просто установите свойства `ErrorMessage` и `ControlToValidate` соответствующим образом в окне Properties среды Visual Studio. Ниже показана результирующая разметка, которая обеспечивает то, что текстовое поле `txtRequiredField` не будет пустым:

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1"
  runat="server" ControlToValidate="txtRequiredField"
  ErrorMessage="Oops! Need to enter data.">
</asp:RequiredFieldValidator>
```

Класс `RequiredFieldValidator` поддерживает свойство `InitialValue`. Его можно применять для проверки того факта, что пользователь ввел в связанном элементе `TextBox` какое-то значение, отличающееся от начального. Например, когда пользователь впервые получает страницу, элемент управления `TextBox` можно сконфигурировать так, чтобы он содержал значение "Please enter your name". Если свойство `InitialValue` элемента `RequiredFieldValidator` не установлено, то исполняющая среда будет предполагать, что строка "Please enter your name" является допустимым значением. Таким образом, чтобы обязательное поле `TextBox` было допустимым только в случае, когда в нем введено значение, отличающееся от строки "Please enter your name", сконфигурируйте виджет, как показано ниже:

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1"
  runat="server" ControlToValidate="txtRequiredField"
  ErrorMessage="Oops! Need to enter data."
  InitialValue="Please enter your name">
</asp:RequiredFieldValidator>
```

Элемент управления RegularExpressionValidator

Элемент управления RegularExpressionValidator можно использовать, когда к символам, введенным в заданном поле, необходимо применить шаблон. Например, чтобы обеспечить наличие в элементе TextBox корректного номера карточки социального страхования США, виджет можно было бы определить следующим образом:

```
<asp:RegularExpressionValidator ID="RegularExpressionValidator1"
  runat="server" ControlToValidate="txtRegExp"
  ErrorMessage="Please enter a valid US SSN."
  ValidationExpression="\d{3}-\d{2}-\d{4}">
</asp:RegularExpressionValidator>
```

Обратите внимание на то, как в RegularExpressionValidator определено свойство ValidationExpression. Если вы ранее не работали с регулярными выражениями, то в рассматриваемом примере достаточно знать лишь то, что они используются для проверки соответствия строки определенному шаблону. Здесь выражение "\d{3}-\d{2}-\d{4}" получает стандартный номер карточки социального страхования США в форме *xxx-xx-xxxx* (где *x* — десятичная цифра).

Это конкретное регулярное выражение вполне очевидно; тем не менее, предположим, что требуется проверить правильность телефонного номера в Японии. Нужное выражение становится намного сложнее: "(0\d{1,4}-|\(0\d{1,4}\)?)?\d{1,4}-\d{4}". Хорошая новость в том, что при выборе свойства ValidationExpression в окне Properties по щелчку на кнопке с троеточием доступен заранее определенный список распространенных регулярных выражений.

На заметку! За программное манипулирование регулярными выражениями в .NET отвечают два пространства имен — System.Text.RegularExpressions и System.Web.RegularExpressions.

Элемент управления RangeValidator

В дополнение к свойствам MinimumValue и MaximumValue класс RangeValidator имеет свойство по имени Type. С учетом того, что вы заинтересованы в проверке пользовательского ввода на вхождение в диапазон целых чисел, понадобится указать тип Integer (который *не* является стандартной установкой):

```
<asp:RangeValidator ID="RangeValidator1"
  runat="server" ControlToValidate="txtRange"
  ErrorMessage="Please enter value between 0 and 100."
  MaximumValue="100" MinimumValue="0" Type="Integer">
</asp:RangeValidator>
```

Класс RangeValidator также может применяться для проверки вхождения в диапазон денежных значений, дат, чисел с плавающей точкой и строковых данных (стандартная установка).

Элемент управления CompareValidator

Наконец, обратите внимание, что CompareValidator поддерживает следующее свойство Operator:

```
<asp:CompareValidator ID="CompareValidator1" runat="server"
  ControlToValidate="txtComparison"
  ErrorMessage="Enter a value less than 20." Operator="LessThan"
  ValueToCompare="20" Type="Integer">
</asp:CompareValidator>
```

Поскольку предназначение этого элемента управления проверкой достоверности заключается в сравнении значения в текстовом поле с другим значением, используя бинарную операцию, не должен вызывать удивление тот факт, что свойство `Operator` может принимать такие значения, как `LessThan`, `GreaterThan`, `Equal` и `NotEqual`. Свойство `ValueToCompare` применяется для установки значения, с которым производится сравнение. Обратите внимание, что в `Type` указано `Integer`. По умолчанию `CompareValidator` будет выполнять сравнение со строковыми значениями.

На заметку! Элемент управления `CompareValidator` также может быть сконфигурирован для сравнения со значением внутри другого элемента управления Web Forms (вместо жестко закодированного значения) с использованием свойства `ControlToCompare`.

Чтобы завершить код этой страницы, обработайте событие `Click` элемента управления `Button` и проинформируйте пользователя об успешном прохождении логики проверки достоверности:

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {

  }
  protected void btnPostback_Click(object sender, EventArgs e)
  {
    lblValidationComplete.Text = "You passed validation!";
  }
}
```

Загрузите готовую страницу в браузер. Пока что вы не должны видеть какие-либо заметные изменения. Однако при попытке щелкнуть на кнопке Submit (Отправить) после ввода некорректных данных появится сообщение об ошибке. После ввода правильных данных сообщение об ошибке исчезнет и произойдет обратная отправка. Просмотрев разметку HTML, визуализированную браузером, вы обнаружите, что элементы управления проверкой достоверности генерируют функцию JavaScript клиентской стороны, которая задействует специфическую библиотеку функций JavaScript, автоматически загружаемую на пользовательскую машину. Как только проверка достоверности успешно прошла, данные формы отправляются обратно серверу, где исполняющая среда проведет ту же самую проверку еще раз на веб-сервере (просто чтобы удостовериться в том, что данные не были искажены по пути).

Кстати, если HTTP-запрос был отправлен браузером, который не поддерживает JavaScript клиентской стороны, то вся проверка достоверности выполняется на сервере. Таким образом, программировать элементы управления проверкой достоверности можно, не задумываясь о целевом браузере; возвращенная страница HTML переадресует обработку ошибок веб-серверу.

Создание итоговой панели проверки достоверности

Следующая тема, касающаяся проверки достоверности, которую мы рассмотрим — применение виджета `ValidationSummary`. В настоящий момент каждый элемент управления проверкой достоверности отображает свое сообщение об ошибке в месте, куда он был помещен во время проектирования. Во многих случаях именно это и требуется. Однако в сложных формах с многочисленными виджетами ввода вариант с засорением формы многочисленными надписями красного цвета может не устроить. С использованием элемента управления `ValidationSummary` можно заставить все типы проверки

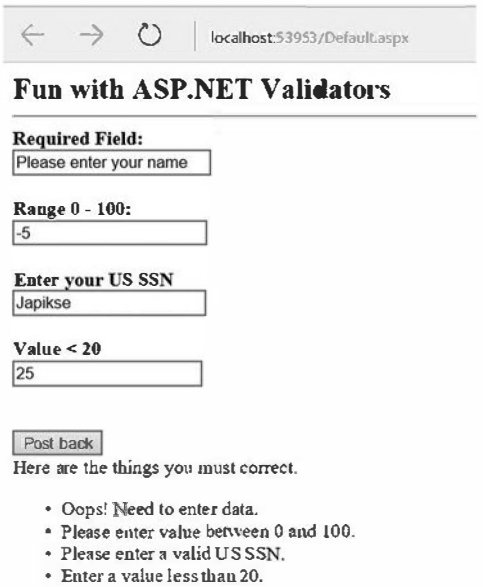


Рис. 32.22. Применение итоговой панели проверки достоверности

достоверности отображать свои сообщения об ошибках в определенном местоположении на странице.

Первый шаг предусматривает помещение элемента управления `ValidationSummary` в файл `*.aspx`. Дополнительно можно установить его свойство `HeaderText`, а также свойство `DisplayMode`, которое по умолчанию будет отображать список всех сообщений об ошибках в виде маркированного списка.

```
<asp:ValidationSummary
  id="ValidationSummary1"
  runat="server" Width="353px"
  HeaderText="Here are the things you
must correct.">
</asp:ValidationSummary>
```

Далее понадобится установить свойство `Display` в `None` для всех индивидуальных элементов управления проверкой достоверности (например, `RequiredFieldValidator`, `RangeValidator` и т.д.). Это гарантирует отсутствие дублированных сообщений об ошибках при каждой неудачной проверке достоверности (одно в итоговой панели и еще одно в месте расположения элемента управления проверкой достоверности). На рис. 32.22 приведена итоговая панель в действии.

И последнее: если взамен вы хотите отображать сообщения об ошибках в окне сообщений клиентской стороны, то установите свойство `ShowMessageBox` элемента управления `ValidationSummary` в `true`, а свойство `ShowSummary` — в `false`.

Определение групп проверки достоверности

Также есть возможность определять *группы*, к которым принадлежат элементы управления проверкой достоверности. Такой прием очень удобен при наличии на странице областей, работающих как единое целое. Например, может существовать одна группа элементов управления в объекте `Panel`, предназначенная для ввода пользователем адреса электронной почты, и другая группа в другом объекте `Panel` для ввода информации о кредитной карте. Используя группы, можно сконфигурировать каждый набор элементов управления для независимой проверки достоверности.

Рис. 32.23. Объекты `Panel` будут независимо конфигурировать свои области ввода

Вставьте в текущий проект новую страницу по имени `ValidationGroups.aspx`, на которой определены два элемента управления `Panel`. Первый объект `Panel` будет содержать элемент `TextBox` для пользовательского ввода (проверяемого посредством `RequiredFieldValidator`), а во втором объекте `Panel` будет находиться элемент `TextBox` для ввода номера карточки социального страхования США (проверяемого с помощью `RegularExpressionValidator`). На рис. 32.23 показан возможный вид пользовательского интерфейса.

Чтобы обеспечить независимое функционирование элементов управления проверкой достоверности, просто назначьте элемент управления проверкой достоверности и проверяемый элемент управления уникально именованной группе с применением свойства ValidationGroup. В следующем примере разметки обратите внимание на то, что используемые обработчики события Click по существу являются пустыми заглушками в файле кода:

```
<form id="form1" runat="server">
  <asp:Panel ID="Panel1" runat="server" Height="83px" Width="296px">
    <asp:TextBox ID="txtRequiredData" runat="server"
                 ValidationGroup="FirstGroup">
    </asp:TextBox>
    <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
      ErrorMessage="*Required field!" ControlToValidate="txtRequiredData"
      ValidationGroup="FirstGroup">
    </asp:RequiredFieldValidator>
    <asp:Button ID="bntValidateRequired" runat="server"
        OnClick="bntValidateRequired_Click"
        Text="Validate" ValidationGroup="FirstGroup" />
  </asp:Panel>
  <asp:Panel ID="Panel2" runat="server" Height="119px" Width="295px">
    <asp:TextBox ID="txtSSN" runat="server"
      ValidationGroup="SecondGroup">
    </asp:TextBox>
    <asp:RegularExpressionValidator ID="RegularExpressionValidator1"
        runat="server" ControlToValidate="txtSSN"
        ErrorMessage="*Need SSN" ValidationExpression="\d{3}-\d{2}-\d{4}"
        ValidationGroup="SecondGroup">
    </asp:RegularExpressionValidator> 
    <asp:Button ID="btnValidateSSN" runat="server"
        OnClick="btnValidateSSN_Click" Text="Validate"
        ValidationGroup="SecondGroup" />
  </asp:Panel>
</form>
```

Теперь щелкните правой кнопкой мыши на поверхности визуального конструктора этой страницы и выберите в контекстном меню пункт View In Browser (Просмотреть в браузере), чтобы удостовериться, что проверка данных в элементах каждой панели происходит во взаимоисключающей манере.

Проверка достоверности с помощью аннотаций данных

В дополнение к элементам управления проверкой достоверности инфраструктура ASP.NET Web Forms поддерживает проверку достоверности с применением аннотаций данных. Вспомните из главы 23, что классы модели можно помечать атрибутами, которые определяют бизнес-требования для модели (скажем, Required). Используя новый элемент управления ModelErrorMessage и дополнительное свойство элемента управления ValidationSummary, можно представлять ошибки, связанные с нарушениями аннотаций данных, с помощью совсем небольшого объема кода.

Создание модели

Хотя определенно можно было бы обратиться к библиотеке AutoLotDAL из главы 31, в целях упрощения мы создадим новый класс Inventory. Начните с добавления новой папки App_Code, щелкнув правой кнопкой мыши на имени проекта и выбрав в контекстном меню пункт Add⇨Add ASP.NET Folder⇨App_Code (Добавить⇨Добавить папку

ASP.NET⇨App_Code). Добавьте в эту папку новый файл класса по имени Inventory.cs и поместите в него следующий код (не забудьте об операторе using для пространства имен System.ComponentModel.DataAnnotations):

```
public class Inventory
{
  [Key,Required]
  public int CarID { get; set; }

  [Required(ErrorMessage="Make is required.")]
  [StringLength(30,ErrorMessage="Make can only be 30 charaters or less")]
  public string Make { get; set; }

  [Required, StringLength(30)]
  public string Color { get; set; }

  [StringLength(30, ErrorMessage = "Pet Name can only be 30 charaters or less")]
  public string PetName { get; set; }
}
```

Построение пользовательского интерфейса

Далее добавьте новую веб-форму по имени Annotations.aspx. Поместите внутрь дескриптора Form элемент управления asp:FormView. Элемент управления FormView позволяет очень просто переключаться между режимами отображения, редактирования и вставки. Модифицируйте атрибуты, как показано ниже (установка DefaultMode="Insert" приводит к загрузке элемента управления FormView в режиме вставки):

```
<asp:FormView runat="server" ID="fvDataBinding" DataKeyNames="CarID"
     ItemType="Inventory" DefaultMode="Insert" InsertMethod="SaveCar"
     UpdateMethod="UpdateCar" SelectMethod="GetCar">
```

Теперь создайте разметку ItemTemplate. Это содержимое, которое будет отображаться в режиме только для чтения. Атрибут ItemType строго типизирует элемент управления FormView и поддерживает синтаксис <%# Item.*ИмяПоля* %>. Добавьте следующую разметку:

```
<ItemTemplate>
  <table style="width:100%">
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="make">Make:
          </asp:Label></td>
      <td><asp:Label runat="server" ID="make" Text='<%# Item.Make %>' /></td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="color">Color:
          </asp:Label></td>
      <td><asp:Label runat="server" ID="color" Text='<%#: Item.Color %>' />
      </td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="petname">Pet Name:
          </asp:Label></td>
      <td><asp:Label runat="server" ID="customerAge"
                     Text='<%#: Item.PetName %>' /></td>
    </tr>
    <tr>
      <td colspan="2">
```

```
      <asp:Button ID="EditButton" runat="server" CommandName="Edit"
                  Text="Edit" /> 
    </td>
   </tr>
  </table>
</ItemTemplate>
```

Атрибут `CommandName="Edit"` **указывает на то, что щелчок на кнопке будет переводить** `FormView` **в режим редактирования, в котором отображается разметка** `EditItemTemplate`**, создаваемая следующей. Существует ряд отличий между синтаксисом для шаблонов редактирования и отображения. Прежде всего, в шаблонах редактирования вместо синтаксиса** `<%# Item.`*ИмяПоля*`%>` **применяется синтаксис** `<%# BindItem.`*ИмяПоля*`%>`**. С помощью** `BindItem` **элемент управления настраивается на двухстороннюю привязку. Еще одно отличие касается элемента управления** `ModelErrorMessage`**, сопровождающего элементы управления редактированием. Элемент** `ModelErrorMessage` **будет отображать любые ошибки привязки модели для свойства, указанного в** `ModelStateKey`**. Обратите внимание, что это зависит от строгой типизации** `FormView`**.**

```
<EditItemTemplate>
  <table style="width:100%">
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="make">Make:
          </asp:Label></td>
      <td>
        <asp:TextBox runat="server" ID="make" Text='<%# BindItem.Make %>' />
        <asp:ModelErrorMessage ModelStateKey="make"
            runat="server" ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="color">Color:
          </asp:Label></td>
      <td>
       <asp:TextBox runat="server" ID="color" Text='<%#: BindItem.Color %>' />
       <asp:ModelErrorMessage ModelStateKey="color"
            runat="server" ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="petname">Pet Name:
          </asp:Label></td>
      <td>
        <asp:TextBox ID="petname" runat="server"
            Text='<%#: BindItem.PetName %>' />
        <asp:ModelErrorMessage ModelStateKey="petname" runat="server"
            ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td colspan="2">
        <asp:Button runat="server" CommandName="Update" Text="Save" />
        <asp:Button runat="server" CommandName="Cancel" Text="Cancel"
            CausesValidation="false" />
      </td>
    </tr>
  </table>
</EditItemTemplate>
```

Разметка в `InsertItemTemplate` фактически является той же самой, как и в `EditItemTemplate`, с единственной разницей (помимо имени дескриптора), состоящей в том, что именем команды (`CommandName`) для `Button` будет `Insert`:

```
<InsertItemTemplate>
  <table style="width:100%">
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="make">Make:
          </asp:Label></td>
      <td>
        <asp:TextBox runat="server" ID="make" Text='<%#: BindItem.Make %>' />
        <asp:ModelErrorMessage ModelStateKey="make" runat="server"
            ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="color">Color:
          </asp:Label></td>
      <td>
       <asp:TextBox runat="server" ID="color" Text='<%#: BindItem.Color %>' />
       <asp:ModelErrorMessage ModelStateKey="color"
           runat="server" ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="petname">Pet Name:
          </asp:Label></td>
      <td>
        <asp:TextBox ID="petname" runat="server"
                     Text='<%#: BindItem.PetName %>' />
        <asp:ModelErrorMessage ModelStateKey="petname" runat="server"
            ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td colspan="2">
        <asp:Button runat="server" CommandName="Insert" Text="Save" />
      </td>
    </tr>
  </table>
</InsertItemTemplate>
```

Закройте дескриптор элемент управления `FormView`:

```
</asp:FormView>
```

Наконец, добавьте элемент управления `ValidationSummary`. Отличие от предшествующих примеров здесь в том, что свойство `ShowModelStateErrors` устанавливается в `true`. Это инструктирует элемент управления об отображении любых ошибок привязки модели. После закрывающего дескриптора элемента управления `FormView` добавьте приведенную ниже разметку:

```
<asp:ValidationSummary runat="server" ShowModelStateErrors="true"
  ForeColor="Red" HeaderText="Please check the following errors:" />
```

Добавление кода

Мы собираемся добавить объем кода, достаточный для демонстрации проверки достоверности. В реальном приложении методы, поддерживающие элемент управления FormView, обращались бы к уровню DAL, а не только взаимодействовали бы с локальной переменной. Но ради сохранения простоты примера добавьте в Annotations.aspx.cs следующий код. Вам придется добавить NuGet-пакет Microsoft.CodeDom.Providers.DotNetCompilerPlatform, чтобы включить функциональные средства C# 6. Обратите внимание на использование неявной привязки модели в методе SaveCar() и явной привязки модели в методе UpdateCar().

```
private Inventory _model = null;
public void SaveCar(Inventory car)
{
  if (ModelState.IsValid)
  {
    _model = car;
    // Добавить новую запись
  }
}
public void UpdateCar(int carID)
{
  Inventory car = new Inventory();
  if (TryUpdateModel(car))
  {
    _model = car;
    // Обновить запись
  }
}
public Inventory GetCar() => _model;
```

Тестирование приложения

Запустив приложение, вы увидите страницу, подобную показанной на рис. 32.24.

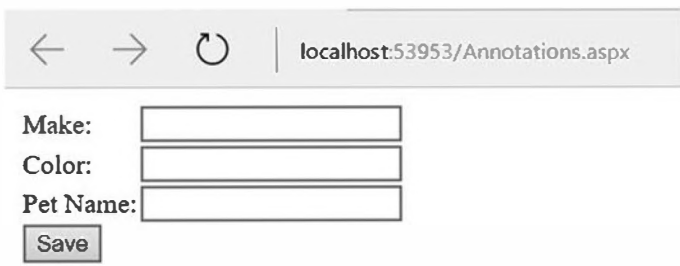


Рис. 32.24. Элемент управления FormView в режиме вставки

Оставьте поля Make (Производитель) и Color (Цвет) пустыми, введите в поле Pet Name (Дружественное имя) строку длиннее 30 символов и щелкните на кнопке Save (Сохранить). Вы должны получить те же самые ошибки, что и на рис. 32.25.

Как и другие элементы управления Web Forms, элементы ModelErrorMessage и ValidationSummary могут быть стилизованы намного лучше, чем было сделано здесь.

Исходный код. Веб-сайт ValidatorCtrls доступен в подкаталоге Chapter_32.

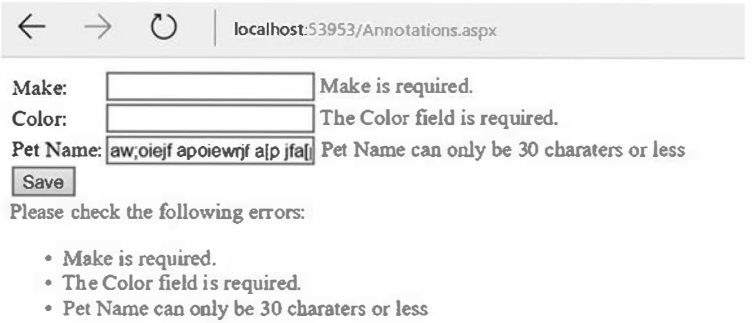


Рис. 32.25. Элемент управления `FormView` и отображение сообщений об ошибках

Работа с темами

К настоящему моменту вы поработали с многочисленными элементами управления Web Forms. Вы видели, что каждый элемент управления открывает доступ к набору свойств (многие из которых унаследованы от `System.Web.UI.Webcontrols.WebControl`), позволяющих настраивать внешний вид и поведение пользовательского интерфейса (цвет фона, размер шрифта, стиль рамки и т.п.). Конечно, на многостраничном веб-сайте принято определять общий внешний вид и поведение виджетов различных типов. Например, все элементы `TextBox` могут быть сконфигурированы на поддержку заданного шрифта, все элементы `Button` — принятого в компании вида, а все `Calendar` — ярко-синей рамки.

Очевидно, что установка *одних и тех же* значений для свойств *каждого* виджета на *каждой* странице веб-сайта была бы очень трудоемкой (и чреватой ошибками) задачей. Даже если вы в состоянии вручную обновить свойства каждого виджета пользовательского интерфейса на каждой странице, только вообразите, насколько утомительным может стать процесс изменения цвета фона для каждого элемента `TextBox`, когда в этом возникнет необходимость. Ясно, что должен существовать более эффективный способ применения настроек пользовательского интерфейса на уровне всего сайта.

Один из подходов к упрощению установки общего внешнего вида и поведения пользовательского интерфейса предусматривает определение *таблиц стилей*. Если у вас есть опыт разработки веб-приложений, то вам известно, что таблицы стилей определяют общий набор настроек пользовательского интерфейса, которые применяются в браузере. Как и можно было ожидать, элементам управления Web Forms может быть назначен стиль за счет установки свойства `CssStyle`.

Тем не менее, инфраструктура Web Forms поставляется с дополняющей технологией для определения общего пользовательского интерфейса, которая называется *темами*. В отличие от таблиц стилей темы применяются на веб-сервере (а не в браузере) и могут использоваться программно или декларативно. Поскольку тема применяется на веб-сервере, она имеет доступ ко всем ресурсам веб-сайта серверной стороны. Более того, темы определяются путем написания той же разметки, которую можно найти в любом файле `*.aspx` (вы наверняка согласитесь, что синтаксис таблиц стилей чересчур сжат).

Вспомните из главы 31, что веб-приложения ASP.NET могут определять любое количество специальных каталогов, одним из которых является `App_Themes`. Этот подкаталог может иметь подкаталоги, каждый из которых представляет одну из возможных тем веб-сайта. Например, на рис. 32.26 показан подкаталог `App_Themes`, содержащий три подкаталога, каждый из которых имеет набор файлов, образующих саму тему.

Рис. 32.26. Один подкаталог `App_Themes` может определять многочисленные темы

Файлы `*.skin`

Каждый подкаталог темы обязательно содержит файл обложки `*.skin`. Эти файлы определяют внешний вид и поведение различных веб-элементов управления. В целях иллюстрации создайте новый проект пустого веб-сайта по имени `FunWithThemes` и вставьте в него новую веб-форму `Default.aspx`. Поместите на нее элементы управления `Calendar`, `TextBox` и `Button`. Конфигурировать каким-то специальным образом эти элементы управления не понадобится, а их точные имена не важны. Как будет показано, данные элементы управления будут служить целями для специальных обложек.

Добавьте новый файл `*.skin` (используя пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент)) по имени `BasicGreen.skin` (рис. 32.27).

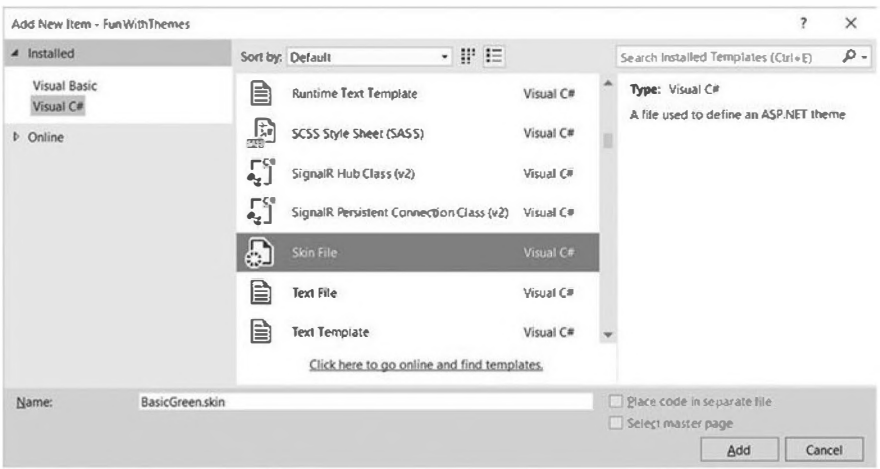


Рис. 32.27. Вставка файла `*.skin`

Среда Visual Studio предложит подтвердить добавление этого файла в `App_Themes` (что как раз и требуется). Если теперь заглянуть в окно Solution Explorer, то в подкаталоге `App_Themes` будет виден подкаталог `BasicGreen`, который содержит новый файл `BasicGreen.skin`.

В файле `*.skin` определяется внешность и поведение разнообразных виджетов с применением синтаксиса для объявления элементов управления Web Forms. К сожалению, в IDE-среде отсутствует поддержка визуального конструирования файлов `*.skin`. Один из способов сокращения объема ввода заключается в добавлении к программе временного файла `*.aspx` (скажем, `temp.aspx`), который может использоваться при построении пользовательского интерфейса виджетов с помощью визуального конструктора страниц Visual Studio.

Результирующую разметку можно затем скопировать в файл `*.skin`. Однако вы *должны* удалить атрибут `ID` из каждого веб-элемента управления! Смысл вполне очевиден: мы хотим определить внешний вид и поведение не *отдельного* элемента `Button` (например), а *всех* элементов `Button`.

Учитывая сказанное, вот как может выглядеть разметка внутри файла `BasicGreen.skin`, которая определяет стандартный внешний вид и поведение для типов `Button`, `TextBox` и `Calendar`:

```
<asp:Button runat="server" BackColor="#80FF80"/>
<asp:TextBox runat="server" BackColor="#80FF80"/>
<asp:Calendar runat="server" BackColor="#80FF80"/>
```

Обратите внимание, что каждый виджет по-прежнему содержит атрибут `runat="server"` (что обязательно), и ни одному из них не назначен атрибут `ID`.

Давайте определим вторую тему по имени `CrazyOrange`. В окне Solution Explorer щелкните правой кнопкой мыши на папке `App_Themes` и добавьте новую тему по имени `CrazyOrange`. В итоге внутри папки `App_Themes` сайта создается новый подкаталог.

Затем щелкните правой кнопкой мыши на новой папке `CrazyOrange` в окне Solution Explorer и выберите в контекстном меню пункт Add New Item (Добавить новый элемент). В открывшемся диалоговом окне добавьте новый файл `*.skin`. Модифицируйте файл `CrazyOrange.skin`, определив уникальный внешний вид и поведение для тех же самых веб-элементов управления:

```
<asp:Button runat="server" BackColor="#FF8000"/>
<asp:TextBox runat="server" BackColor="#FF8000"/>
<asp:Calendar BackColor="White" BorderColor="Black"
  BorderStyle="Solid" CellSpacing="1"
  Font-Names="Verdana" Font-Size="9pt" ForeColor="Black" Height="250px"
  NextPrevFormat="ShortMonth" Width="330px" runat="server">
  <SelectedDayStyle BackColor="#333399" ForeColor="White" />
  <OtherMonthDayStyle ForeColor="#999999" />
  <TodayDayStyle BackColor="#999999" ForeColor="White" />
  <DayStyle BackColor="#CCCCCC" />
  <NextPrevStyle Font-Bold="True" Font-Size="8pt" ForeColor="White" />
  <DayHeaderStyle Font-Bold="True" Font-Size="8pt"
    ForeColor="#333333" Height="8pt" />
  <TitleStyle BackColor="#333399"
      BorderStyle="Solid"
    Font-Bold="True" Font-Size="12pt"
    ForeColor="White" Height="12pt" />
</asp:Calendar>
```

В данный момент окно Solution Explorer должно выглядеть так, как показано на рис. 32.28.

Теперь, когда сайт имеет несколько определенных тем, следующий логический шаг заключается в их применении к страницам. Как и можно было догадаться, делать это можно многими способами.

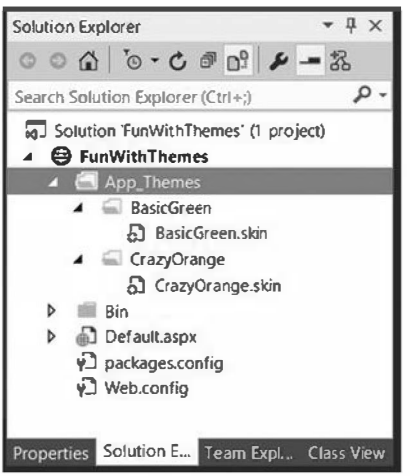


Рис. 32.28. Веб-сайт с несколькими темами

На заметку! Дизайн рассматриваемых примеров тем довольно прост (с целью экономии места на печатной странице). Вы можете усовершенствовать их по своему вкусу.

Применение тем ко всему сайту

Если вы хотите обеспечить оформление каждой страницы сайта согласно той же самой теме, то проще всего обновить файл `Web.config`. Откройте текущий файл `Web.config` и определите элемент `<pages>` внутри области корневого элемента `<system.web>`. Добавление атрибута `theme` к элементу `<pages>` гарантирует применение одной и той же темы ко всем страницам сайта (разумеется, значением атрибута должно быть имя одного из подкаталогов внутри `App_Themes`). Вот основное изменение:

```
<configuration>
  <system.web>
    ...
    <pages controlRenderingCompatibilityVersion="4.5"
           theme="BasicGreen">
    </pages>
  </system.web>
</configuration>
```

Открыв эту страницу, вы обнаружите, что каждый виджет имеет пользовательский интерфейс, определенный темой `BasicGreen`. Если изменить значение атрибута `theme` на `CrazyOrange` и снова открыть страницу, то пользовательский интерфейс будет соответствовать тому, который определен темой `CrazyOrange`.

Применение тем на уровне страницы

Темы также можно назначать на уровне страниц. Такой прием удобен в различных обстоятельствах. Например, в файле `Web.config` может быть определена тема для всего сайта (как описано в предыдущем разделе), но вы хотите назначить определенной странице другую тему. В таком случае понадобится просто обновить директиву `<%@ Page %>`. При использовании Visual Studio средство IntelliSense отобразит все доступные темы, определенные в папке `App_Themes`.

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" Theme ="CrazyOrange" %>
```

Из-за того, что данной странице назначена тема `CrazyOrange`, а в файле `Web.config` указана тема `BasicGreen`, все страницы *кроме этой* будут визуализированы с применением темы `BasicGreen`.

Свойство `SkinID`

Временами может возникнуть необходимость в определении набора возможных внешних видов и линий поведения для одиночного виджета. Например, предположим, что нужно определить два пользовательских интерфейса для типа `Button` внутри темы `CrazyOrange`. Различать варианты внешнего вида и поведения можно с использованием свойства `SkinID` элемента управления внутри файла `*.skin`:

```
<asp:Button runat="server" BackColor="#FF8000"/>
<asp:Button runat="server" SkinID = "BigFontButton"
            Font-Size="30pt" BackColor="#FF8000"/>
```

Теперь при наличии страницы, которая работает с темой `CrazyOrange`, каждому элементу `Button` по умолчанию будет назначена неименованная обложка `Button`. Если необходимо, чтобы в файле `*.aspx` обложку `BigFontButton` имело несколько разных кнопок, то следует просто указать свойство `SkinID` внутри разметки:

```
<asp:Button ID="Button2" runat="server"
            SkinID="BigFontButton" Text="Button" /><br />
```

Назначение тем программным образом

И последнее, но не менее важное: темы допускается назначать в коде. Это может пригодиться, когда конечным пользователям должен быть предоставлен способ выбора темы для текущего сеанса. Разумеется, пока еще не было показано, каким образом строить веб-приложения с поддержкой состояния, так что выбранная тема будет утеряна между обратными отправками. На сайте производственного уровня выбранная в текущий момент тема пользователя сохраняется внутри переменной сеанса или же записывается в базу данных.

Чтобы продемонстрировать назначение темы программным образом, добавьте к пользовательскому интерфейсу в файле `Default.aspx` три новых элемента управления `Button` (рис. 32.29). Затем для каждого элемента `Button` обработайте событие `Click`.

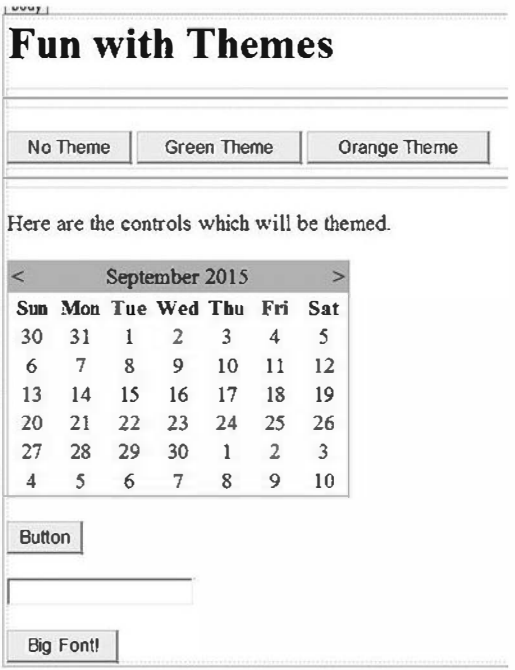


Рис. 32.29. Модифицированный пользовательский интерфейс примера работы с темами

Имейте в виду, что назначать тему в коде можно только на определенных фазах жизненного цикла страницы. Обычно это делается внутри обработчика события `Page_PreInit`. С учетом сказанного модифицируйте файл кода, как показано ниже:

```
partial class _Default : System.Web.UI.Page
{
  protected void btnNoTheme_Click(object sender, System.EventArgs e)
  {
    // Пустая строка означает, что никакая тема не применена.
    Session["UserTheme"] = "";

    // Снова инициировать событие PreInit.
    Server.Transfer(Request.FilePath);
  }
```

```
protected void btnGreenTheme_Click(object sender, System.EventArgs e)
{
  Session["UserTheme"] = "BasicGreen";
  // Снова инициировать событие PreInit.
  Server.Transfer(Request.FilePath);
}
protected void btnOrangeTheme_Click(object sender, System.EventArgs e)
{
  Session["UserTheme"] = "CrazyOrange";
  // Снова инициировать событие PreInit.
  Server.Transfer(Request.FilePath);
}
protected void Page_PreInit(object sender, System.EventArgs e)
{
  try
  {
    Theme = Session["UserTheme"].ToString();
  }
  catch
  {
    Theme = "";
  }
}
}
```

Обратите внимание, что выбранная тема сохраняется в переменной сеанса (детали ищите в главе 33) по имени `UserTheme`, значение которой формально присваивается внутри обработчика событий `Page_PreInit()`. Когда пользователь щелкает на заданном элементе управления `Button`, программно инициируется событие `PreInit` путем вызова метода `Server.Transfer()` и запрашивания текущей страницы еще раз. Загрузив страницу в браузер, вы обнаружите, что темы можно устанавливать посредством щелчков на разных элементах `Button`.

Исходный код. Веб-сайт `FunWithThemes` доступен в подкаталоге `Chapter_32`.

Резюме

В настоящей главе было показано, как использовать разнообразные элементы управления Web Forms. Сначала вы изучили роль базовых классов `Control` и `WebControl`, а затем узнали, каким образом динамически взаимодействовать с внутренней коллекцией элементов управления панели. Попутно вы ознакомились с новой моделью навигации по сайту (файлы `*.sitemap` и компонент `SiteMapDataSource`), новым механизмом привязки данных и различными элементами управления проверкой достоверности.

Вторая половина главы была посвящена обсуждению роли мастер-страниц и тем. Мастер-страницы могут быть задействованы с целью определения общей компоновки для набора страниц сайта. В файле `*.master` определяется любое количество мест заполнения, куда страницы содержимого подключают свое специальное содержимое пользовательского интерфейса. Наконец, вы узнали, что механизм тем Web Forms позволяет декларативно или программно применять общий внешний вид и поведение пользовательского интерфейса к виджетам на веб-сервере.

ГЛАВА 33

Управление состоянием в ASP.NET

В предшествующих двух главах внимание было сосредоточено на структуре и поведении страниц ASP.NET, а также на веб-элементах управления, которые они содержат. Настоящая глава основана на этой информации и посвящена роли файла `Global.asax` и лежащего в основе типа `HttpApplication`. Вы увидите, что функциональность `HttpApplication` позволяет перехватывать многочисленные события, которые делают возможной трактовку веб-приложения как единого целого, а не набора автономных файлов `*.aspx`, управляемых мастер-страницей.

В дополнение к исследованию типа `HttpApplication` здесь также рассматриваются все важные темы, касающиеся управления состоянием. Вы узнаете о роли состояния представления, переменных сеанса и приложения (включая кеш приложения), cookie-данных и API-интерфейса профилей ASP.NET (ASP.NET Profile API).

Проблема поддержки состояния

В начале главы 31 было указано, что HTTP является протоколом *без хранения состояния*. Данный факт делает разработку веб-приложений совершенно отличающейся от процесса построения исполняемой сборки. Например, при создании приложения с настольным пользовательским интерфейсом Windows можно рассчитывать на то, что переменные-члены, определенные в производном от `Form` классе, обычно будут существовать в памяти до тех пор, пока пользователь явно не завершит исполняемую программу:

```
public partial class MainWindow : Window
{
  // Данные состояния!
  private string userFavoriteCar = "Yugo";
}
```

Однако в среде World Wide Web нельзя исходить из такого же удобного предположения. Чтобы проверить это, создайте новый проект пустого веб-сайта по имени `SimpleStateExample` и вставьте в него веб-форму. В файле отделенного кода определите строковую переменную уровня страницы по имени `userFavoriteCar`:

```
public partial class _Default : System.Web.UI.Page
{
  // Данные состояния?
  private string userFavoriteCar = "Yugo";
  protected void Page_Load(object sender, EventArgs e)
  {
  }
}
```

Затем сконструируйте очень простой пользовательский интерфейс, который показан на рис. 33.1.

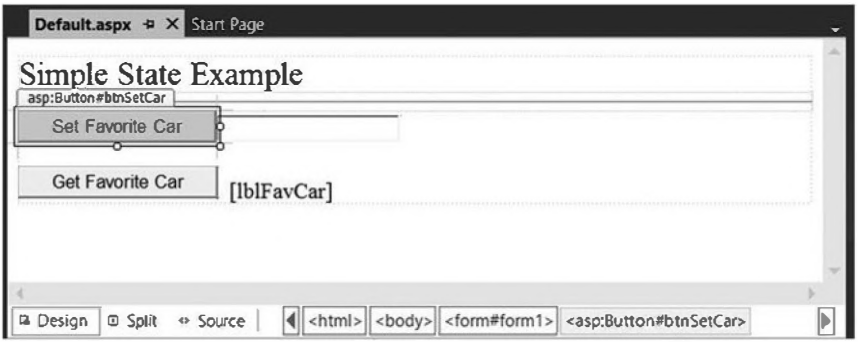


Рис. 33.1. Пользовательский интерфейс для простой страницы с состоянием

Обработчик события `Click` серверной стороны для кнопки Set Favorite Car (Установить предпочитаемый автомобиль) по имени `btnSetCar` позволяет пользователю присвоить переменной-члену типа `string` значение, находящееся внутри элемента `TextBox` (с именем `txtFavCar`):

```
protected void btnSetCar_Click(object sender, EventArgs e)
{
  // Сохранить предпочитаемый автомобиль в переменной-члене.
  userFavoriteCar = txtFavCar.Text;
}
```

Обработчик события `Click` для кнопки Get Favorite Car (Получить предпочитаемый автомобиль) по имени `btnGetCar` отображает текущее значение переменной-члена внутри элемента `Label` (с именем `lblFavCar`) страницы:

```
protected void btnGetCar_Click(object sender, EventArgs e)
{
  // Отобразить значение переменной-члена.
  lblFavCar.Text = userFavoriteCar;
}
```

При построении приложения с графическим пользовательским интерфейсом Windows вполне корректно предполагать, что после того, как пользователь установил начальное значение, оно запоминается на все время жизни настольного приложения. К сожалению, запустив созданное веб-приложение, вы обнаружите, что всякий раз, когда осуществляется обратная отправка веб-серверу (посредством щелчка на любой из кнопок), для строковой переменной `userFavoriteCar` устанавливается начальное значение `"Yugo"`. Следовательно, текст в `Label` постоянно фиксирован.

Поскольку протокол HTTP не имеет ни малейшего понятия о том, как автоматически запоминать данные после отправки HTTP-ответа, само собой разумеется, что объект `Page` уничтожается практически мгновенно. В результате, когда клиент отправляет обратно файл `*.aspx`, конструируется новый объект `Page`, который сбрасывает значение всех переменных-членов уровня страницы. Безусловно, это серьезная проблема. Вообразите, до какой степени бесполезной была бы онлайновая торговля, если бы вся введенная ранее информация (вроде наименований приобретаемых товаров) отбрасывалась при каждой отправке данных веб-серверу. Когда необходимо запоминать информацию о пользователях, вошедших на сайт, приходится использовать разнообразные приемы управления состоянием.

На заметку! Описанная проблема никоим образом не ограничивается ASP.NET. Веб-приложения Java, приложения CGI, классические приложения ASP и приложения PHP также сталкиваются с необходимостью решения задачи управления состоянием.

Один из подходов к запоминанию значения строковой переменной `userFavoriteCar` между обратными отправками предусматривает сохранение этого значения в *переменной сеанса*. Состояние сеанса будет детально исследоваться далее в главе. Тем не менее, ради полноты ниже показаны необходимые изменения для текущей страницы (обратите внимание, что закрытая переменная-член типа `string` больше не применяется, так что ее определение можно закомментировать либо вообще удалить):

```
public partial class _Default : System.Web.UI.Page
{
  // Данные состояния?
  // private string userFavoriteCar = "Yugo";

  protected void Page_Load(object sender, EventArgs e)
  {
  }

  protected void btnSetCar_Click(object sender, EventArgs e)
  {
    // Сохранить значение в переменной сеанса.
    Session["UserFavCar"] = txtFavCar.Text;
  }

  protected void btnGetCar_Click(object sender, EventArgs e)
  {
    // Получить значение из переменной сеанса.
    lblFavCar.Text = (string)Session["UserFavCar"];
  }
}
```

Если теперь запустить приложение, то значение предпочитаемого автомобиля будет сохранено между обратными отправками благодаря объекту `HttpSessionState`, которым можно манипулировать косвенно через унаследованное свойство `Session`. Данные сеанса (которые будут подробно рассматриваться позже в главе) — лишь один способ "запоминания" информации на веб-сайтах. В последующих разделах вы ознакомитесь с другими возможными вариантами, поддерживаемыми ASP.NET.

Исходный код. Веб-сайт `SimpleStateExample` доступен в подкаталоге `Chapter_33`.

Приемы управления состоянием ASP.NET

Инфраструктура ASP.NET предоставляет несколько механизмов, которые можно использовать для удержания информации о состоянии в веб-приложениях. Ниже перечислены распространенные варианты:

- применение состояния представления ASP.NET;
- использование состояния элемента управления ASP.NET;
- определение переменных уровня приложения;
- применение объекта кеша;
- определение переменных уровня сеанса;
- определение cookie-данных.

В дополнение к указанным приемам инфраструктура ASP.NET предлагает готовый API-интерфейс Profile для хранения пользовательских данных на постоянной основе. Мы по очереди рассмотрим детали каждого подхода, начиная с состояния представления ASP.NET.

Роль состояния представления ASP.NET

Термин *состояние представления* уже несколько раз встречался здесь и в предшествующих главах, но без формального определения, поэтому давайте проясним его прямо сейчас. Без поддержки со стороны инфраструктуры разработчики веб-приложений вынуждены вручную перезаполнять значениями виджеты ввода на форме во время процесса конструирования исходящего HTTP-ответа.

В случае использования ASP.NET мы больше не обязаны вручную собирать и заново заполнять значениями виджеты HTML, т.к. исполняющая среда ASP.NET автоматически встраивает в форму скрытое поле (по имени `__VIEWSTATE`), которое будет передаваться между браузером и специфической страницей. Данные, присваиваемые этому полю, являются закодированной по алгоритму Base64 строкой, которая содержит набор пар "имя-значение", представляющих значения всех виджетов пользовательского интерфейса на текущей странице.

За чтение входных значений из поля `__VIEWSTATE` с целью заполнения соответствующих переменных-членов в производном классе несет ответственность обработчик события `Init` базового класса `System.Web.UI.Page`. (Именно по этой причине обращаться к состоянию виджета внутри области действия обработчика события `Init` страницы в лучшем случае рискованно.)

Кроме того, непосредственно перед отправкой исходящего ответа запросившему браузеру данные `__VIEWSTATE` применяются для повторного заполнения виджетов формы. Очевидно, самая лучшая характеристика этого аспекта ASP.NET заключается в том, что все происходит без какого-либо участия с вашей стороны. Разумеется, при необходимости всегда можно взаимодействовать, изменять или отключать такую стандартную функциональность. Чтобы понять, как это делается, давайте рассмотрим конкретный пример работы с состоянием представления.

Демонстрация работы с состоянием представления

Создайте новый проект пустого веб-сайта по имени `ViewStateApp` и добавьте в него новую веб-форму под названием `Default.aspx`. Щелкните правой кнопкой мыши на имени проекта и выберите в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet). Добавьте NuGet-пакет `Microsoft.CodeDom.Providers.DotNetCompilerPlatform`, который предоставляет веб-сайту возможности C# 6. Добавьте на страницу `*.aspx` веб-элемент управления `ListBox` по имени `myListBox` и элемент управления `Button` по имени `btnPostback`.

На заметку! Для всех примеров в настоящей главе понадобится добавлять NuGet-пакет `Microsoft.CodeDom.Providers.DotNetCompilerPlatform`, чтобы обеспечить поддержку возможностей C# 6.

В окне Properties (Свойства) среды Visual Studio отыщите свойство `Items` и добавьте в `ListBox` четыре элемента `ListItem`, используя ассоциированное диалоговое окно. Результирующая разметка должна выглядеть так:

```
<asp:ListBox ID="myListBox" runat="server">
  <asp:ListItem>Item One</asp:ListItem>
  <asp:ListItem>Item Two</asp:ListItem>
```

```
  <asp:ListItem>Item Three</asp:ListItem>
  <asp:ListItem>Item Four</asp:ListItem>
</asp:ListBox>
```

Обратите внимание, что элементы списка `ListBox` жестко закодированы внутри файла `*.aspx`. Вы уже знаете, что перед отправкой финального HTTP-ответа все определения `<asp:>` в веб-форме ASP.NET будут автоматически преобразованы в свои представления HTML (если только они снабжены атрибутом `runat="server"`).

Директива `<%@ Page %>` имеет необязательный атрибут по имени `EnableViewState`, который по умолчанию установлен в `true`. Чтобы отключить поведение состояния представления, измените директиву `<%@ Page %>` следующим образом:

```
<%@ Page Language="C#" AutoEventWireup="true"
         CodeFile="Default.aspx.cs" Inherits="_Default"
         EnableViewState ="false" %>
```

Итак, что же в точности означает отключение состояния представления? Ответ: в зависимости от обстоятельств. Исходя из предыдущего определения термина, могло бы показаться, что если отключить состояние представления для файла `*.aspx`, то значения в `ListBox` не будут запоминаться между обратными отправками веб-серверу. Однако если вы запустите приложение в том виде, как есть, то можете быть удивлены, увидев, что информация в `ListBox` сохраняется независимо от того, сколько раз производится обратная отправка страницы.

На самом деле, если вы просмотрите разметку HTML, возвращенную браузеру (щелкнув правой кнопкой мыши на странице внутри браузера и выбрав в контекстном меню пункт View Source (Исходный код страницы)), то удивитесь еще больше, обнаружив, что скрытое поле `__VIEWSTATE` *по-прежнему присутствует*:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
       value="/wEPDwUKLTM4MTM2MDM4NGRkqGC6gjEV25JnddkJiRmoIc10SIA=" />
```

Тем не менее, предположим, что элемент управления `ListBox` заполняется динамически в файле отделенного кода, а не в HTML-дескрипторе `<form>`. Для начала удалите объявления `<asp:ListItem>` из текущего файла `*.aspx`:

```
<asp:ListBox ID="myListBox" runat="server">
</asp:ListBox>
```

Затем заполните список элементами внутри обработчика события `Load` в файле отделенного кода:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
  {
    // Заполнить ListBox динамически!
    myListBox.Items.Add("Item One");
    myListBox.Items.Add("Item Two");
    myListBox.Items.Add("Item Three");
    myListBox.Items.Add("Item Four");
  }
}
```

Отправив эту обновленную страницу, вы увидите, что при первом ее запросе браузером значения в `ListBox` присутствуют в том виде, как они были добавлены в коде. Однако после обратной отправки элемент управления `ListBox` неожиданно становится пустым. Первое правило состояния представления ASP.NET заключается в том, что его работу можно заметить только тогда, когда есть виджеты, для которых значения гене-

рируются динамически в коде. Если значения жестко закодированы внутри дескрипторов <form> файла *.aspx, то состояние этих элементов будет всегда запоминаться между обратными отправками (даже если для заданной страницы атрибут EnableViewState установлен в false).

Если идея отключения состояния представления для всего файла *.aspx кажется излишне радикальной, то имейте в виду, что каждый потомок базового класса System.Web.UI.Control наследует свойство EnableViewState, которое делает очень простым отключение состояния представления на поэлементной основе:

```
<asp:GridView id="myHugeDynamicallyFilledGridOfData" runat="server"
     EnableViewState="false">
</asp:GridView>
```

На заметку! Начиная с версии .NET 4.0, объемные значения данных состояния представления автоматически сжимаются, чтобы уменьшить размер этого скрытого поля формы.

Добавление специальных данных состояния представления

Вдобавок к свойству EnableViewState класс System.Web.UI.Control предлагает защищенное свойство по имени ViewState. "За кулисами" это свойство обеспечивает доступ к объекту типа System.Web.UI.StateBag, который представляет все данные, содержащиеся в поле __VIEWSTATE. С применением индексатора типа StateBag в скрытое поле формы VIEWSTATE можно встраивать специальную информацию, используя набор пар "имя-значение". Вот простой пример:

```
protected void btnAddToVS_Click(object sender, EventArgs e)
{
  ViewState["CustomViewStateItem"] = "Some user data";
  lblVSValue.Text = (string)ViewState["CustomViewStateItem"];
}
```

Из-за того, что тип System.Web.UI.StateBag был спроектирован для оперирования над типами System.Object, при доступе к значению по заданному ключу потребуется явно приводить его к лежащему в основе типу данных (System.String в этом случае). Тем не менее, имейте в виду, что значения, помещенные в поле __VIEWSTATE, не могут быть буквально любым объектом. В частности, допустимыми типами являются только String, Integer, bool, ArrayList, Hashtable и массивы перечисленных типов.

Следовательно, поскольку страницы *.aspx могут вставлять специальные фрагменты информации в строку __VIEWSTATE, имеет смысл выяснить, когда это может требоваться. В большинстве случаев специальные данные состояния представления лучше всего подходят для хранения пользовательских предпочтений. Скажем, можно было бы установить данные состояния представления, которые указывают, каким пользователь желает видеть интерфейс элемента GridView (например, порядок сортировки). Однако данные состояния представления не слишком хорошо подходят для хранения полномасштабных пользовательских данных, таких как элементы в корзине для покупок или кешированные объекты DataSet. Когда нужно хранить сложную информацию подобного рода, лучше работать с данными сеанса или данными приложения. Но прежде чем перейти к их рассмотрению, следует прояснить роль файла Global.asax.

Исходный код. Веб-сайт ViewStateApp доступен в подкаталоге Chapter_33.

Роль файла Global.asax

К настоящему моменту приложение ASP.NET может выглядеть всего лишь набором файлов *.aspx и соответствующих веб-элементов управления. Наряду с тем, что веб-приложение можно строить, просто связывая вместе набор веб-страниц, скорее всего, вам понадобится способ взаимодействия с веб-приложением как с единым целым. Для такой цели в веб-приложение ASP.NET можно включить дополнительный файл Global.asax через пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент), как показано на рис. 33.2. (Обратите внимание, что здесь выбирается элемент Global Application Class (Глобальный класс приложения).)

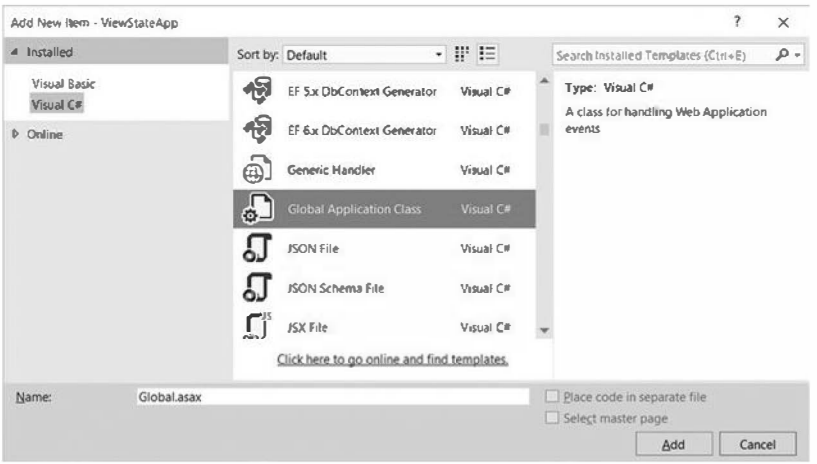


Рис. 33.2. Добавление файла Global.asax

Попросту говоря, Global.asax очень похож на запускаемый двойным щелчком файл *.exe, который можно получить в мире ASP.NET, в том смысле, что этот тип представляет поведение времени выполнения самого веб-сайта. После вставки файла Global.asax в веб-проект вы увидите, что в нем находится всего лишь блок <script>, содержащий набор обработчиков событий:

```
<%@ Application Language="C#" %>

<script runat="server">
  void Application_Start(object sender, EventArgs e)
  {
    // Код, выполняемый при запуске приложения.
  }
  void Application_End(object sender, EventArgs e)
  {
    // Код, выполняемый при прекращении работы приложения.
  }
  void Application_Error(object sender, EventArgs e)
  {
    // Код, выполняемый при возникновении необработанной ошибки.
  }
  void Session_Start(object sender, EventArgs e)
  {
    // Код, выполняемый при запуске нового сеанса.
  }
```

```
  void Session_End(object sender, EventArgs e)
  {
    // Код, выполняемый при завершении сеанса.
    // Примечание. Событие Session_End инициируется, только если в файле
    // Web.config режим состояния сеанса установлен в InProc.
    // Если режим сеанса установлен в StateServer или SQLServer,
    // то данное событие не возникает.
  }
</script>
```

Тем не менее, внешность может быть обманчивой. Во время выполнения код внутри этого блока `<script>` организуется в класс, производный от `System.Web.HttpApplication`. Следовательно, в любом из предоставленных обработчиков событий можно получать доступ к членам родительского класса посредством ключевых слов `this` или `base`.

Как уже упоминалось, определенные внутри `Global.asax` члены представляют собой обработчики событий, которые позволяют взаимодействовать с событиями уровня приложения (и уровня сеанса). Обработчики событий кратко описаны в табл. 33.1.

Таблица 33.1. Основные обработчики событий в файле `Global.asax`

Обработчик события	Описание
`Application_Start()`	Этот обработчик событий вызывается в самом начале при запуске веб-приложения. Таким образом, за время жизни веб-приложения данное событие будет инициировано только однократно. Представляет собой идеальное место для определения данных уровня приложения, применяемых повсюду в веб-приложении
`Application_End()`	Этот обработчик событий вызывается при прекращении работы приложения, которое происходит, когда последний пользователь покидает приложение по тайм-ауту или вы вручную завершаете приложение в IIS
`Session_Start()`	Этот обработчик событий вызывается, когда новый пользователь входит в приложение. Здесь можно устанавливать все специфические для пользователя данные, которые нужно сохранить между обратными отправками
`Session_End()`	Этот обработчик событий вызывается при завершении пользовательского сеанса (обычно по истечении предопределенного тайм-аута)
`Application_Error()`	Это глобальный обработчик ошибок, который вызывается, когда веб-приложение сгенерировало необработанное исключение

Глобальный обработчик исключений "последнего шанса"

Давайте сначала обсудим роль обработчика события `Application_Error()`. Вспомните, что определенная страница может обрабатывать событие `Error` с целью перехвата любого необработанного исключения, которое произошло в области действия самой страницы. Вдобавок обработчик события `Application_Error()` — это последнее место, где можно обработать исключение, которое не было обработано страницей. Как и в случае события `Error` уровня страницы, обращаться к определенному исключению `System.Exception` можно с использованием унаследованного свойства `Server`:

```
void Application_Error(object sender, EventArgs e)
{
  // Получить необработанную ошибку.
  Exception ex = Server.GetLastError();
```

```
    // Обработать ошибку...

    // По завершении обработки очистить ошибку.
    Server.ClearError();
}
```

Поскольку обработчик событий `Application_Error()` является обработчиком исключений "последнего шанса" для веб-приложения, довольно часто этот метод реализуется так, что пользователь переносится на заранее определенную страницу ошибки на сервере. Другие распространенные действия могут включать отправку сообщения электронной почты веб-администратору или запись во внешний журнал ошибок.

Базовый класс `HttpApplication`

Как упоминалось ранее, сценарий `Global.asax` динамически генерируется в виде класса, производного от базового класса `System.Web.HttpApplication`, который предлагает часть того же рода функциональности, что и класс `System.Web.UI.Page` (без видимого пользовательского интерфейса). В табл. 33.2 документированы основные свойства класса `HttpApplication`.

Таблица 33.2. Основные свойства класса `System.Web.HttpApplication`

Свойство	Описание
`Application`	Это свойство позволяет взаимодействовать с данными уровня приложения посредством типа `HttpApplicationState`
`Request`	Это свойство позволяет взаимодействовать с входящим запросом HTTP с применением лежащего в основе объекта `HttpRequest`
`Response`	Это свойство позволяет взаимодействовать с исходящим ответом HTTP, используя лежащий в основе объект `HttpResponse`
`Server`	Это свойство позволяет получить встроенный серверный объект для текущего запроса с применением лежащего в основе объекта `HttpServerUtility`
`Session`	Это свойство позволяет взаимодействовать с данными уровня сеанса, используя лежащий в основе объект `HttpSessionState`

И снова, учитывая, что файл `Global.asax` явно не документирует `HttpApplication` в качестве лежащего в основе базового класса, важно помнить, что все правила отношения "является" на самом деле будут применены.

Разница между состоянием приложения и состоянием сеанса

В ASP.NET состояние приложения поддерживается экземпляром типа `HttpApplicationState`. Этот класс позволяет разделять глобальную информацию между всеми пользователями (и всеми страницами) приложения ASP.NET. Можно не только разделять все данные приложения между всеми пользователями сайта; если значение элемента данных уровня приложения изменяется, то новое значение становится видимым всем пользователям после того, как они выполнят следующую обратную отправку.

С другой стороны, состояние сеанса используется, чтобы запоминать информацию для специфического пользователя (подобную товарам в корзине покупок). Физически состояние сеанса пользователя представляется типом класса `HttpSessionState`.

Когда новый пользователь входит в веб-приложение ASP.NET, исполняющая среда автоматически назначает ему новый идентификатор сеанса, который по умолчанию устаревает после 20-минутного отсутствия активности. Таким образом, если на сайт вошло 20 000 пользователей, то существует 20 000 отдельных объектов `HttpSessionState`, каждому из которых автоматически назначен уникальный идентификатор сеанса. Отношения между веб-приложением и веб-сеансами демонстрируется на рис. 33.3.

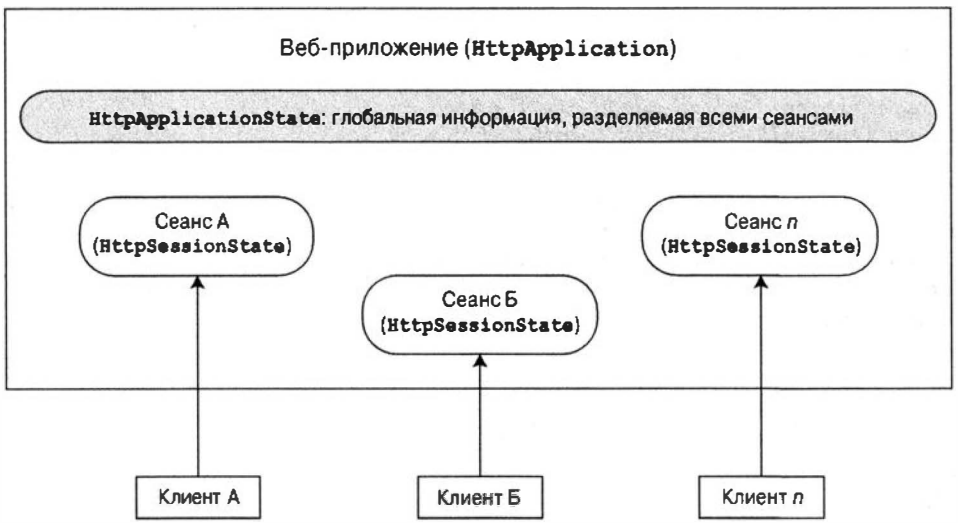


Рис. 33.3. Отличие между состоянием приложения и состоянием сеанса

Поддержка данных состояния уровня приложения

Тип `HttpApplicationState` позволяет разработчикам разделять глобальную информацию между множеством пользователей в приложении ASP.NET. В табл. 33.3 описаны некоторые основные члены этого типа.

Таблица 33.3. Члены типа `HttpApplicationState`

Член	Описание
`Add()`	Этот метод позволяет добавить в объект `HttpApplicationState` новую пару "имя-значение". Обратите внимание, что данный метод обычно *не* применяется, а вместо него используется индексатор класса `HttpApplicationState`
`AllKeys`	Это свойство возвращает массив объектов `string`, которые представляют все имена в объекте `HttpApplicationState`
`Clear()`	Этот метод очищает все элементы в объекте `HttpApplicationState`. Его функциональность эквивалентна методу `RemoveAll()`
`Count`	Это свойство получает количество элементов, хранимых в объекте `HttpApplicationState`
`Lock()`, `Unlock()`	Эти два метода применяются, когда необходимо изменить набор переменных приложения в безопасной к потокам манере
`RemoveAll()`, `Remove()`, `RemoveAt()`	Эти методы удаляют определенный элемент (по строковому имени) из объекта `HttpApplicationState`. Метод `RemoveAt()` удаляет элемент по числовому индексу

Чтобы проиллюстрировать работу с состоянием приложения, создайте проект пустого веб-сайта по имени AppState (и вставьте в него веб-форму). Затем добавьте новый файл Global.asax. Когда создаются данные-члены, которые могут разделяться между всеми пользователями, необходимо устанавливать пары "имя-значение". В большинстве случаев самым естественным местом для этого является обработчик события Application_Start() в файле Global.asax.cs:

```
void Application_Start(Object sender, EventArgs e)
{
  // Установить некоторые переменные приложения.
  Application["SalesPersonOfTheMonth"] = "Chucky";
  Application["CurrentCarOnSale"] = "Colt";
  Application["MostPopularColorOnLot"] = "Black";
}
```

На протяжении жизненного цикла веб-приложения (т.е. до тех пор, пока работа приложения не будет прекращена вручную либо не истечет тайм-аут последнего пользователя) любой пользователь на любой странице может обращаться к этим значениям, когда они ему нужны. Предположим, что есть страница, которая отображает текущую скидку на автомобиль в элементе Label посредством обработчика события Click кнопки:

```
protected void btnShowCarOnSale_Click(object sender, EventArgs arg)
{
  lblCurrCarOnSale.Text = string.Format("Sale on {0}'s today!",
    (string)Application["CurrentCarOnSale"]);
}
```

Как и в случае свойства ViewState, возвращаемое из объекта HttpApplicationState значение потребуется привести к корректному типу, потому что свойство Application оперирует с общими типами System.Object.

Поскольку свойство Application может хранить объект любого типа, само собой разумеется, что внутрь состояния приложения можно помещать объекты специальных типов (или любые объекты .NET). Предположим, что в строго типизированном классе CarLotInfo решено поддерживать три переменные приложения:

```
public class CarLotInfo
{
  public CarLotInfo(string salesPerson, string currentCar, string mostPopular)
  {
    SalesPersonOfTheMonth = salesPerson;
    CurrentCarOnSale = currentCar;
    MostPopularColorOnLot = mostPopular;
  }
  public string SalesPersonOfTheMonth { get; set; }
  public string CurrentCarOnSale { get; set; }
  public string MostPopularColorOnLot { get; set; }
}
```

Имея такой вспомогательный класс, обработчик события Application_Start() можно было бы модифицировать следующим образом:

```
void Application_Start(Object sender, EventArgs e)
{
  // Поместить специальный объект в раздел данных приложения.
  Application["CarSiteInfo"] =
    new CarLotInfo("Chucky", "Colt", "Black");
}
```

Затем можно было бы получать доступ к информации с использованием открытых полей данных внутри обработчика события Click серверной стороны для элемента управления Button по имени btnShowAppVariables:

```
protected void btnShowAppVariables_Click(object sender, EventArgs e)
{
  CarLotInfo appVars =
    ((CarLotInfo)Application["CarSiteInfo"]);
  string appState = $"<li>Car on sale: { appVars.CurrentCarOnSale }</li>"
  appState += $"<li>Most popular color: { appVars.MostPopularColorOnLot }</li>";
  appState += $"<li>Big shot SalesPerson: { appVars.SalesPersonOfTheMonth }</li>"
  lblAppVariables.Text = appState;
}
```

С учетом того, что текущие данные о продаже машины теперь доступны из специального класса, обработчик события Click кнопки btnShowCarOnSale также должен быть изменен:

```
protected void btnShowCarOnSale_Click(object sender, EventArgs e)
{
  lblCurrCarOnSale.Text =
    $"Sale on {((CarLotInfo)Application["CarSiteInfo"]).CurrentCarOnSale }'s
today!";
}
```

Модификация данных приложения

Во время выполнения веб-приложения можно программно обновлять либо удалять любые или вообще все элементы данных уровня приложения с применением членов типа HttpApplicationState. Например, чтобы удалить специфический элемент данных, просто вызовите метод Remove(). Если вы хотите уничтожить все данные уровня приложения, то вызовите RemoveAll().

```
private void CleanAppData()
{
  // Удалить один элемент по строковому имени.
  Application.Remove("SomeItemIDontNeed");

  // Уничтожить все данные приложения!
  Application.RemoveAll();
}
```

Если необходимо изменить значение существующего элемента данных уровня приложения, то понадобится только сделать новое присваивание интересующему элементу данных. Предположим, что страница теперь имеет элемент управления Button, который позволяет пользователю изменять текущего преуспевающего продавца, прочитав его имя из элемента TextBox по имени txtNewSP. Обработчик события Click выглядит вполне ожидаемо:

```
protected void btnSetNewSP_Click(object sender, EventArgs e)
{
  // Установить нового продавца.
  ((CarLotInfo)Application["CarSiteInfo"]).SalesPersonOfTheMonth
    = txtNewSP.Text;
}
```

Запустив веб-приложение, после щелчка на новой кнопке вы обнаружите, что элемент данных уровня приложения обновился. Более того, поскольку переменные приложения доступны всем пользователям на любой странице веб-приложения, после запус-

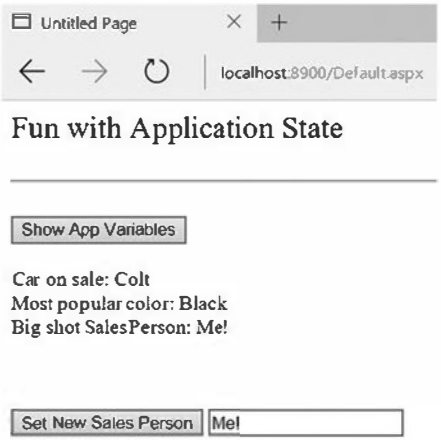


Рис. 33.4. Отображение данных приложения

ка трех или четырех экземпляров веб-браузера можно заметить, что при изменении текущего преуспевающего продавца в одном экземпляре все остальные экземпляры отобразят новое значение, как только произойдет обратная отправка. Возможный вывод показан на рис. 33.4.

Имейте в виду, что в ситуации, при которой набор переменных уровня приложения должен быть обновлен как единое целое, возникает риск повредить данные, т.к. формально возможно, что данные уровня приложения будут изменяться именно в тот момент, когда другой пользователь пытается получить к ним доступ. Вы могли бы избрать долгий путь и вручную реализовать блокировку с использованием потоковых примитивов из пространства имен `System.Threading`, но тип `HttpApplicatiobState` имеет два метода, `Lock()` и `Unlock()`, которые автоматически обеспечивают безопасность в отношении потоков:

```
// Безопасно обратиться к связанным данным приложения.
Application.Lock();
Application["SalesPersonOfTheMonth"] = "Maxine";
Application["CurrentBonusedEmployee"] = Application["SalesPersonOfTheMonth"];
Application.UnLock();
```

Обработка прекращения работы веб-приложения

Тип `HttpApplicationState` рассчитан на сохранение значений содержащихся в нем элементов до тех пор, пока не произойдет одна из двух ситуаций: у последнего пользователя сайта истечет время тайм-аута (или он выйдет явно) либо кто-то вручную прекратит работу сайта на веб-сервере IIS. В любом случае будет автоматически вызван метод `Application_End()` производного от `HttpApplication` типа. Внутрь этого обработчика событий можно поместить необходимый код очистки:

```
void Application_End(Object sender, EventArgs e)
{
  // Записать текущие переменные в базу данных или куда-нибудь еще.
}
```

Исходный код. Веб-сайт `AppState` доступен в подкаталоге `Chapter_33`.

Работа с кешем приложения

Инфраструктура ASP.NET предоставляет еще один более гибкий способ для обработки данных уровня приложения. Как вы помните, значения внутри объекта `HttpApplicationState` остаются в памяти до тех пор, пока веб-приложение активно и с ним производятся действия. Однако иногда может возникнуть необходимость в хранении фрагмента данных приложения в течение только определенного периода времени. Например, нужно получить ADO.NET-объект `DataSet`, который действителен на протяжении пяти минут. По истечении этого времени требуется получить обновленный объект `DataSet` для учета всех возможных изменений данных. Наряду с тем, что решить такую задачу формально возможно с применением объекта `HttpApplicationState` и

реализованного вручную мониторинга, все значительно упростится за счет использования кеша приложения ASP.NET.

Объект System.Web.Caching.Cache в ASP.NET (к которому можно обратиться через свойство Context.Cache) позволяет определить объекты, доступные всем пользователям на всех страницах в течение фиксированного периода времени. В простейшей форме взаимодействие с кешем выглядит идентичным взаимодействию с типом HttpApplicationState:

```
// Добавить элемент в кеш.
// Этот элемент *не* устареет.
Context.Cache["SomeStringItem"] = "This is the string item";

// Получить элемент из кеша.
string s = (string)Context.Cache["SomeStringItem"];
```

На заметку! Для доступа к кешу из Global.asax необходимо применять свойство Context. Тем не менее, внутри области определения типа, производного от System.Web.UI.Page, к объекту Cache можно обращаться напрямую через свойство Cache страницы.

Помимо индексатора в классе System.Web.Caching.Cache определено лишь небольшое количество членов. Метод Add() можно использовать для вставки в кеш нового элемента, который в текущий момент не определен (если указанный элемент уже присутствует, то метод Add() ничего не делает). Метод Insert() также помещает элемент в кеш, но если элемент уже определен, то текущий элемент заменяется новым. Поскольку именно такое поведение обычно ожидается, мы сосредоточим внимание исключительно на методе Insert().

Использование кеширования данных

Давайте рассмотрим пример. Создайте новый проект пустого веб-сайта по имени CacheState и добавьте в него веб-форму и файл Global.asax. Подобно любому элементу данных уровня приложения, поддерживаемому типом HttpApplicationState, кеш может хранить объекты производных от System.Object типов и часто заполняется внутри обработчика события Application_Start(). Целью настоящего примера будет автоматическое обновление содержимого DataSet каждые 15 секунд. Интересующий нас объект DataSet будет содержать текущий набор записей из таблицы Inventory базы данных AutoLot, созданной во время обсуждения ADO.NET.

С учетом сказанного выше установите ссылку на сборку AutoLotDAL.dll (см. главу 31; сборка также включена в состав загружаемого кода примеров для настоящей главы), добавьте к веб-сайту инфраструктуру Entity Framework (щелкнув правой кнопкой мыши на имени проекта и выбрав в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet)) и поместите в файл Web.config следующую строку подключения (в зависимости от установки SQL Server Express она может отличаться):

```
<connectionStrings>
  <add name="AutoLotConnection" connectionString="data source=(local)\
SQLEXPRESS2014;initial catalog=AutoLot;integrated security=True;
MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

Затем модифицируйте файл Global.asax, как показано ниже:

```
<%@ Application Language="C#" %>
<%@ Import Namespace = "AutoLotDAL.Repos" %>
```

```
<script runat="server">
  // Определить статическую переменную-член Cache.
  static Cache _theCache;

  void Application_Start(Object sender, EventArgs e)
  {
    // Присвоить значение статической переменной theCache.
    _theCache = Context.Cache;

    // При запуске приложения прочитать текущие записи
    // из таблицы Inventory базы данных AutoLot.
    var theCars = new InventoryRepo().GetAll();

    // Сохранить DataTable в кеше.
    _theCache.Insert("CarList",
      theCars,
      null,
      DateTime.Now.AddSeconds(15),
      Cache.NoSlidingExpiration,
      CacheItemPriority.Default,
      UpdateCarInventory);
  }

  // Целевой метод для делегата CacheItemRemovedCallback.
  static void UpdateCarInventory(string key, object item,
    CacheItemRemovedReason reason)
  {
    var theCars = new InventoryRepo().GetAll();

    // Сохранить в кеше.
    _theCache.Insert("CarList",
      theCars,
      null,
      DateTime.Now.AddSeconds(15),
      Cache.NoSlidingExpiration,
      CacheItemPriority.Default,
      UpdateCarInventory);
  }
</script>
```

Первым делом обратите внимание на определение статической переменной-члена `Cache`. Причина в том, что были определены два статических члена, которым необходим доступ к объекту `Cache`. Вспомните, что статические методы не имеют доступа к унаследованным членам, а потому применять свойство `Context` нельзя!

Внутри обработчика события `Application_Start()` сначала получается список записей `Inventory`, который затем помещается в кеш приложения. Как и можно было предположить, метод `Context.Cache.Insert()` имеет несколько перегруженных версий. Здесь указывается значение для каждого возможного параметра. Взгляните на следующий прокомментированный вызов `Insert()`:

```
_theCache.Insert("CarList",       // Имя для идентификации элемента в кеше.
  theCars,                        // Объект, помещаемый в кеш.
  null,                           // Есть ли зависимости у этого объекта?
  DateTime.Now.AddSeconds(15),    // Абсолютное время устаревания.
  Cache.NoSlidingExpiration, //Не использовать скользящее устаревание (см. ниже).
  CacheItemPriority.Default,      // Уровень приоритета элемента кеша.

  // Делегат для события CacheItemRemove.
  UpdateCarInventory);
```

Первые два параметра просто составляют пару "имя-значение" для элемента. Третий параметр позволяет определить объект `CacheDependency` (который в данном случае равен `null`, т.к. реализация `IList<Inventory>` ни от чего не зависит).

Параметр `DateTime.Now.AddSeconds(15)` задает абсолютное время устаревания. Это значит, что элемент гарантированно будет удален из кеша через 15 секунд. Абсолютное время устаревания удобно для элементов данных, которые нуждаются в постоянном обновлении (вроде биржевых показателей).

Параметр `Cache.NoSlidingExpiration` указывает, что элемент кеша не применяет скользящее устаревание. Скользящее устаревание представляет собой способ сохранения элемента в кеше на протяжении минимум определенного периода времени. Например, если установить значение скользящего устаревания в 60 секунд для элемента кеша, то он будет находиться там, по меньшей мере, 60 секунд. Если в течение этого времени любая веб-страница обращается к элементу кеша, то таймер сбрасывается, и существование элемента кеша продлевается еще на 60 секунд. Если же в заданный промежуток времени обращения отсутствуют, то элемент из кеша удаляется. Скользящее устаревание удобно для данных, которые требуют высоких затрат (времени) на генерацию, но не слишком часто используются веб-страницами.

Обратите внимание, что для отдельного элемента кеша указывать одновременно абсолютное устаревание и скользящее устаревание не допускается. Должно быть установлено либо абсолютное устаревание (с помощью `Cache.NoSlidingExpiration`), либо скользящее устаревание (посредством `Cache.NoAbsoluteExpiration`).

Наконец, как видно из сигнатуры метода `UpdateCarInventory()`, делегат `CacheItemRemovedCallback` может вызывать только методы с такой сигнатурой:

```
void UpdateCarInventory(string key, object
item,
     CacheItemRemovedReason reason)
{
}
```

Итак, теперь при запуске приложения объект `DataTable` заполняется и кешируется. Каждые 15 секунд объект `DataTable` очищается, обновляется и заново вставляется в кеш. Чтобы увидеть это в действии, понадобится создать страницу, которая позволит осуществлять некоторое взаимодействие с пользователем.

Модификация файла *.aspx

На рис. 33.5 показан пользовательский интерфейс, который позволяет вводить необходимые данные для вставки новой записи в базу данных (посредством трех элементов управления `TextBox`). Обработчик события `Click` для единственного элемента управления `Button` будет закодирован с целью поддержки манипуляций в базе данных. Наконец, вдобавок к описательным элементам `Label` элемент управления `GridView` в нижней части страницы будет применяться для отображения набора текущих записей в таблице `Inventory`.

CarId	Make	Color	PetName
1	VW1	Black	Zippy
2	Ford	Rust	Rusty
3	Saab	Black	Mel
4	Yugo	Yellow	Clunker
5	BMW	Black	Bimmer
6	BMW	Green	Hank
7	BMW	Pink	Pinky
13	Pinto	Black	Pete
45	BMW	Blue	Blueberry

Рис. 33.5. Графический пользовательский интерфейс приложения, работающего с кешем

В обработчике события Load страницы сконфигурируйте элемент управления GridView для отображения текущих кешированных данных при первом получении страницы пользователем (не забудьте импортировать в файл кода пространства имен AutoLotDAL.Models и AutoLotDAL.Repos):

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
  {
    carsGridView.DataSource = (IList<Inventory>)Cache["AppDataTable"];
    carsGridView.DataBind();
  }
}
```

В обработчике события Click кнопки Add This Car (Добавить этот автомобиль) вставьте новую запись в базу данных AutoLot, используя тип InventoryRepo. После того как запись будет вставлена, вызовите вспомогательный метод по имени RefreshGrid(), который обновит пользовательский интерфейс:

```
protected void btnAddCar_Click(object sender, EventArgs e)
{
  // Обновить таблицу Inventory и вызвать RefreshGrid().
  new InventoryRepo().Add(new Inventory()
  {
    Color = txtCarColor.Text,
    Make = txtCarMake.Text,
    PetName = txtCarPetName.Text
  });
  RefreshGrid();
}

private void RefreshGrid()
{
  carsGridView.DataSource = new InventoryRepo().GetAll();
  carsGridView.DataBind();
}
```

Чтобы проверить работу кеша, начните с запуска текущей программы (нажав <Ctrl+F5>) и скопируйте URL, появившийся в браузере, в буфер обмена. Запустите второй экземпляр браузера и вставьте скопированный URL в его строку адреса. Вы должны видеть на экране два экземпляра браузера с загруженной страницей Default.aspx, отображающей идентичные данные.

В первом экземпляре браузера добавьте новую запись об автомобиле. Очевидно, что в результате получится обновленный элемент управления GridView, отображаемый в браузере, который инициировал обратную отправку.

Во втором экземпляре браузера щелкните на кнопке обновления (или нажмите <F5>). Вы не сразу должны увидеть новую запись об автомобиле, т.к. обработчик события Page_Load читает напрямую из кеша. (Если вы видите новую запись, то 15 секунд истекли. Либо действуйте быстрее, либо увеличьте период времени, в течение которого DataTable остается в кеше.) Подождите несколько секунд и еще раз щелкните на кнопке обновления во втором экземпляре браузера. Теперь вы должны увидеть новый элемент, поскольку данные в кеше устарели, и целевой метод делегата CacheItemRemoveCallback автоматически обновил кешированные данные.

Итак, главное преимущество типа Cache заключается в том, что появляется шанс отреагировать на удаление элемента из кеша. В приведенном примере определенно можно было бы избежать применения Cache и просто заставить обработчик события

Page_Load всегда читать прямо из базы данных AutoLot (но тогда работа страницы существенно замедлилась бы). Тем не менее, идея должна быть ясна: кеш позволяет автоматически обновлять данные, используя механизм кеширования.

Исходный код. Веб-сайт CacheState доступен в подкаталоге Chapter_33.

Поддержка данных сеанса

На этом исследование данных уровня приложения и кешированных данных завершено. Теперь давайте выясним роль данных, специфичных для пользователя. Как уже упоминалось, *сеанс* — это просто взаимодействие конкретного пользователя с веб-приложением, которое представлено уникальным объектом HttpSessionState. Чтобы поддерживать информацию о состоянии для конкретного пользователя, можно применять свойство Session в классе веб-страницы или в файле Global.asax. Классическим примером необходимости в поддержке данных пользователя является корзина покупок. Если 10 человек зашли в электронный магазин, то каждый из них будет иметь уникальный набор наименований товаров, который он намерен приобрести, и эти данные должны поддерживаться.

Когда новый пользователь входит в веб-приложение, исполняющая среда .NET автоматически назначает ему уникальный идентификатор сеанса, служащий для идентификации данного пользователя. С каждым идентификатором сеанса связан специальный экземпляр типа HttpSessionState, в котором хранятся специфичные для пользователя данные. Вставка или извлечение данных сеанса синтаксически идентично манипулированию данными приложения, например:

```
// Добавить/извлечь данные сеанса для текущего пользователя.
Session["DesiredCarColor"] = "Green";
string color = (string) Session["DesiredCarColor"];
```

В файле Global.asax можно перехватывать момент начала и конца сеанса посредством обработчиков событий Session_Start() и Session_End(). Внутри Session_Start() допускается свободно создавать любые элементы данных, специфичные для пользователя, в то время как Session_End() позволяет выполнять любую работу, которая может требоваться при завершении пользовательского сеанса.

```
<%@ Application Language="C#" %>
...
void Session_Start(Object sender, EventArgs e)
{
  // Новый сеанс! При необходимости выполнить подготовительные действия.
}

void Session_End(Object sender, EventArgs e)
{
  // Пользователь вышел или отключен по тайм-ауту.
  // При необходимости выполнить очистку.
}
```

Подобно состоянию приложения состояние сеанса может хранить объекты любых производных от System.Object типов, включая специальные классы. Например, предположим, что создан новый проект пустого веб-сайта (по имени SessionState), в котором определен класс UserShoppingCart:

```
public class UserShoppingCart
{
  public string DesiredCar {get; set;}
```

```
  public string DesiredCarColor {get; set;}
  public float DownPayment {get; set;}
  public bool IsLeasing {get; set;}
  public DateTime DateOfPickUp {get; set;}

  public override string ToString() =>
   $"Car: {DesiredCar}<br>Color: {DesiredCarColor}<br>$ Down: {DownPayment}" +
   $"<br>Lease: {IsLeasing}<br>Pick-up Date: {DateOfPickUp.ToShortDateString()}";
}
```

Вставьте в проект файл `Global.asax`. **Внутри обработчика** `Session_Start()` **теперь каждому пользователю можно назначить новый экземпляр класса** `UserShoppingCart`:

```
void Session_Start(Object sender, EventArgs e)
{
  Session["UserShoppingCartInfo"] = new UserShoppingCart();
}
```

Во время посещения пользователем веб-страниц можно брать экземпляр `UserShoppingCart` **и заполнять его поля специфичными для пользователя данными. Предположим, что есть простая страница** `*.aspx`, **которая определяет набор элементов управления для ввода, соответствующих полям типа** `UserShoppingCart`, **элемент управления** `Button`, **используемый для установки значений, и два элемента** `Label`, **предназначенные для отображения идентификатора сеанса пользователя и информации о сеансе (рис. 33.6).**

Обработчик события `Click` **серверной стороны для элемента управления** `Button` **довольно прост (он извлекает значения из элементов** `TextBox` **и отображает данные корзины покупок в элементе** `Label`):

```
protected void btnSubmit_Click(
  object sender, EventArgs e)
{
  // Установить предпочтения
  // для текущего пользователя.
  var cart = (UserShoppingCart)Session[
  "UserShoppingCartInfo"];
  cart.DateOfPickUp =
    myCalendar.SelectedDate;
  cart.DesiredCar = txtCarMake.Text;
  cart.DesiredCarColor = txtCarColor.Text;
  cart.DownPayment =
    float.Parse(txtDownPayment.Text);
  cart.IsLeasing = chkIsLeasing.Checked;
  lblUserInfo.Text = cart.ToString();
  Session["UserShoppingCartInfo"] = cart;
}
```

Рис. 33.6. Графический пользовательский интерфейс приложения, демонстрирующего работу с сеансами

Внутри метода `Session_End()` **можно сохранить поля** `UserShoppingCart` **в базе данных или где-нибудь еще (как будет показано в конце главы, API-интерфейс Profile делает это автоматически). Кроме того, можно реализовать метод** `Session_Error()` **для перехва-**

та любого ошибочного ввода (или задействовать разнообразные элементы управления проверкой достоверности на странице Default.aspx для обработки ошибок такого рода).

Запустив два или три экземпляра браузера с одним и тем же URL, вы обнаружите, что каждый пользователь может наполнять собственную корзину покупок, которая отображается на его уникальный экземпляр класса HttpSessionState.

Дополнительные члены класса HttpSessionState

Помимо индексатора в классе HttpSessionState определены другие интересные члены. Свойство SessionID возвращает уникальный идентификатор текущего пользователя. Если хотите увидеть в этом примере автоматически назначенный идентификатор сеанса, то обработайте событие Load страницы, как показано ниже:

```
protected void Page_Load(object sender, EventArgs e)
{
  lblUserID.Text = $"Here is your ID: { Session.SessionID }";
}
```

Методы Remove() и RemoveAll() могут применяться для очистки элементов экземпляра HttpSessionState, связанного с пользователем:

```
Session.Remove("SomeItemWeDontNeedAnymore");
```

В классе HttpSessionState также определен набор членов, которые управляют политикой устаревания текущего сеанса. По умолчанию каждый пользователь располагает 20 минутами отсутствия активности, прежде чем объект HttpSessionState будет уничтожен. Таким образом, если пользователь входит в веб-приложение (и, следовательно, получает уникальный идентификатор сеанса), но на протяжении 20 минут не проявляет активности, то исполняющая среда предполагает, что пользователь больше не заинтересован в сайте и уничтожает его данные сеанса. Период устаревания сеанса можно изменить, установив его для каждого пользователя индивидуально с помощью свойства Timeout. Наиболее подходящим местом для этого является метод Session_Start():

```
void Session_Start(Object sender, EventArgs e)
{
  // Для каждого пользователя установить 5-минутный период отсутствия активности.
  Session.Timeout = 5;
  Session["UserShoppingCartInfo"]
    = new UserShoppingCart();
}
```

На заметку! Если настраивать значение Timeout для каждого пользователя не требуется, то можно изменить 20-минутное стандартное значение для всех пользователей через атрибут timeout элемента <sessionState> внутри файла Web.config (рассматривается в конце главы).

Преимущество свойства Timeout связано с возможностью установки значения тайм-аута для каждого пользователя по отдельности. Предположим, что создано веб-приложение, которое позволяет пользователям платить дифференцированную плату за различные уровни членства. Вы могли бы указать, что привилегированные пользователи должны иметь тайм-аут длительностью в один час, тогда как обычные — только 30 секунд. Такая возможность вызывает вопрос: как запомнить специфическую для пользователя информацию (вроде текущего уровня членства) между визитами на сайт? Один из вариантов предусматривает использование типа HttpCookie.

Исходный код. Веб-сайт SessionState доступен в подкаталоге Chapter_33.

Cookie-наборы

Следующий прием управления состоянием предусматривает сохранение данных внутри *cookie-набора*, который часто реализуется как текстовый файл (или набор файлов) на машине пользователя. Когда пользователь входит на определенный сайт, браузер проверяет наличие на пользовательской машине cookie-файла для конкретного URL, и если файл присутствует, то добавляет его данные к запросу HTTP.

Принимающая веб-страница серверной стороны может затем прочитать cookie-данные для создания графического интерфейса на основе предпочтений текущего пользователя. Наверняка вы замечали, что после посещения одного из любимых веб-сайтов он каким-то образом "знает" разновидность содержимого, которое вы предпочитаете просматривать. Причина (отчасти) в том, что в cookie-наборах, хранящихся на вашем компьютере, содержится информация, которая касается этого веб-сайта.

На заметку! Точное местоположение cookie-файлов зависит от применяемого браузера и установленной операционной системы.

Вполне очевидно, что содержимое cookie-файлов будет варьироваться от веб-сайта к веб-сайту, но имейте в виду, что в конечном итоге они все равно являются текстовыми файлами. Таким образом, cookie-наборы — наихудший выбор для хранения конфиденциальной информации о текущем пользователе (такой как номер кредитной карточки, пароль и т.п.). Даже если вы зашифруете данные, то все равно есть шанс, что настойчивый хакер расшифрует их и воспользуется в злоумышленных целях. Но, так или иначе, cookie-наборы играют определенную роль в разработке веб-приложений, поэтому давайте выясним, как ASP.NET поддерживает данный прием управления состоянием.

Создание cookie-наборов

Прежде всего, cookie-наборы ASP.NET могут быть сконфигурированы как постоянные или временные. *Постоянный* cookie-набор обычно рассматривается как классическое определение cookie-данных в том, что набор пар "имя-значение" физически сохраняется на жестком диске пользователя. *Временный* cookie-набор (также называемый *сеансовым* cookie-набором) содержит те же данные, что и постоянный cookie-набор, но пары "имя-значение" никогда не сохраняются на жестком диске пользователя, а существуют *только* в течение периода, пока открыт браузер. Как только пользователь закрывает браузер, все данные, содержащиеся в сеансовом cookie-наборе, уничтожаются.

Рис. 33.7. Пользовательский интерфейс приложения `CookieStateApp`

Класс `System.Web.HttpCookie` представляет серверную сторону cookie-данных (постоянных или временных). Для создания нового cookie-набора в коде веб-страницы необходимо обратиться к свойству `Response.Cookies`. Как только новый элемент `HttpCookie` вставлен во внутреннюю коллекцию, пары "имя-значение" отправляются браузеру внутри заголовка HTTP.

Чтобы наглядно увидеть поведение cookie-наборов, создайте новый проект пустого веб-сайта по имени `CookieStateApp` и добавьте в него веб-форму с пользовательским интерфейсом, показанным на рис. 33.7.

Внутри обработчика события `Click` кнопки Write This Cookie (Записать этот cookie-набор)

создайте новый объект HttpCookie и вставьте его в коллекцию Cookie, доступную через свойство HttpRequest.Cookies. Имейте в виду, что данные не будут сохраняться на жестком диске пользователя, если только вы явно не установите срок хранения в свойстве HttpCookie.Expires. Таким образом, следующая реализация создаст временный cookie-набор, который будет уничтожен, когда пользователь закроет браузер:

```
protected void btnCookie_Click(object sender, EventArgs e)
{
  // Создать временный cookie-набор.
  HttpCookie theCookie = new HttpCookie(txtCookieName.Text,txtCookieValue.Text);
  Response.Cookies.Add(theCookie);
}
```

Однако приведенный ниже код сгенерирует постоянный cookie-набор, который будет действителен в течение трех месяцев, начиная с текущей даты:

```
protected void btnCookie_Click(object sender, EventArgs e)
{
  HttpCookie theCookie = new HttpCookie(txtCookieName.Text,txtCookieValue.Text);
  theCookie.Expires = DateTime.Now.AddMonths(3);
  Response.Cookies.Add(theCookie);
}
```

Чтение входящих cookie-данных

Вспомните, что браузер является той сущностью, которая отвечает за доступ к постоянным cookie-наборам при переходе на ранее посещенную страницу. Если браузер решает отправить cookie-набор серверу, то доступ к входящим cookie-данным в коде страницы *.aspx можно получить через свойство HttpRequest.Cookies. В целях иллюстрации реализуйте обработчик события Click для кнопки Show Cookie Data (Показать cookie-данные):

```
protected void btnShowCookie_
Click(object sender, EventArgs e)
{
  string cookieData = "";
  foreach (string s in Request.Cookies)
  {
    cookieData +=
     $"<li><b>Name</b>: {s}, <b>Value</b>: { Request.Cookies[s]?.Value }</li>";
  }
  lblCookieData.Text = cookieData;
}
```

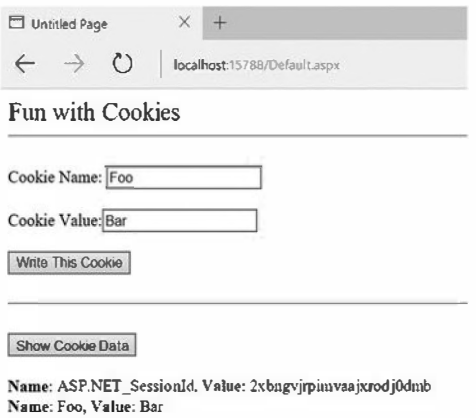


Рис. 33.8. Выполнение приложения CookieStateApp

Запустив приложение и щелкнув на кнопке Show Cookie Data, вы увидите, что cookie-данные действительно были оправлены браузером и успешно доступны в коде *.aspx на стороне сервера (рис. 33.8).

Исходный код. Веб-сайт CookieStateApp доступен в подкаталоге Chapter_33.

Роль элемента <sessionState>

К настоящему моменту вы ознакомились с многочисленными способами запоминания информации о пользователях. Вы видели, что манипулирование состоянием представления и данными приложения, кеша, сеанса и cookie-набора в коде осуществляется более или менее сходным образом (посредством индексатора класса). Вы также узнали, что в файле `Global.asax` имеются методы, которые позволяют перехватывать и реагировать на события, возникающие на протяжении времени жизни веб-приложения.

По умолчанию ASP.NET будет хранить состояние сеанса внутри процесса. Положительной стороной является максимально быстрый доступ к информации. Тем не менее, отрицательная сторона заключается в том, что если произойдет аварийный отказ этого домена приложения (по любой причине), то все данные состояния пользователя разрушатся. Более того, когда данные состояния хранятся во внутрипроцессной сборке `*.dll`, нет возможности взаимодействовать с сетевой веб-фермой. Такой стандартный режим хранения работает достаточно хорошо, если веб-приложение размещено на единственном веб-сервере. Однако, как и можно было предположить, эта модель не идеальна для фермы веб-серверов, т.к. состояние сеанса "замкнуто" внутри определенного домена приложения.

Хранение данных сеанса на сервере состояния сеансов ASP.NET

В ASP.NET исполняющую среду можно инструктировать о необходимости размещения сборки `*.dll` состояния сеанса в суррогатном процессе, который называется сервером состояния сеансов ASP.NET (`aspnet_state.exe`). В таком случае сборку `*.dll` можно выгрузить из `aspnet_wp.exe` в отдельный файл `*.exe`, который может располагаться на любой машине внутри веб-фермы. Даже если вы намерены запускать процесс `aspnet_wp.exe` на той же машине, что и веб-сервер, вы все равно получите выигрыш от вынесения данных состояния в отдельный процесс (т.к. это более надежно).

Чтобы задействовать сервер состояния сеансов, первым делом запустите Windows-службу `aspnet_state.exe` на целевой машине посредством ввода следующей команды в командной строке разработчика Visual Studio (обратите внимание, что для этого понадобятся права администратора):

```
net start aspnet_state
```

В качестве альтернативы Windows-службу `aspnet_state.exe` можно запустить через значок Services (Службы) в группе Administrative Tools (Администрирование) панели управления (рис. 33.9).

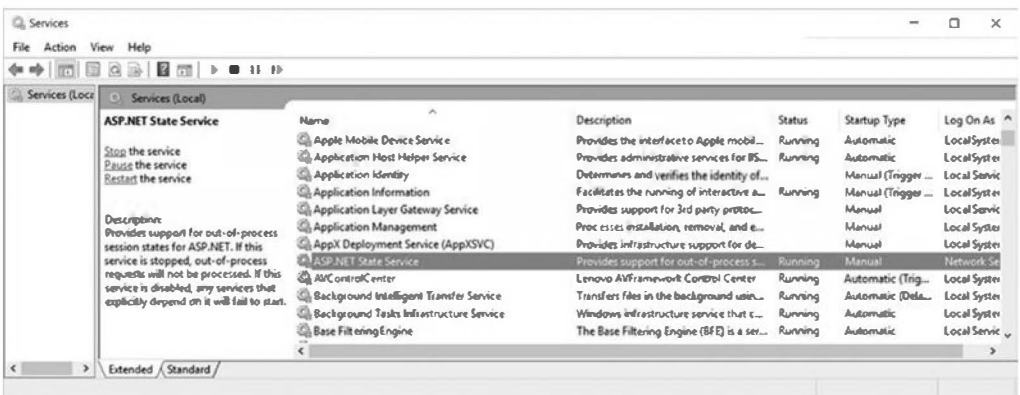


Рис. 33.9. Запуск Windows-службы `aspnet_state.exe` через значок Services

Основное преимущество такого подхода связано с возможностью конфигурирования службы `aspnet_state.exe` в окне ее свойств на автоматический запуск при начальной загрузке операционной системы на машине. После того, как сервер состояния сеансов запущен, добавьте в файл `Web.config` следующий элемент `<sessionState>`:

```
<system.web>
  <sessionState
    mode="StateServer"
    stateConnectionString="tcpip=127.0.0.1:42626"
    sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
    cookieless="false"
    timeout="20"
  />
  ...
</system.web>
```

Вот и все! Начиная с этого момента, среда CLR будет хранить данные, относящиеся к сеансу, внутри процесса `aspnet_state.exe`. Если произойдет аварийный отказ домена приложения, в котором размещено веб-приложение, то данные сеанса останутся в целости. Вдобавок обратите внимание, что элемент `<sessionState>` может также поддерживать атрибут `stateConnectionString`. Стандартное значение адреса TCP/IP (`127.0.0.1`) указывает на локальную машину. Если вы хотите, чтобы исполняющая среда .NET использовала службу `aspnet_state.exe`, функционирующую на другой машине (т.е. случай веб-фермы), то можете соответствующим образом изменить адрес TCP/IP.

Хранение информации о сеансах в выделенной базе данных

Наконец, если для веб-приложения требуется самая высокая степень изоляции и надежности, то можно заставить исполняющую среду хранить все данные о состоянии сеанса внутри Microsoft SQL Server. Необходимое для этого изменение файла `Web.config` выглядит очень просто:

```
<sessionState
  mode="SQLServer"
  stateConnectionString="tcpip=127.0.0.1:42626"
  sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
  cookieless="false"
  timeout="20" />
```

Тем не менее, прежде чем попробовать запустить ассоциированное веб-приложение, следует удостовериться, что целевая машина (указанная в атрибуте `sqlConnectionString`) подходящим образом сконфигурирована. При установке .NET Framework 4.6 SDK (или Visual Studio соответствующей версии) предоставляются два файла с именами `InstallSqlState.sql` и `UninstallSqlState.sql`, по умолчанию находящиеся в каталоге `C:\Windows\Microsoft.NET\Framework\<версия>`. На целевой машине потребуется запустить файл `InstallSqlState.sql` с применением инструмента наподобие Microsoft SQL Server Management Studio (который поставляется вместе с Microsoft SQL Server).

После выполнения SQL-сценария `InstallSqlState.sql` вы обнаружите новую базу данных SQL Server (`ASPState`), которая содержит несколько хранимых процедур, вызываемых исполняющей средой ASP.NET, и набор таблиц, используемых для хранения данных сеансов. (Кроме того, база данных `tempdb` обновляется набором таблиц, предназначенных для подкачки.) Как и можно было предположить, конфигурация веб-приложения с хранением сеансовых данных в SQL Server является самой медленной из всех возможных вариантов. Преимущество связано с тем, что пользовательские данные хранятся надежнее всего (они не утрачиваются, даже когда происходит перезагрузка веб-сервера).

На заметку! Если для хранения сеансовых данных применяется сервер состояния сеансов ASP.NET или SQL Server, то вы должны удостовериться, что все специальные типы, помещенные в объект `HttpSessionState`, помечены атрибутом `[Serializable]`.

Введение в API-интерфейс ASP.NET Profile

До сих пор вы исследовали различные приемы, которые позволяли запоминать данные уровня пользователей и уровня приложения. Однако многие веб-сайты требуют возможности сохранения пользовательской информации между сеансами. Например, предположим, что нужно предоставить пользователям функциональность создания учетных записей на сайте. Или, может быть, необходимо сохранять экземпляры `ShoppingCart` между сеансами (для сайта онлайнового магазина). Либо интересует, скажем, сохранение базовых пользовательских предпочтений (темы и т.д.).

Хотя для хранения такой информации можно было бы построить специальную базу данных (с несколькими хранимыми процедурами), впоследствии пришлось бы создавать специальную библиотеку кода для взаимодействия с объектами базы данных. Задача не обязательно будет сложной, но суть в том, что именно *вы* отвечаете за построение инфраструктуры подобного рода.

Чтобы помочь справиться с такими ситуациями, в состав ASP.NET входит готовый API-интерфейс управления профилями пользователей (ASP.NET Profile API) и система базы данных для этой конкретной цели. В дополнение к обеспечению необходимой инфраструктуры API-интерфейс Profile позволяет определять подлежащие сохранению данные прямо внутри файла `Web.config` (для упрощения); тем не менее, можно также сохранять любой тип, помеченный атрибутом `[Serializable]`. Перед тем как погрузиться в эту тему, давайте посмотрим, где API-интерфейс Profile будет хранить указанные данные.

База данных `ASPNETDB.mdf`

Каждый веб-сайт ASP.NET, построенный с помощью Visual Studio, поддерживает папку `App_Data`. По умолчанию API-интерфейс Profile (а также другие службы вроде API-интерфейса членства в ролях ASP.NET, который здесь не рассматривается) конфигурируется на работу с локальной базой данных SQL Server по имени `ASPNETDB.mdf`, расположенной в папке `App_Data`. Это стандартное поведение определяется настройками в файле `machine.config` для текущей установки платформы .NET на машине. В действительности, когда код задействует службу ASP.NET, требующую наличия папки `App_Data`, файл `ASPNETDB.mdf` будет автоматически создан, если он еще не существует.

Если взамен нужно, чтобы исполняющая среда ASP.NET взаимодействовала с файлом `ASPNETDB.mdf`, находящимся на другой машине в сети, или предпочтительнее установить эту базу данных на экземпляре SQL Server 7.0 (либо последующей версии), то файл `ASPNETDB.mdf` придется создать вручную с использованием утилиты командной строки `aspnet_regsql.exe`. Подобно любой хорошей утилите командной строки `aspnet_regsql.exe` поддерживает многочисленные аргументы, но если запустить ее без аргументов (в окне командной строки разработчика):

```
aspnet_regsql
```

то откроется мастер с графическим пользовательским интерфейсом, который проведет по всему процессу создания и установки базы данных `ASPNETDB.mdf` на выбранной машине (и версии SQL Server).

Предполагая, что сайт не работает с локальной копией базы данных в папке App_Data, финальный шаг заключается в модификации файла Web.config с целью указания уникального местоположения файла ASPNETDB.mdf. Пусть файл ASPNETDB.mdf установлен на машине по имени ProductionServer. Приведенное ниже неполное содержимое файла machine.config будет инструктировать API-интерфейс Profile о том, где по умолчанию расположены необходимые элементы базы данных (для изменения этих стандартных настроек можно добавить специальный файл Web.config):

```
<configuration>
  <connectionStrings>
    <add name="LocalSqlServer"
         connectionString ="Data Source=ProductionServer;Integrated
         Security=SSPI;Initial Catalog=aspnetdb;"
         providerName="System.Data.SqlClient"/>
  </connectionStrings>
  <system.web>
    <profile>
      <providers>
        <clear/>
        <add name="AspNetSqlProfileProvider"
             connectionStringName="LocalSqlServer"
             applicationName="/"
             type="System.Web.Profile.SqlProfileProvider, System.Web,
             Version=4.0.0.0,
             Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
      </providers>
    </profile>
  </system.web>
</configuration>
```

Подобно большинству файлов *.config содержимое выглядит намного сложнее, чем есть на самом деле. По существу здесь определяется элемент <connectionString> с необходимыми данными, за которым следует именованный экземпляр SqlProfileProvider (стандартный поставщик, применяемый независимо от физического местоположения ASPNETDB.mdf).

На заметку! Для простоты предположим, что будет использоваться автоматически сгенерированная база данных ASPNETDB.mdf, расположенная в подкаталоге App_Data веб-приложения.

Определение пользовательского профиля в файле Web.config

Как уже упоминалось, пользовательский профиль определяется внутри файла Web.config. По-настоящему замечательный аспект этого подхода заключается в том, что с профилем можно взаимодействовать в строго типизированной манере, применяя в файлах кода унаследованное свойство Property. В качестве примера создайте новый проект пустого веб-сайта по имени FunWithProfiles, добавьте в него новый файл *.aspx и откройте файл Web.config для редактирования.

Нашей целью будет создание профиля, который моделирует домашние адреса пользователей, находящихся в сеансе, а также общее количество их посещений веб-сайта. Данные профиля вполне предсказуемо определяются внутри элемента <profile> с использованием набора пар "имя-значение". Взгляните на следующий профиль, который создан внутри элемента <system.web>:

```
<profile>
  <properties>
   <add name="StreetAddress" type="System.String" />
   <add name="City" type="System.String" />
   <add name="State" type="System.String" />
   <add name="TotalPost" type="System.Int32" />
  </properties>
</profile>
```

Здесь для каждого элемента профиля указано имя и тип данных CLR (разумеется, можно было бы добавить дополнительные элементы для почтового кода, имени и т.д., но и без них идея должна быть ясна). Строго говоря, атрибут `type` не является обязательным и по умолчанию для него принимается тип `System.String`. Существует много других атрибутов, которые могут быть указаны в элементе профиля для дальнейшего уточнения способа хранения данной информации в `ASPNETDB.mdf`. В табл. 33.4 кратко описаны избранные атрибуты.

Таблица 33.4. Избранные атрибуты данных профиля

Атрибут	Примеры значений	Описание
`allowAnonymous`	`true` | `false`	Ограничивает или разрешает анонимный доступ к данному значению. Если установлен в `false`, то анонимные пользователи не будут иметь доступ к этому значению профиля
`defaultValue`	Строка	Значение, которое должно быть возвращено, если свойство не было явно установлено
`name`	Строка	Уникальный идентификатор для данного свойства
`provider`	Строка	Поставщик, применяемый для управления этим значением. Переопределяет настройку `defaultProvider` в файле `Web.config` или `machine.config`
`readOnly`	`true` | `false`	Ограничивает или разрешает доступ для записи. Стандартным значением является `false` (т.е. допускается не только чтение)
`serializeAs`	`String` | `XML` | `Binary`	Формат значения при записи в хранилище данных
`type`	Элементарный тип | Тип, определяемый пользователем	Элементарный тип или класс .NET. Имена классов должны быть полностью заданными (например, `MyApp.UserData.ColorPrefs`)

Вы увидите некоторые из этих атрибутов в действии при модификации текущего профиля. А пока давайте посмотрим, как программно обращаться к этим данным в коде страниц.

Доступ к данным профиля в коде

Вспомните, что общее предназначение API-интерфейса ASP.NET Profile заключается в автоматизации процесса записи данных в выделенную базу данных (и чтения из нее). Чтобы попрактиковаться, измените пользовательский интерфейс страницы `Default.aspx`, добавив набор элементов управления `TextBox` (и описательных элементов `Label`) для ввода адреса пользователя — улицы, города и штата. Кроме того, добавьте элемент управления `Button` (по имени `btnSubmit`) и еще один элемент `Label` (с именем `lblUserData`) для отображения сохраненных данных (рис. 33.10).

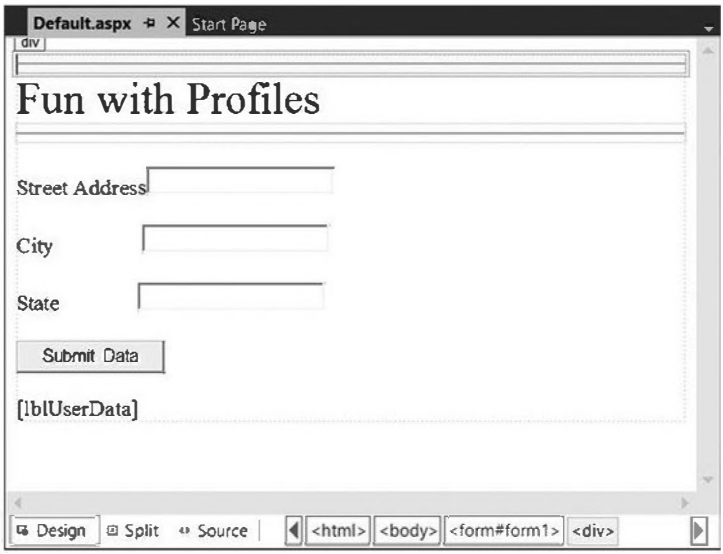


Рис. 33.10. Пользовательский интерфейс страницы `Default.aspx` веб-сайта `FunWithProfiles`

Внутри обработчика события `Click` кнопки Submit Data (Отправить данные) посредством унаследованного свойства `Profile` сохраните каждую единицу данных профиля на основе информации, введенной пользователем в соответствующем поле `TextBox`. После сохранения всех элементов данных в `ASPNETDB.mdf` прочитайте их из базы данных и сформируйте строку для отображения в элементе `Label` по имени `lblUserData`. Наконец, обработайте событие `Load` страницы и отобразите ту же информацию в элементе `Label`. Таким образом, когда пользователь зайдет на страницу, он увидит текущие настройки. Ниже приведен полный код:

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    GetUserAddress();
  }
  protected void btnSubmit_Click(object sender, EventArgs e)
  {
    // Здесь происходит запись в базу данных.
    Profile.StreetAddress = txtStreetAddress.Text;
    Profile.City = txtCity.Text;
    Profile.State = txtState.Text;

    // Получить настройки из базы данных.
    GetUserAddress();
  }

  private void GetUserAddress()
  {
    // Здесь происходит чтение из базы данных.
    lblUserData.Text =
      $"You live here: {Profile. StreetAddress }, {Profile.City},
                       {Profile.State}";
  }
}
```

Запустив приложение, вы заметите длительную задержку при первом запросе Default.aspx. Причина связана с созданием на лету файла ASPNETDB.mdf в папке App_Data (в этом можно убедиться, обновив окно Solution Explorer и заглянув в папку App_Data).

Вы также обнаружите, что при первом посещении страницы элемент Label по имени lblUserData не отображает никаких данных профиля, потому что они еще не были добавлены в подходящую таблицу базы данных ASPNETDB.mdf. После ввода значений в полях TextBox и обратной отправки страницы серверу в элементе lblUserData отобразятся сохраненные данные.

Теперь перейдем к действительно интересному аспекту этой технологии. Если вы закроете браузер и перезапустите веб-сайт, то обнаружите, что ранее введенные данные профиля на самом деле сохранились, т.к. в Label отображается корректная информация. Возникает очевидный вопрос: как они были сохранены?

В рассматриваемом примере API-интерфейс Profile использует сетевую идентичность Windows, которая получена из текущих учетных данных машины. Однако при построении публично доступных веб-сайтов (где пользователи не относятся к какому-то домену) понадобится обеспечить интеграцию API-интерфейса Profile с моделью аутентификации ASP.NET на основе форм, а также поддержку понятия "анонимных профилей", которые позволяют хранить данные профиля для пользователей, не имеющих в данный момент активной идентичности на сайте.

На заметку! Темы, связанные с безопасностью ASP.NET (такие как аутентификация на основе форм или анонимные профили), в книге не рассматриваются. Подробные сведения ищите в документации .NET Framework 4.6 SDK.

Группирование данных профиля и сохранение специальных объектов

В завершение давайте посмотрим, каким образом данные профиля могут определяться в файле Web.config. В текущем профиле просто определены четыре порции данных, которые доступны напрямую через тип Profile. При построении более сложных профилей может быть удобно группировать связанные данные под уникальным именем. Взгляните на следующее изменение:

```
<profile>
  <properties>
    <group name ="Address">
      <add name="StreetAddress" type="String" />
      <add name="City" type="String" />
      <add name="State" type="String" />
    </group>
    <add name="TotalPost" type="Integer" />
  </properties>
</profile>
```

На этот раз определена специальная группа по имени Address, включающая название улицы, город и штат пользователя. Для доступа к этим данным внутри страниц потребуется модифицировать код, указав Profile.Address для получения каждого подэлемента. Например, вот обновленный метод GetUserAddress() (обработчик события Click для кнопки Submit Data нуждается в похожих изменениях):

```
private void GetUserAddress()
{
  // Здесь происходит чтение из базы данных.
  lblUserData.Text =
   $"You live here: {Profile.Address.StreetAddress}, {Profile.Address.City}, "
   + $"{Profile.Address.State}";
}
```

Перед запуском приложения необходимо удалить файл ASPNETDB.mdf из папки App_Data, чтобы обеспечить обновление схемы базы данных. После этого пример вебсайта должен выполняться без ошибок.

На заметку! Профиль может содержать столько групп, сколько вы считаете нужным. Просто определите множество элементов <group> внутри <properties>.

И, наконец, полезно отметить, что профиль может также хранить в ASPNETDB.mdf специальные объекты (и извлекать их). В целях иллюстрации предположим, что необходимо построить специальный класс (или структуру) для представления данных адреса пользователя. Единственное требование, предъявляемое API-интерфейсом Profile к такому типу, заключается в том, что он должен быть помечен атрибутом [Serializable]:

```
[Serializable]
public class UserAddress
{
  public string Street = string.Empty;
  public string City = string.Empty;
  public string State = string.Empty;
}
```

При наличии такого класса определение профиля может быть модифицировано следующим образом (специальная группа удалена, хотя это вовсе не обязательно):

```
<profile>
  <properties>
    <add name="AddressInfo" type="UserAddress" serializeAs ="Binary"/>
    <add name="TotalPost" type="Integer" />
  </properties>
</profile>
```

Обратите внимание, что в случае добавления к профилю типов [Serializable] в атрибуте type указывается полностью заданное имя сохраняемого типа. Как вы увидите в окне IntelliSense среды Visual Studio, основными вариантами являются двоичные, XML и строковые данные. Теперь, когда информация адреса организована в виде специального класса, понадобится модифицировать кодовую базу:

```
private void GetUserAddress()
{
  // Здесь происходит чтение из базы данных.
  lblUserData.Text =
    $"You live here: {Profile.AddressInfo.Street}, {Profile.AddressInfo.City}, "
    + $"{Profile.AddressInfo.State}";
}
```

Конечно, с API-интерфейсом Profile связаны многие другие аспекты кроме тех, которые здесь обсуждались. Например, свойство Profile в действительности инкапсулирует тип ProfileCommon. С помощью этого типа можно программно получать всю инфор-

мацию о конкретном пользователе, удалять (или добавлять) профили в ASPNETDB.mdf, обновлять данные профиля и т.д.

Более того, API-интерфейс Profile имеет многочисленные точки расширения, которые позволяют оптимизировать способ доступа диспетчера профилей к таблицам базы данных ASPNETDB.mdf. Как и можно было ожидать, существует много способов сокращения количества обращений к базе данных. Заинтересованным читателям рекомендуем обратиться за подробностями в документацию .NET Framework 4.6 SDK.

Исходный код. Веб-сайт FunWithProfiles доступен в подкаталоге Chapter_33.

Резюме

В этой главе имеющиеся у вас знания ASP.NET были дополнены сведениями о применении типа HttpApplication. Вы видели, что тип HttpApplication предлагает несколько стандартных обработчиков событий, которые позволяют перехватывать разнообразные события уровня приложения и уровня сеанса. Большая часть главы была посвящена рассмотрению различных приемов управления состоянием. Вспомните, что состояние представления используется для автоматического заполнения значениями виджетов HTML между обратными отправками определенной страницы. Кроме того, вы узнали о разнице между данными уровня приложения и данными уровня сеанса, об управлении cookie-наборами и о кеше приложения ASP.NET.

Наконец, вы ознакомились с интерфейсом ASP.NET Profile API. Как было показано, эта технология предлагает готовое решение задачи сохранения пользовательских данных между сеансами. С применением конфигурационного файла Web.config веб-сайта можно определять любое количество элементов профиля (включая группы элементов и типы [Serializable]), которые будут автоматически сохраняться в базе данных ASPNETDB.mdf.

ГЛАВА 34

ASP.NET MVC и ASP.NET Web API

В предшествующих трех главах была раскрыта инфраструктура ASP.NET Web Forms, а также связанные с веб-приложениями концепции наподобие HTTP и HTML. В настоящей главе представлены два нововведения в экосистеме ASP.NET — инфраструктуры ASP.NET MVC и Web API. Инфраструктура MVC происходит из сообщества пользователей (конкретно движения ALT.NET), желающих заполучить инфраструктуру, которая более тесно взаимодействует с HTTP, обеспечивает более высокую тестируемость и поддерживает концепцию разделения ответственности. Несмотря на то что Web Forms по-прежнему занимает львиную долю рынка разработки веб-приложений .NET, инфраструктура MVC быстро наращивает темпы своего внедрения.

Мы начнем главу с краткого объяснения шаблона MVC и затем перейдем непосредственно к процессу создания проекта MVC. В стандартном типе проекта присутствует множество шаблонных частей, которые рассматриваются в последующих разделах. После получения хорошего представления о шаблоне MVC мы построим веб-приложение `CarLotMVC`, являющееся основанным на MVC подмножеством веб-сайта, который создавался на протяжении последних трех глав.

Далее вы ознакомитесь с инфраструктурой Web API, в значительной степени построенной на основе MVC, и многими концепциями, включая маршрутизацию, контроллеры и действия. Инфраструктура ASP.NET Web API позволяет задействовать имеющиеся знания MVC при разработке служб REST без конфигурирования и написания связующего кода, обязательного в случае использования WCF (см. главу 25). Мы создадим службу REST по имени `CarLotWebAPI`, которая открывает доступ к функциональности CRUD (create, read, update, delete — создание, чтение, обновление, удаление) для записей таблицы `Inventory`. Глава завершается модификацией веб-приложения `CarLotMVC` с целью применения в качестве источника данных службы `CarLotWebAPI` вместо инфраструктуры Entity Framework и библиотеки `AutoLotDAL`.

Введение в шаблон MVC

Шаблон "модель-представление-контроллер" (Model-View-Controller — MVC) существовал с 1970-х годов (и был создан для использования в Smalltalk), но относительно недавно его популярность резко возросла. Инфраструктурами MVC располагают многие языки, в том числе Java (в особенности Spring Framework), Ruby (Ruby on Rails), .NET (после появления ASP.NET MVC в 2007 году) и многие клиентские библиотеки JavaScript, такие как Angular и EmberJS.

Если следующее далее описание шаблона MVC напомнит вам определение шаблона MVVM из главы 30, то вы будете совершенно правы. Шаблон MVVM задействует многие компоненты MVC (наряду с шаблоном "Модель представления" (Presentation Model)). Но достаточно истории. Давайте перейдем к исследованиям шаблона.

Модель

Точно так же, как в MVVM, модель — это данные в приложении. Данные обычно представлены с помощью простых старых объектов CLR (POCO), подобных построенным в библиотеке `AutoLotDAL` (см. главу 23) и применяемым в примерах MVVM (см. главу 30). Классы модели могут (и часто будут) иметь встроенную логику проверки достоверности, и в зависимости от используемой инфраструктуры JavaScript клиентской стороны (такой как `knockout.js`) могут быть сконфигурированы как наблюдаемые объекты.

Представление

Представление — это пользовательский интерфейс приложения, который визуализирует вывод. Представление должно быть как можно более легковесным.

Контроллер

Контроллер является своего рода мозговым центром функционирования. Контроллеры имеют две сферы ответственности; во-первых, они принимают от пользователя команды/запросы (называемые *действиями*) и упорядочивают их подходящим образом (как для хранилища), а во-вторых, они отправляют любые изменения представлению. Контроллеры (а также модели и представления) должны быть легковесными и задействовать другие компоненты для поддержки разделения ответственности. Все выглядит просто, не так ли? Прежде чем переходить к построению приложения MVC, обратимся к старому вопросу.

Почему появилась инфраструктура MVC?

Ко времени выпуска ASP.NET MVC в 2007 году инфраструктура ASP.NET Web Forms находилась в производственной эксплуатации уже шесть лет. С применением Web Forms были построены тысячи сайтов, число которых ежедневно увеличивалось. Так почему в Microsoft решили создать новую инфраструктуру с нуля? Перед тем, как ответить на этот вопрос, уместно провести краткий экскурс в прошлое.

Когда появилась инфраструктура ASP.NET Web Forms, разработка веб-приложений не была настолько продуктивной, как в наши дни. Парадигма отсутствия состояния оказалась трудной в восприятии, особенно для разработчиков интеллектуальных клиентов (занимающихся созданием настольных приложений с помощью Visual Basic 6, MFC и PowerBuilder). Чтобы заполнить брешь в знаниях и упростить разработчикам процесс построения веб-сайтов, в Web Forms поддерживалось много концепций настольных приложений, таких как состояние (посредством состояния представления) и готовые элементы управления.

План сработал. Инфраструктура Web Forms в целом была воспринята хорошо, и многие разработчики сделали шаг в сторону разработки веб-приложений. Количество веб-сайтов на основе Web Forms продолжало расти, а история платформы .NET развивалась. Вместе с Web Forms и .NET совершенствовалась экосистема элементов управления Web Forms (и многих других элементов управления .NET) от преуспевающих независимых поставщиков. Словом, обстоятельства складывались удачно.

В то же время многие разработчики более глубоко изучили и освоились с концепцией отсутствия состояния, присущей программированию веб-приложений, протоколом HTTP, HMTL и JavaScript. Эти разработчики все меньше и меньше нуждались в связующих технологиях и хотели иметь намного больший контроль над визуализируемыми представлениями.

С каждой новой версией Web Forms в инфраструктуру добавлялись дополнительные средства и возможности, делая приложения все более тяжеловесными. Увеличение сложности разрабатываемых веб-сайтов означало разрастание элементов вроде состояния представления, грозящее выходом из-под контроля. Еще хуже то, что из-за ряда ранних решений, принятых при создании Web Forms (скажем, *местоположение* состояния представления внутри визуализируемой страницы), начали возникать проблемы, такие как ухудшение показателей производительности. Это стало причиной заметного отказа от .NET в пользу других языков, подобных Ruby (Ruby on Rails).

Но в Microsoft не могли (и разумно не стали) удалять из ASP.NET связующие технологии и другой код, чтобы не рисковать утерей работоспособности миллионов строк кода. Что-то нужно было предпринять, и модификация Web Forms не была вариантом (хотя, как вы узнали в предшествующих главах, для решения массы проблем в версии ASP.NET Web Forms 4.5 была проведена значительная работа). В Microsoft столкнулись с необходимостью принятия ряда трудных решений: как сохранить высокую продуктивность существующим разработчикам веб-приложений (и поддержать экосистему элементов управления, выросшую благодаря Web Forms), одновременно предоставив платформу для тех разработчиков, кто желал быть ближе к самой сути веб.

Появление ASP.NET MVC

Таким образом, благодаря всем указанным выше причинам появилась новая инфраструктура ASP.NET MVC. Она была создана для того, чтобы служить альтернативой ASP.NET Web Forms. Между ASP.NET Web Forms и ASP.NET MVC существуют заметные отличия, которые перечислены далее.

- Удалено:
  - файлы отделенного кода для представлений;
  - поддержка элементов управления серверной стороны;
  - состояние представления.
- Добавлено:
  - привязка моделей;
  - маршрутизация;
  - механизм представлений Razor (начиная с версии MVC 3).

Результатом стала легковесная инфраструктура, которая характеризуется высокой скоростью визуализации и спроектирована на поддержку тестируемости и разделения ответственности, но дополнительно требует глубоких знаний HTML и JavaScript, а также особенностей действительного функционирования протокола HTTP. Поскольку версии инфраструктуры, включая MVC 5, по-прежнему построены на основе тех же самых библиотек .NET, что и Web Forms и Web API, сочетание Web Forms, MVC и/или Web API становится жизнеспособным шаблоном разработки. Каждая инфраструктура имеет свои сильные и слабые стороны, и при выполнении имеющейся работы вы должны подбирать подходящий инструмент.

Соглашения по конфигурации

Одним из принципов ASP.NET MVC является соглашение по конфигурации (convention over configuration; или соглашение над конфигурацией, если делать акцент на преимуществе соглашения перед конфигурацией). Другими словами, для проектов MVC приняты специфические соглашения (такие как соглашения об именовании и структура папок), которые делают возможным выполнение большей части "магии" со стороны Visual Studio и .NET. В результате сокращается объем ручной или шаблонной конфигурации, но возникает необходимость в знании таких соглашений. По мере чтения этой главы вы будете наблюдать многие соглашения в действии.

Построение первого приложения ASP.NET MVC

Наступило время перейти от теории к практическому кодированию. Среда Visual Studio поставляется с довольно полным шаблоном проекта, предназначенного для построения приложений ASP.NET MVC, и вы освоите его при создании примера приложения `CarLotMVC`.

Мастер создания проекта

Начните с запуска Visual Studio, выберите пункт меню File⇨New Project (Файл⇨Создать проект). В древовидном представлении слева выберите узел Web внутри узла Visual C#, в панели по центру укажите ASP.NET Web Application (Веб-приложение ASP.NET) и введите в поле Name (Имя) имя `CarLotMVC` (рис. 34.1).

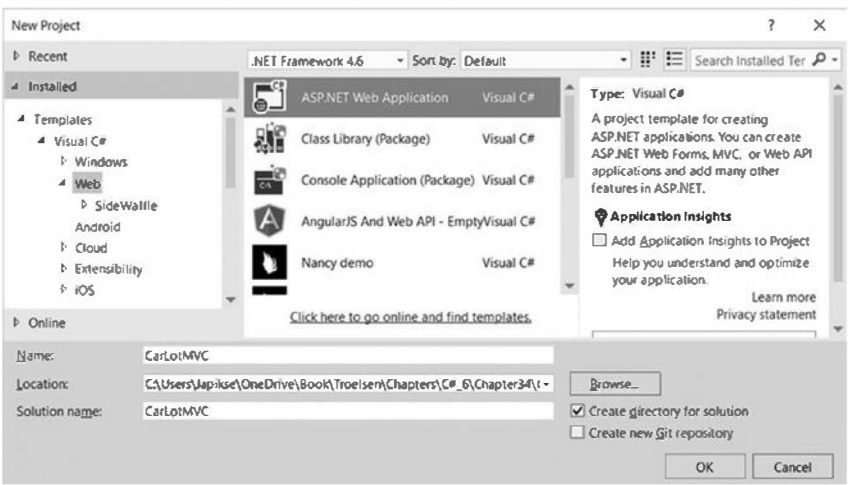


Рис. 34.1. Создание нового проекта веб-приложения ASP.NET

Если вы прорабатывали примеры в предшествующих главах, посвященных ASP.NET, то заметите, что тем же самым способом начинается построение приложения ASP.NET Web Forms. Процесс разработки приложений ASP.NET всех типов (Web Forms, MVC, Web API) начинается с единственного выбора в мастере создания проекта без необходимости в указании специфичной инфраструктуры. Известная как концепция "One ASP.NET" ("Единая ASP.NET"), такая возможность появилась в версии .NET 4.5.

На появившемся далее экране Select a Template (Выберите шаблон) мастера выберите шаблон MVC в области ASP.NET 4.6 Templates (Шаблоны ASP.NET 4.6). Обратите внимание, что в области Add folders and core references for (Добавить папки и основ-

ные ссылки для) флажок MVC отмечен, а другие флажки — нет. При желании создать гибридное приложение, которое поддерживает MVC и Web Forms, можно также отметить флажок Web Forms. В настоящем примере просто оставьте все так, как показано на рис. 34.2. Здесь также имеется флажок Add unit tests (Добавить модульные тесты). Если его отметить, то будет создан еще один проект, который предоставит базовую инфраструктуру для модульного тестирования разрабатываемого приложения ASP.NET. Пока не щелкайте на кнопке OK, т.к. нужно исследовать механизмы аутентификации, доступные для проекта.

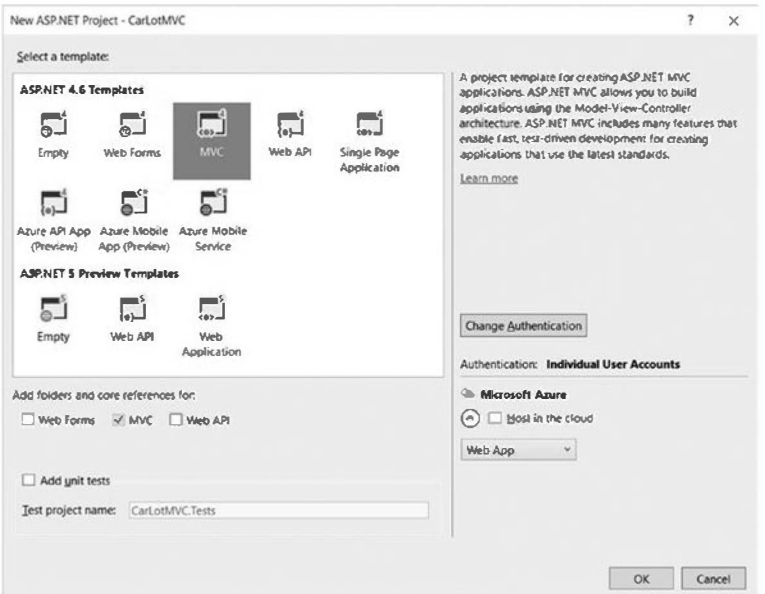


Рис. 34.2. Выбор шаблона MVC

Щелкните на кнопке Change Authentication (Изменить аутентификацию), в результате чего откроется диалоговое окно, приведенное на рис. 34.3. Оставьте выбранным переключатель Individual User Accounts (Учетные записи индивидуальных пользователей), щелкните на кнопке OK и затем на кнопке OK на экране Select a Template мастера создания нового проекта.

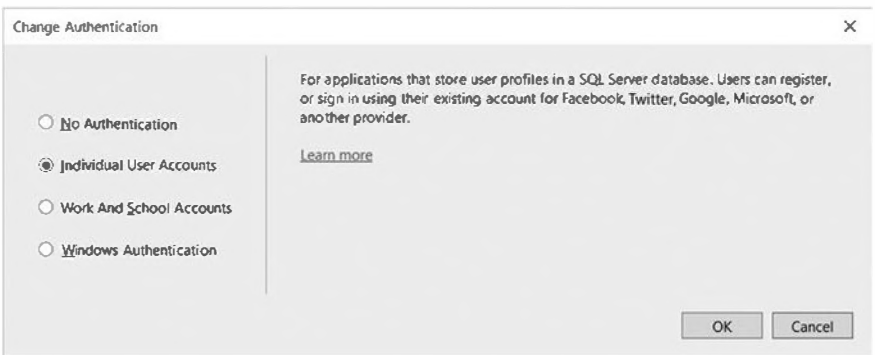


Рис. 34.3. Варианты аутентификации для проекта

В табл. 34.1 кратко описаны четыре варианта аутентификации, доступные приложениям MVC.

Таблица 34.1. Варианты аутентификации

Вариант	Описание
No Authentication (Аутентификация отсутствует)	Отсутствует механизм для входа, сущностные классы для членства или база данных членства
Individual User Accounts (Учетные записи индивидуальных пользователей)	Применяет для аутентификации пользователей систему ASP.NET Identity (ранее известную как ASP.NET Membership)
Work and School Accounts (Учетные записи мест работы или учебных заведений)	Предназначен для приложений, которые производят аутентификацию с помощью Active Directory, Azure Active Directory или Office 365
Windows Authentication (Аутентификация Windows)	Использует аутентификацию Windows. Предназначен для веб-сайтов в корпоративной сети

На заметку! Из-за ограничений по объему аутентификация в настоящей книге не рассматривается. За дополнительными сведениями по аутентификации в MVC обращайтесь к книге *ASP.NET MVC 5 с примерами на C# для профессионалов*, 5-е изд. (ИД "Вильямс", 2015 г.).

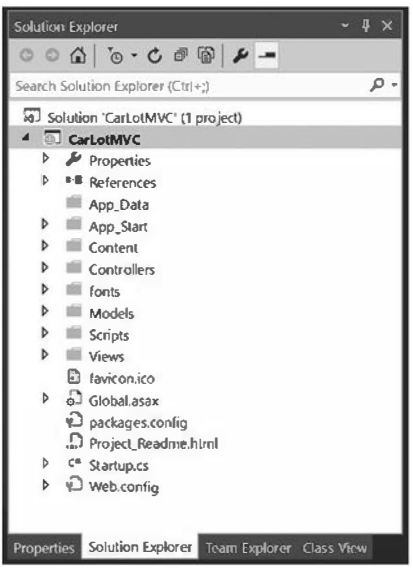


Рис. 34.4. Файлы и папки, сгенерированные для проекта приложения MVC

После завершения создания проекта вы обнаружите *массу* сгенерированных файлов и папок (рис. 34.4). Мы исследуем их в следующем разделе.

Компоненты базового проекта MVC

Некоторые из этих папок и файлов должны выглядеть знакомо, т.к. они имеют те же самые имена, что и у папок и файлов, доступных в проектах ASP.NET Web Forms.

Файлы в корневой папке проекта

Большинство файлов в проектах MVC должны быть помещены в специфические местоположения. Однако есть несколько файлов, находящихся в корневой папке проекта, не все из которых развертываются с веб-сайтом. В табл. 34.2 перечислены файлы из корневой папки сайта MVC с указанием, развертываются они или нет.

Папка `Models`

В папке `Models` находятся классы моделей. В крупных приложениях классы моделей должны быть организованы в библиотеки доступа к данным. Папка `Models` чаще всего применяется для хранения моделей, связанных с представлениями, таких как классы моделей, которые генерируются Visual Studio для системы ASP.NET Identity.

Папка `Controllers`

В папке `Controllers` располагаются контроллеры, используемые внутри приложения. Контроллеры подробно рассматриваются позже в главе.

Таблица 34.2. Файлы в корневой папке проекта

Файл	Описание	Развертывается?
`favicon.ico`	Значок, который браузер отображает в адресной строке возле имени страницы. Отсутствие этого файла может привести к проблемам с производительностью, т.к. браузер будет непрерывно его искать	Да
`Global.asax`/ `Global.asax.cs`	Точка входа в приложение (подобное ASP.NET Web Forms)	Да
`packages.config`	Конфигурационная информация для пакетов NuGet, применяемых в проекте	Да
`Project_Readme.html`	Специфичный для Visual Studio файл, который предоставляет полезные ссылки и другую информацию об инфраструктуре ASP.NET MVC	Нет
`Startup.cs`	Начальный класс для OWIN (ASP.NET Identity)	Да (скомпилированный)
`Web.config`	Конфигурационный файл проекта	Да

Файл `Global.asax.cs`

Вмешательство в конвейер обработки ASP.NET производится внутри файла `Global.asax.cs`. Здесь доступны те же самые события, что и в ASP.NET Web Forms. В стандартном шаблоне проекта используется только обработчик события `Application_Start`, но существует намного больше событий, которые можно при необходимости перехватывать. Самые распространенные события приведены в табл. 34.3.

Таблица 34.3. Распространенные события из файла `Global.asax.cs`

Событие	Описание
`Application_Start`	Возникает при первом обращении к приложению
`Application_End`	Возникает, когда приложение завершается
`Application_Error`	Возникает при появлении необработанной ошибки
`Session_Start`	Возникает при первом запросе нового сеанса
`Session_End`	Возникает, когда сеанс завершается (в том числе по причине тайм-аута)
`Application_BeginRequest`	Возникает, когда выполнен запрос к серверу
`Application_EndRequest`	Возникает в качестве последнего события в конвейерной цепочке выполнения HTTP, когда ASP.NET реагирует на запрос

Папка `Views`

В папке `Views` хранятся представления MVC, но в отличие от папок `Models` и `Controllers` существует соглашение относительно структуры папок, содержащихся внутри папки `Views`.

В самой папке `Views` находятся файлы `Web.config` и `_ViewStart.cshtml`. Файл `Web.config` является специфичным для представлений в этой иерархии папок, определяет базовый тип страницы (например, `System.Web.Mvc.WebViewPage`) и в проектах, основанных на Razor, добавляет все ссылки на сборки и операторы `using` для механиз-

ма Razor. В файле `_ViewStart.cshtml` указывается стандартная страница компоновки для применения в ситуации, когда представлению явно не назначена страница компоновки. Подробно это будет обсуждаться при рассмотрении компоновок. Страница компоновки аналогична мастер-странице в Web Forms и рассматривается позже в главе.

На заметку! Почему имя файла `_ViewStart.html` (и `_Layout.cshtml`) начинается с символа подчеркивания? Механизм представлений Razor первоначально был создан для платформы WebMatrix, которая позволяла визуализировать любой файл, имя которого *не* начинается с подчеркивания, поэтому все основные файлы (такие как компоновка и конфигурация) имеют имена, начинающиеся с символа подчеркивания. Вы также увидите, что такое соглашение об именовании используется для частичных представлений. Тем не менее, это не то соглашение, за соблюдением которого следит инфраструктура MVC, поскольку в MVC отсутствует проблема, присущая WebMatrix, но по традиции символ подчеркивания продолжает применяться.

Каждый контроллер получает собственную папку внутри папки `Views`. Такая структура папок является частью соглашения MVC; контроллер ищет свои представления в папке, имеющей такое же имя, как у класса контроллера (без слова `Controller`). Например, папка `Views/Home` содержит все представления для класса контроллера `HomeController`.

Папка `Shared`

Внутри папки `Views` есть специальная папка по имени `Shared`. Она доступна всем представлениям.

Папки ASP.NET

Существуют также папки, зарезервированные для ASP.NET. Примером может служить папка ASP.NET по имени `App_Data`, которая включена в стандартный шаблон проекта MVC. Она предназначена для хранения любых файлов данных, необходимых сайту. Также предусмотрены папки для хранения кода, ресурсов и тем. Папки ASP.NET можно добавить, щелкнув правой кнопкой мыши на имени проекта, выбрав в контекстном меню пункт Add⇨Add ASP.NET Folder (Добавить⇨Добавить папку ASP.NET) и затем выбрав в предложенном списке нужную папку (рис. 34.5). Папки ASP.NET не могут просматриваться из веб-сайта, даже если навигация по папкам включена.

Доступные папки ASP.NET кратко описаны в табл. 34.4.

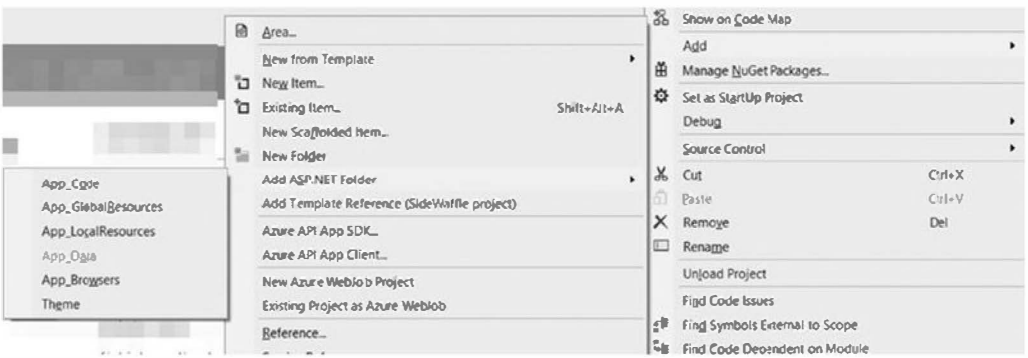


Рис. 34.5. Добавление новых папок ASP.NET

Таблица 34.4. Доступные папки ASP.NET

Папка	Описание
App_Code	Файлы кода, содержащиеся в этой папке, будут динамически компилироваться
App_GlobalResources	Хранит файлы ресурсов, доступные всему приложению. Обычно используется для локализации
App_LocalResources	Содержит ресурсы, доступные специфичной странице. Обычно применяется для локализации
App_Data	Содержит файлы данных, используемые приложением
App_Browsers	Хранит файлы возможностей браузеров
App_Themes	Хранит темы для сайта

Папка App_Start

В ранних версиях MVC весь код конфигурации сайта (наподобие маршрутизации и безопасности) содержался в классе Global.asax.cs. С ростом объема конфигурации разработчики инфраструктуры MVC разумно разнесли код по отдельным классам, чтобы более четко соблюдать принцип разделения ответственности. Вследствие такой переделки проектирования появилась папка App_Start и содержащиеся в ней классы (они перечислены в табл. 34.5). Любой код в папке App_Start автоматически компилируется в результирующий сайт.

Таблица 34.5. Файлы в папке App_Start

Файл	Описание
BundleConfig.cs	Создает пакеты файлов JavaScript и CSS. В этом классе могут (и должны) создаваться дополнительные пакеты
FilterConfig.cs	Регистрирует фильтры действий (таких как аутентификация или авторизация) на глобальном уровне
IdentityConfig.cs	Содержит классы поддержки для ASP.NET Identity
RouteConfig.cs	Класс, в котором настраивается таблица маршрутизации
Startup.Auth.cs	Точка входа для конфигурации ASP.NET Identity

Класс BundleConfig

Этот класс настраивает пакеты файлов CSS и JavaScript. По умолчанию в случае применения ScriptBundle все включенные файлы объединяются в пакет и подвергаются минификации (объединение в пакеты и минификация более подробно рассматриваются в следующем разделе) для версии выпуска, но не для отладочной версии. Управлять процессом можно посредством файла Web.config или в самом классе ScriptBundle. Чтобы отключить объединение в пакеты и минификацию, введите показанную ниже разметку в раздел system.web файла Web.config верхнего уровня (если она еще не существует):

```
<system.web>
  <compilation debug="true" targetFramework="4.6" />
</system.web>
```

Или же установите свойство `BundleTable.EnableOptimizations` в `false` внутри метода `RegisterBundles()` класса `BundleConfig`:

```
public static void RegisterBundles(BundleCollection bundles)
{
  bundles.Add(new ScriptBundle("~/bundles/jquery")
.Include("~/Scripts/jquery-{version}.js"));
  // Для краткости остальной код не показан.
  BundleTable.EnableOptimizations = false;
}
```

Объединение в пакеты

Объединение в пакеты — это процесс комбинирования множества файлов в один. Причин объединения несколько; главная из них — увеличение скорости работы сайта. В браузерах имеется ограничение на количество параллельно загружаемых файлов из отдельно взятого сервера. Если ваш сайт содержит много мелких файлов (что обычно способствует лучшему разделению ответственности), то пользовательский интерфейс может замедлить реагирование. Помочь в решении проблемы может объединение в пакеты и использование сетей доставки содержимого (content delivery network — CDN). Естественно, вы должны разумным образом реализовать поддерживаемые действия, т.к. наличие одного гигантского файла, возможно, будет ничем не лучше огромного количества небольших файлов.

Минификация

Подобно объединению в пакеты минификация направлена на ускорение загрузки веб-страниц. В целях читабельности в файлах CSS и JavaScript применяются значащие имена переменных и функций, комментарии и разнообразное форматирование (во всяком случае, так должно быть). Проблема в том, что при передаче по сети учитывается каждый мелкий фрагмент, особенно когда дело приходится иметь с мобильными клиентами.

Минификация представляет собой процесс замены длинных имен короткими (иногда даже односимвольными) именами, а также удаления дополнительных пробелов и другого форматирования. Наиболее современные инфраструктуры поставляются с двумя версиями своих файлов CSS и JavaScript. Инфраструктура Bootstrap в данном отношении ничем не отличается, предлагая файл `bootstrap.css` для использования во время разработки приложения и файл `bootstrap.min.css` для производственной версии приложения.

Класс `FilterConfig`

Фильтры — это специальные классы, которые предоставляют механизм для перехвата действий и запросов. Они могут применяться на уровне действия, контроллера или глобальном уровне. В MVC существуют четыре типа фильтров, которые кратко описаны в табл. 34.6.

Удостоверение

Файлы `Identity.config.cs` и `Startup.Auth.cs` используются для поддержки системы ASP.NET Identity, которая настолько обширна, что в настоящей главе не рассматривается. На самом деле безопасности и удостоверениям можно было бы посвятить отдельную книгу. Как упоминалось ранее, система ASP.NET Identity основана на OWIN (Open Web Interface for .NET — открытый веб-интерфейс для .NET), что отделяет систему Identity и ее зависимости от IIS. Наряду с тем, что это отделение не вступает в игру для MVC в .NET 4.6, оно может быть существенным в ASP.NET Web API, если вы самостоятельно размещаете свою службу.

Таблица 34.6. Фильтры в ASP.NET MVC

Тип	Описание
Фильтры авторизации	Реализуют интерфейс `IAuthorizationFilter` и запускаются перед любым другим фильтром. Двумя примерами являются атрибуты `[Authorize]` и `[AllowAnonymous]`. Скажем, класс `AccountController` снабжен атрибутом `[Authorize]`, чтобы требовать аутентифицированного с помощью ASP.NET Identity пользователя, а действие `Login` помечено атрибутом `[AllowAnonymous]`, чтобы разрешить доступ любому пользователю
Фильтры действий	Реализуют интерфейс `IActionFilter` и позволяют перехватывать выполнение действия посредством методов `OnActionExecuting()` и `OnActionExecuted()`
Фильтры результатов	Реализуют интерфейс `IResultFilter` и позволяют перехватывать результат действия с помощью методов `OnResultExecuting()` и `OnResultExecuted()`
Фильтры исключений	Реализуют интерфейс `IExceptionFilter` и выполняются при появлении необработанного исключения внутри конвейера выполнения ASP.NET. По умолчанию фильтр `HandleError` конфигурируется на глобальном уровне. Он отображает страницу представления ошибок `Error.cshtml`, расположенную в папке `Shared/Error`

Класс `RouteConfig`

В ранних версиях ASP.NET Web Forms определение URL сайта осуществлялось на основе физической структуры папок проекта. Это можно было изменить посредством модулей (`HttpModule`) и обработчиков (`HttpHandler`), но результат оказывался далеким от идеала. Инфраструктура MVC изначально поддерживала маршрутизацию, которая позволяет придавать URL форму, лучше подходящую для пользователей. Мы рассмотрим это позже в главе.

Папка `Content`

Папка `Content` предназначена для хранения файлов CSS сайта. Кроме того, она часто применяется для хранения файлов изображений и другого отличного от кода содержимого. В отличие от многих перечисленных здесь папок зависимости, связанной с именем данной папки, не существует; это просто удобное соглашение.

В состав ASP.NET MVC входит Bootstrap, которая на сегодняшний является одной из наиболее популярных инфраструктур HTML, CSS и JavaScript. Два из стандартных файла CSS (`bootstrap.css` и `bootstrap.min.css`) относятся к инфраструктуре Bootstrap, а в файл `site.css` вы будете помещать стили CSS, специфичные для сайта.

Инфраструктура Bootstrap

Bootstrap — это инфраструктура HTML, CSS и JavaScript с открытым кодом для разработки быстрореагирующих веб-сайтов с использованием метода Mobile First (Сначала мобильный). В Microsoft приняли решение включить Bootstrap в версию MVC 4 и продолжили делать это в версии MVC 5, так что в стандартном шаблоне проекта для MVC 5 инфраструктура Bootstrap применяется для стилизации шаблонных страниц. Хотя для подробного раскрытия Bootstrap в книге не хватит места, в главе будут применяться некоторые средства Bootstrap для обеспечения дополнительной стилизации разрабатываемого примера сайта.

Папка Fonts

Инфраструктура Bootstrap поставляется с набором шрифтов GlyphIcons Halflings, который будет использоваться позже в главе для улучшения пользовательского интерфейса приложения. Версия Bootstrap, с которой имеет дело шаблон проекта MVC, требует нахождения шрифтов в папке `Fonts`.

Папка Scripts

В папке `Scripts` размещены файлы JavaScript. В табл. 34.7 перечислены файлы, входящие в состав стандартного проекта, и приведено их краткое описание.

Таблица 34.7. Файлы JavaScript в шаблоне проекта ASP.NET MVC

Файл JavaScript	Описание
`_references.js`	Файл `_references.js` предназначен для средства IntelliSense в Visual Studio. В него можно добавлять дополнительные ссылки, указывающие на специальные файлы JavaScript
`bootstrap.js` `bootstrap.min.js`	Это файлы JavaScript для Bootstrap. Файл `.min` представляет собой минифицированную версию

`jquery-1.x.intellisense.js`
`jquery-1.x.js`
`jquery-1.x.min.js`
`jquery-1.x.min.map`
	jQuery является доминирующей инфраструктурой JavaScript для разработчиков веб-приложений. В дополнение к возможностям манипулирования моделью DOM от библиотеки jQuery зависят многие инфраструктуры, в том числе подключаемый модуль проверки достоверности, который применяется в шаблоне проекта MVC. Шаблон проекта MVC поставляется с более старыми версиями jQuery. В следующем разделе вы узнаете, как его модернизировать до текущей версии

`jquery.validate-vsdoc.js`
`jquery.validate.js`
`jquery.validate.min.js`
	Подключаемый модуль jQuery Validate значительно упрощает проверку достоверности на стороне клиента. Файл `vsdoc` предназначен для средства IntelliSense в Visual Studio, а файл `.min` — это минифицированная версия

`jquery.validate.unobtrusive.js` `jquery.validate.unobtrusive.min.js`	Подключаемый модуль Unobtrusive jQuery Validation работает с jQuery Validation, используя атрибуты HTML 5.0 для выполнения проверки достоверности на стороне клиента
`modernizr-2.x.js`	Подключаемый модуль Modernizr содержит последовательность быстрых тестов для определения возможностей браузеров. Он работает напрямую с браузером, а не полагается на файлы возможностей браузеров, которые вполне могут оказаться устаревшими
`respond.js` `respond.min.js`	`Respond.js` — это экспериментальный подключаемый модуль jQuery для построения веб-сайтов с быстрореагирующим содержимым

Модернизация пакетов NuGet до текущих версий

Как видите, существует *масса* файлов и пакетов, которые входят в состав шаблона проекта MVC, и многие из них являются инфраструктурами с открытым кодом. Проекты с открытым кодом обновляются намного чаще, чем в Microsoft могут (или должны) выпускать обновления продукта Visual Studio. При создании нового проекта часто обнаруживается, что пакеты уже устарели.

К счастью, их модернизация выполняется легко с помощью графического пользовательского интерфейса NuGet. Щелкните правой кнопкой мыши на имени проекта и выберите в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet). В открывшемся окне NuGet Package Manager (Диспетчер пакетов NuGet) измените выбор в поле со списком Filter (Фильтр) на Installed (Установленные), что приведет к отображению только установленных пакетов. Для пакетов, помеченных значком с изображением стрелки вверх синего цвета, доступны обновления (рис. 34.6). Модернизируйте все пакеты до их текущих версий.

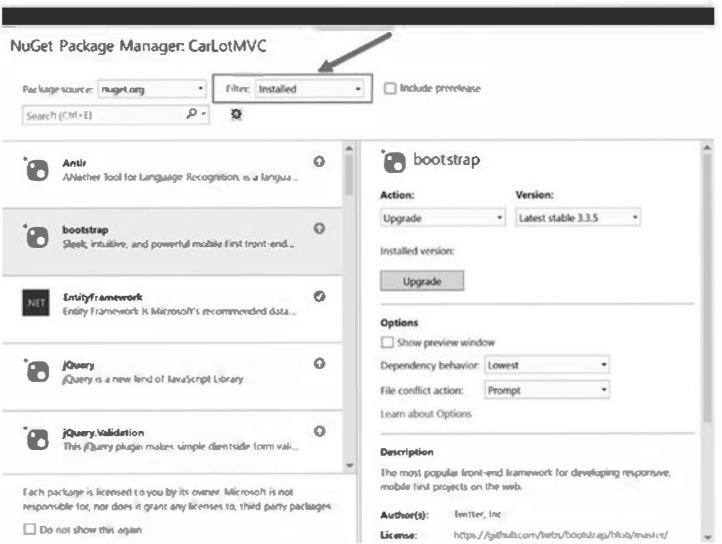


Рис. 34.6. Обновление пакетов NuGet

Пробный запуск сайта

Прежде чем двигаться дальше, запустите проект и просмотрите, что включено в стандартный шаблон проекта. Вы заметите, что доступно довольно много средств. Шаблон имеет меню, несколько экранов и возможности входа (укомплектованные экраном регистрации). На рис. 34.7 показана домашняя страница.

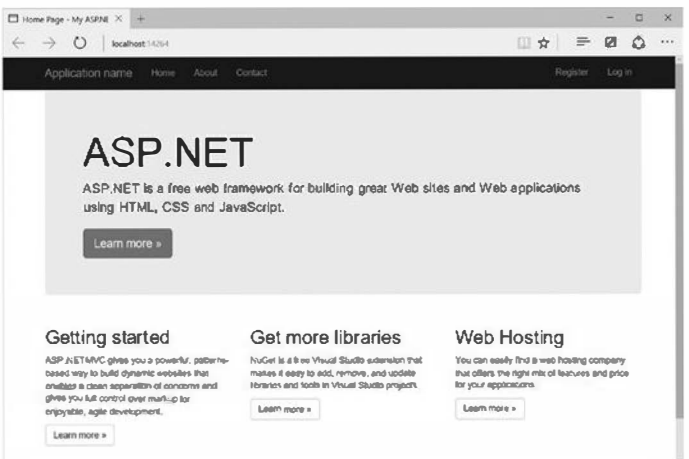


Рис. 34.7. Стандартная домашняя страница

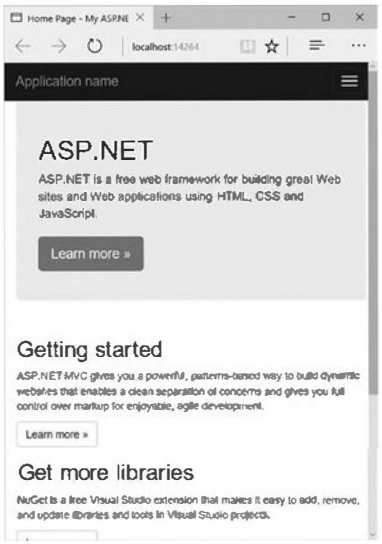


Рис. 34.8. Быстрореагирующее представление домашней страницы

Как упоминалось ранее, Bootstrap является *быстрореагирующей* инфраструктурой, способной адаптировать пользовательский интерфейс на основе окна просмотра. Уменьшите размеры окна браузера; вы увидите, что домашняя страница изменилась, став дружественной больше к мобильным устройствам (рис. 34.8). Меню свернуто до единственного значка, а горизонтальная компоновка трех разделов Learn more (Узнать больше) изменилась на вертикальную.

Маршрутизация

Маршрутизация — это способ, которым MVC сопоставляет запросы URL с контроллерами и действиями в приложении, вместо старого процесса сопоставления URL с файловой структурой, принятого в Web Forms. Снова запустите проект `CarLotMVC` и обратите внимание на URL. Здесь URL выглядит как `http://localhost:14264` (на вашей машине номер порта, скорее всего, будет другим). Теперь щелкните на ссылке Contact (Контакт); в результате URL изменится на `http://localhost:14264/Home/Contact`. Просмотрев решение, вы не обнаружите в нем путь к папке `Home/Contact`. На самом деле таблица маршрутов отображает URL запроса `Home/Contact` на вызов метода действия `Contact()` класса контроллера `HomeController`. (Контроллеры и действия подробно обсуждаются далее в главе.)

Шаблоны URL

Записи маршрутизации состоят из шаблонов URL, включающих в себя переменные-заполнители и литералы, которые помещены в коллекцию, известную как *таблица маршрутов*, причем каждая запись определяет отличающийся шаблон URL для сопоставления. Переменные-заполнители могут быть специальными переменными или браться из списка заранее определенных переменных. Например, `{controller}` и `{action}` направляют на контроллер и действие. Заполнитель `{id}` является специальным и транслируется в параметр для действия. Когда URL проверяется по таблице маршрутов, это делается последовательно и упорядочено. Сопоставление URL с записями в коллекции производится в порядке их добавления. Процесс останавливается, когда найдено первое совпадение; совершенно не имеет значения, что позже в таблице маршрутов может находиться *лучшее* соответствие. Этот важный фактор следует иметь в виду при добавлении записей в таблицу маршрутов. Откройте файл `RouteConfig.cs` (из папки `App_Start`) и просмотрите его содержимое:

```
public class RouteConfig
{
  public static void RegisterRoutes(RouteCollection routes)
  {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
      name: "Default",
      url: "{controller}/{action}/{id}",
      defaults: new { controller = "Home", action = "Index",
                      id = UrlParameter.Optional }
    );
  }
}
```

Первая строка кода указывает механизму маршрутизации на необходимость игнорирования запросов, имеющих расширение .axd, которое обозначает обработчик HTTP. Метод IgnoreRoute() передает запрос веб-серверу, в данном случае IIS. Шаблон {*pathinfo} поддерживает переменное количество параметров, охватывая любой URL, который включает обработчик HTTP.

Метод MapRoute() добавляет новую запись в таблицу маршрутов. В вызове указывается имя, шаблон URL и стандартные значения для переменных в шаблоне URL. Применяемый здесь шаблон URL уже обсуждался ранее; он обеспечивает обращение к заданному действию определенного контроллера и передачу элемента {id} методу действия в качестве параметра. Примером URL, который мог бы обслуживаться таким маршрутом, является Inventory/Add/5. В результате вызывается метод действия Add() класса контроллера InventoryController с передачей значения 5 в параметре id.

В параметре defaults указано, каким образом заполнять пустые фрагменты в частичных URL. С учетом предыдущего кода, если в URL ничего не задано (например, http://localhost:14264), то механизм маршрутизации вызовет метод действия Index() класса HomeController без параметра id. Параметр defaults является последовательным, т.е. допускает исключение справа налево. Ввод URL вида http://localhost14264/Add/5 не пройдет сопоставление с шаблоном {controller}/{action}/{id}.

Создание маршрутов для страниц Contact и About

Разумеется, после развертывания сайта URL будет выглядеть не как http://localhost:14264, а станет чем-нибудь значащим наподобие http://skimedic.com. Одним из преимуществ маршрутизации является возможность придания URL формы, удобной для пользователей. Это позволяет создавать URL, которые легко запоминать и искать с помощью поисковых механизмов. Например, вместо http://skimedic.com/Home/Contact и http://skimedic.com/Home/About было бы лучше также иметь в распоряжении http://skimedic.com/Contact и http://skimedic.com/About (разумеется, без утраты более длинных отображений). Посредством маршрутизации добиться этого несложно.

Откройте файл RouteConfig.cs и поместите следующую строку кода *после* вызова IgnoreRoutes(), но *перед* установкой стандартного маршрута:

```
routes.MapRoute("Contact", "Contact", new { controller = "Home",
                                             action = "Contact" });
```

Эта строка добавляет в таблицу маршрутов новую запись по имени Contact, которая содержит только одно литеральное значение Contact. Она отображается на Home/Contact, но не со стандартными, а с жестко закодированными значениями. Чтобы протестировать ее, запустите приложение и щелкните на ссылке Contact. В результате URL изменится на http://localhost:14264/Contact — именно то, что и требовалось, т.е. URL, легкий для запоминания пользователями.

А теперь введите URL вида http://localhost:14264/Home/Contact/Foo. Он по-прежнему работает! Дело в том, что данный URL не прошел сопоставления с первой записью в таблице маршрутов и попал во вторую запись маршрута, которая дала соответствие. Измените URL на http://localhost:14264/Home/Contact/Foo/Bar. На этот раз сопоставление завершилось неудачей, т.к. данный URL не соответствует ни одному из маршрутов. Исправьте проблему добавлением {*pathinfo} к шаблону, что позволит указывать любое количество дополнительных параметров URL. Модифицируйте запись маршрута Contact, как показано ниже:

```
routes.MapRoute("Contact", "Contact/{*pathinfo}",
                 new { controller = "Home", action = "Contact" });
```

После этого ввод URL вида `http://localhost:14264/Home/Contact/Foo/Bar` приведет к отображению той же самой страницы `Contact`. Проблема решена. Такой URL легко запоминается пользователями, и даже если они допишут к нему дополнительные ненужные данные, то все равно будут попадать на искомую страницу.

В завершение примера добавьте следующую строку непосредственно после создания записи маршрута `Contact`, чтобы создать маршрут для страницы `About`:

```
routes.MapRoute("About", "About/{*pathinfo}",
                new { controller = "Home", action = "About" });
```

Перенаправление пользователей с применением маршрутизации

Еще одно преимущество маршрутизации заключается в том, что вам больше не придется жестко кодировать URL для других страниц сайта. Записи маршрутов используются в двунаправленной манере, т.е. не только для сопоставления с входящими запросами, но также для построения URL для сайта. Например, откройте файл `_Layout.cshtml` из папки `Views/Shared`. Обратите внимание на приведенную далее строку (пока что не беспокойтесь о синтаксисе; вскоре он будет объяснен):

```
@Html.ActionLink("Contact", "Contact", "Home")
```

Вспомогательный метод HTML по имени `ActionLink()` создает гиперссылку HTML с отображаемым текстом Contact для действия `Contact` в контроллере `Home`. Как и в отношении входящих запросов, механизм маршрутизации начинает просмотр сверху и двигается вниз до тех пор, пока не найдет совпадение. Показанная выше строка кода соответствует добавленному ранее маршруту `Contact` и применяется для создания следующей ссылки:

```
<a href="/Contact">Contact</a>
```

Если бы маршрут `Contact` не был добавлен, то механизм маршрутизации создал бы такую ссылку:

```
<a href="/Home/Contact">Contact</a>
```

На заметку! В этом разделе введено несколько новых элементов, которые пока еще не раскрывались, в том числе синтаксис `@`, объект `Html` и файл `_Layout.cshtml`. Все они будут рассматриваться довольно скоро. Важно понимать, что таблица маршрутов используется не только для разбора входящих запросов и их передачи подходящему ресурсу на обработку, но также и для создания URL, основанных на указанных ресурсах.

Добавление библиотеки AutoLotDAL

Приложения нуждаются в данных, и `CarLotMVC` тому не исключение. Начните с копирования проекта `AutoLotDAL` из главы 31 со всеми его файлами в папку `CarLotMVC` (на том же уровне, что и файл решения `CarLotMVC`). Можете также скопировать проект из подкаталога `Chapter_34` загружаемого кода. Библиотека доступа к данным будет обновлена по сравнению с версией, построенной в главе 31, поэтому нельзя просто добавить ссылку на готовую сборку `.dll`.

Добавьте скопированный проект в решение, дл чего щелкните правой кнопкой мыши на решении `CarLotMVC`, выберите в контекстном меню пункт Add⇨Existing Project (Добавить⇨Существующий проект), в открывшемся диалоговом окне перейдите в папку `AutoLotDAL` и укажите файл `AutoLotDAL.csproj`. Добавьте ссылку на `AutoLotDAL`, щелкнув правой кнопкой мыши на проекте `CarLotMVC` и выбрав в контекстном меню пункт Add⇨Reference (Добавить⇨Ссылка). В диалоговом окне Reference Manager (Диспетчер

ссылок) выберите элемент Solution (Решение) внутри узла Projects (Проекты) в левой панели, отметьте флажок рядом с AutoLotDAL (рис. 34.9) и щелкните на кнопке OK.

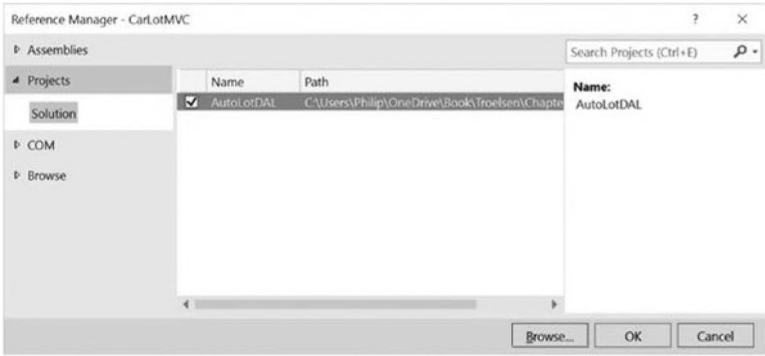


Рис. 34.9. Добавление ссылки на проект для AutoLotDAL

Следующий шаг связан с добавлением в файл Web.config проекта CarLotMVC строки подключения к базе данных AutoLot. Поскольку система ASP.NET Identity работает с инфраструктурой Entity Framework (EF), устанавливать пакет EF, как это делалось при построении сайтов Web Forms, не понадобится. Вам просто необходимо добавить еще одну строку подключения. Откройте файл Web.config и найдите в нем элемент <connectionStrings>. Либо скопируйте значение AutoLotConnection из файла App.config в проекте AutoLotDAL, либо вручную введите значение AutoLotConnection, как показано ниже (точное значение может несколько отличаться в зависимости от установленной копии SQL Server Express):

```
<connectionStrings>
  <!-- Для краткости стандартная строка подключения не показана. -->
    <add name="AutoLotConnection" connectionString="data source=localhost\
SQLEXPRESS2014;initial catalog=AutoLot;integrated security=True;Multiple
ActiveResultSets=True;App=EntityFramework" providerName="System.Data.
SqlClient" />
  </connectionStrings>
```

На заметку! Вероятно, вы заметили, что система ASP.NET Identity использует для источника данных LocalDb (облегченная версия SQL Server, которая не требует администрирования), и вы, безусловно, можете применять LocalDb в проектах ASP.NET MVC (как поступают многие разработчики). Учитывая, что для всех примеров работы с данными в книге используется одна и та же база данных, начиная с главы 21, было решено создать базу данных SQL Express, а не LocalDb. Возникнет ли какая-то разница в том, каким образом будет строиться этот сайт? Нет, не возникнет. Конечный результат тот же, поэтому основное внимание будет сосредоточено на коде C#, в то время как аспект SQL Server останется простым и незаметным.

Контроллеры и действия

Как обсуждалось ранее, когда запрос поступает от браузера, он (обычно) отображается на метод действия из определенного класса контроллера. Хотя звучит загадочно, на самом деле все довольно просто. Контроллер — это класс, который унаследован от одного из двух абстрактных классов, Controller и AsyncController. Обратите внимание, что вы также можете создать контроллер с нуля, реализовав интерфейс IController, но данная темя выходит за рамки настоящей книги. Метод действия является методом класса контроллера.

Добавление контроллера `Inventory`

Чтобы понять контроллеры и действия, лучше всего добавить новый контроллер с действиями с применением средств, встроенных в Visual Studio. Щелкните правой кнопкой мыши на папке `Controllers` в проекте и выберите в контекстном меню пункт Add⇨Controller (Добавить⇨Контроллер), как демонстрируется на рис. 34.10.

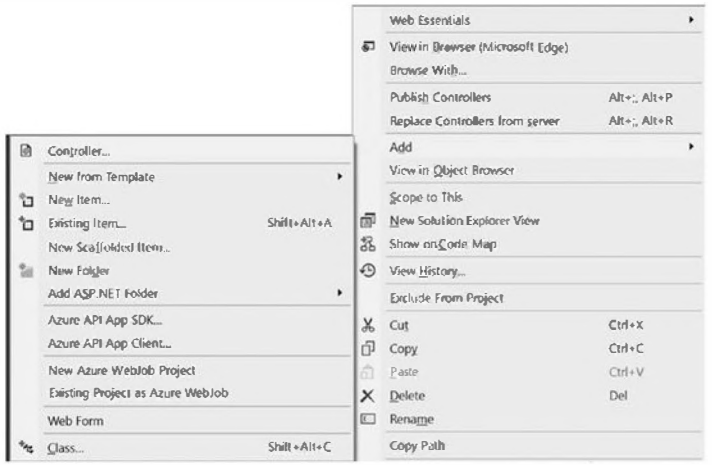


Рис. 34.10. Добавление нового контроллера

В результате откроется диалоговое окно Add Scaffold (Добавление шаблона), показанное на рис. 34.11. Здесь доступно множество вариантов, из которых нужно выбрать MVC 5 Controller with views, using Entity Framework (Контроллер MVC 5 с представлениями, использующий Entity Framework).

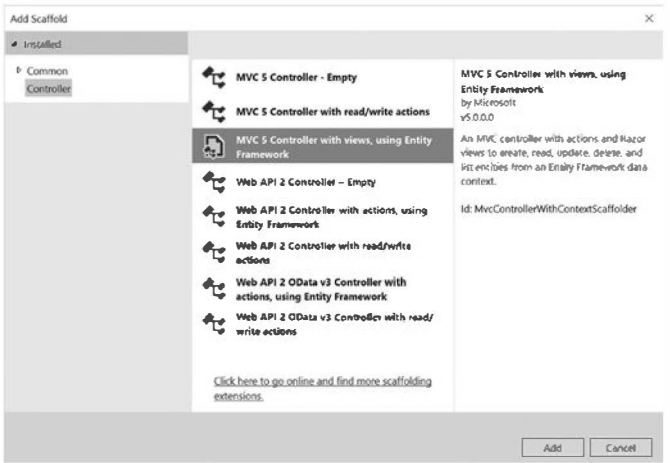


Рис. 34.11. Диалоговое окно Add Scaffold

Откроется диалоговое окно Add Controller (Добавление контроллера), которое позволяет указывать типы для контроллера и методов действий (рис. 34.12). Первым делом необходимо указать класс модели, где будет определен тип контроллера и методы действий. В раскрывающемся списке Model class (Класс модели) выберите `Inventory`. Далее понадобится указать класс контекста. Если он не выбран, то будет создан но-

вый такой класс. В списке Data context class (Класс контекста данных) выберите AutoLotEntities. Затем нужно указать, должны ли применяться асинхронные методы действий. Это зависит от потребностей разрабатываемого проекта. В текущем примере отметьте флажок Use async controller actions (Использовать асинхронные действия контроллера). По умолчанию отмеченный флажок Generate views (Генерировать представления) указывает на необходимость создания связанного представления для каждого метода действия. Флажок Reference script libraries (Ссылаться на библиотеки сценариев) приводит к включению проверки достоверности jQuery. Флажок Use a layout page (Использовать страницу компоновки) обсуждается позже в главе. Оставьте флажки Generate views, Reference script libraries и Use a layout page отмеченными и введите в поле Controller name (Имя контроллера) имя InventoryController (вместо находящегося там InventoriesController).

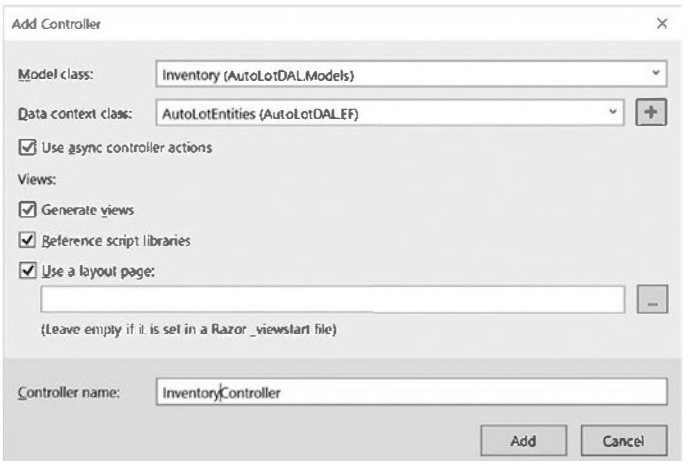


Рис. 34.12. Выбор модели, контекста и других аспектов

На заметку! В среде Visual Studio имеется много инструментов, поддерживающих разработку приложений MVC. Вы только что видели, как открыть диалоговое окно Add Controller, в котором применяется формирование шаблонов для создания контроллера и набора представлений (на основе выбора, произведенного в окне). Если вы щелкнете правой кнопкой мыши на папке Views, то в контекстном меню будет присутствовать пункт для добавления нового представления, выбор которого приведет к открытию диалогового окна, формирующего шаблон представления. Щелкнув правой кнопкой мыши на действии, можно добавить новое представление (которое будет помещено в папку Views/Controller и получит такое же имя, как у действия) или перейти к подходящему представлению. Все эти возможности опираются на соглашения, которые обсуждались ранее, так что если вы следуете правилам, то все будет в порядке.

В результате для вас было решено несколько задач. В папке Controllers создан класс InventoryController. Кроме того, в папке Views создана папка Inventory, а в нее добавлено пять представлений. Далее мы детально исследуем каждое из них.

Исследование шаблонных представлений

Чтобы получать доступ к новым представлениям, не прибегая к ручной корректировке URL, понадобится создать для них ссылки в меню. Откройте файл _Layout.cshtml (в Views/Shared) и отыщите в нем строку с @Html.ActionLink("Home", "Index","Home"). Скопируйте ее и вставьте ниже.

Модифицируйте вставленную строку следующим образом:

```
<li>@Html.ActionLink("Inventory", "Index", "Inventory")</li>
```

Прежде чем запускать программу вы должны изменить настройки запуска для проекта. Щелкните правой кнопкой мыши на проекте `CarLotMVC` в окне Solution Explorer и выберите в контекстном меню пункт Properties (Свойства). Перейдите к элементу Web в левой панели и в разделе Start Action (Начальное действие) выберите переключатель Specific page (Специфическая страница), оставив пустым поле справа (рис. 34.13). Это приведет к тому, что Visual Studio будет запускать ваш сайт в случае указания корневого URL (например, `http://localhost:14264`).

Теперь запустите программу, щелкните на ссылке Inventory и попробуйте просматривать, редактировать, создавать и удалять записи об автомобилях. Внешний вид представлений довольно скромен, но они полностью функциональны. Модернизацией пользовательского интерфейса мы займемся позже, а пока давайте рассмотрим контроллеры и действия более подробно.

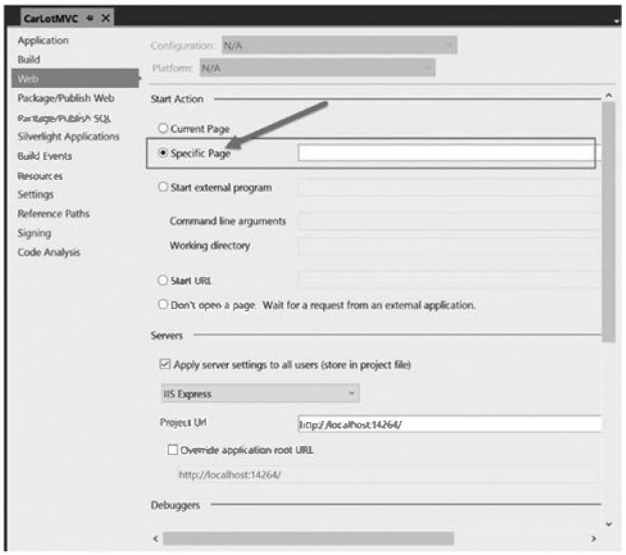


Рис. 34.13. Изменение начального действия

Контроллеры MVC

Откройте файл `InventoryController.cs`. Обратите внимание, что имя класса контроллера по соглашению заканчивается словом `Controller`. Кроме того, класс является производным от абстрактного класса `Controller`. В нем имеется набор методов (действий), таких как `Index()`, `Edit()` и т.д. Мы исследуем их по очереди вместе с атрибутами, которыми они декорированы. Наконец, метод `Dispose()` переопределен, чтобы можно было освобождать любые дорогостоящие ресурсы, задействованные контроллером.

Результаты действий

Действия обычно возвращают тип `ActionResult` (или `Task<ActionResult>` для асинхронных операций). Существует несколько производных от `ActionResult` типов; в табл. 34.8 перечислены часто используемые типы такого рода.

Таблица 34.8. Обычные классы, производные от `ActionResult`

Класс	Описание
`ViewResult` `PartialViewResult`	Возвращают в качестве веб-страницы представление (или частичное представление)
`RedirectResult` `RedirectToRouteResult`	Перенаправляют на другое действие
`JsonResult`	Возвращает клиенту сериализированный результат JSON result to the client
`FileResult`	Возвращает клиенту содержимое двоичного файла
`ContentResult`	Возвращает клиенту тип содержимого, определенный пользователем
`HttpStatusCodeResult`	Возвращает специфический код состояния HTTP

Использование хранилища `InventoryRepo`

В первой строке определения класса `InventoryController` создается новый экземпляр `AutoLotEntities`, как было указано при создании контроллера. Его понадобится заменить классом `InventoryRepo`. Добавьте в начало определения класса следующую переменную-член с экземпляром `InventoryRepo`:

```
private readonly InventoryRepo _repo = new InventoryRepo();
```

В переопределенном методе `Dispose()` освободите этот экземпляр:

```
protected override void Dispose(bool disposing)
{
  if (disposing)
  {
    db.Dispose();
    _repo.Dispose();
  }
  base.Dispose(disposing);
}
```

Действие `Index`

Действие `Index` получает все записи `Inventory` и возвращает данные представлению (представления рассматриваются в следующем разделе). Модифицируйте вызов для работы с классом `InventoryRepo`, а не напрямую с классом `AutoLotEntities`:

```
public async Task<ActionResult> Index()
{
  return View(await _repo.GetAllAsync());
}
```

В коде вызывается перегруженный метод `View()` базового класса `Controller`, который возвращает новый объект `ViewResult`. Когда имя представления не указано (как в данном случае), по соглашению представление именуется соответственно методу действия и помещается в папку с именем контроллера, т.е. `Views/Inventory/Index.cshtml`. Имя представления можно изменить, передавая методу `View()` желаемое имя. Например, чтобы назначить представлению имя `Foo.cshtml`, вот как необходимо вызвать метод `View()`:

```
return View("Foo", await _repo.GetAllAsync());
```

Действие Details

Метод действия `Details()` возвращает все подробности для одной записи `Inventory`. При этом URL в формате `http://mysite.com/Inventory/Details/5` будет отображен на метод действия `Details()` класса `InventoryController` с параметром по имени `id` и его значением 5. Модифицируйте метод с целью вызова в нем `_repo.GetOneAsync(id)` вместо взаимодействия с `AutoLotEntities` напрямую:

```
// GET: Inventory/Details/5
public async Task<ActionResult> Details(int? id)
{
  if (id == null)
  {
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
  }
  var inventory = await _repo.GetOneAsync(id);
  if (inventory == null)
  {
    return HttpNotFound();
  }
  return View(inventory);
}
```

В таком просто выглядящем методе есть пара интересных моментов. Вспомните из обсуждения маршрута, что параметр `id` является необязательным, поэтому URL вида `/Inventory/Details` будет корректно отображаться на данный метод. Однако если методу не передано значение `id`, то получить запись `Inventory` невозможно, так что метод возвращает код состояния HTTP 400 (`HttpStatusCode.BadRequest`). Запустив приложение и введя `Inventory/Details` (без части `id` в URL), вы получите экран ошибки, подобный показанному на рис. 34.14.

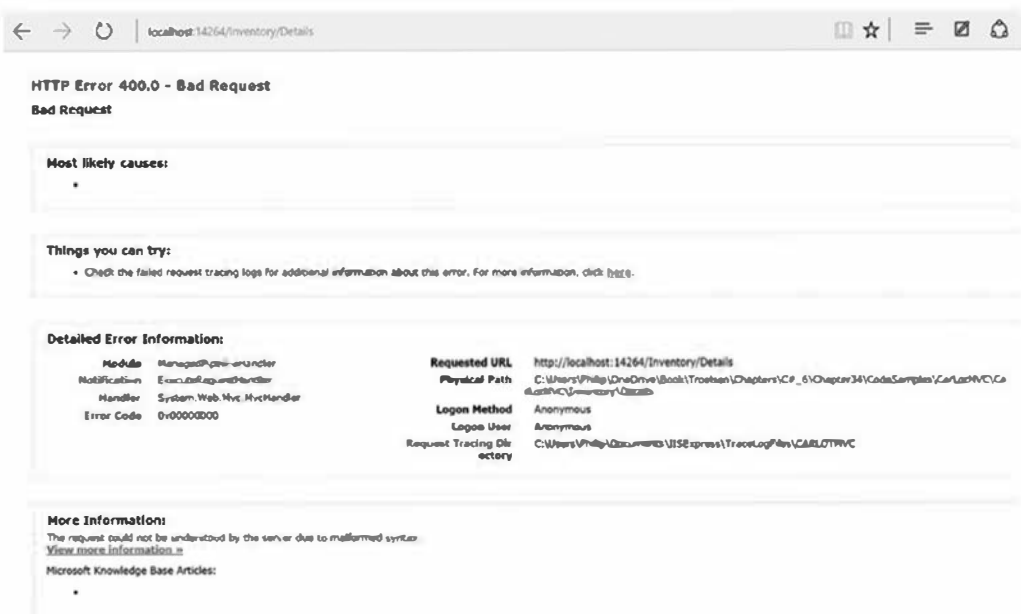


Рис. 34.14. Возвращение кода состояния HTTP 400

Кроме того, если запись `Inventory` найти не удалось, то метод действия возвращает код состояния HTTP 404.

В случае если с форматом URL все в порядке и запись `Inventory` найдена, то клиенту возвращается страница `Views/Inventory/Details.cshtml`.

Действие `Create`

Доступны две версии метода действия `Create()`: без параметров и принимающая объект `Inventory` в качестве параметра.

Версия `HttpGet`

Версия метода `Create()` без параметров обрабатывает запрос `HttpGet`, не обращаясь к базе данных (пользователь пока не создает новую запись), и возвращает представление `~/Views/Inventory/Create.cshtml`:

```
// GET: Inventory/Create
public ActionResult Create()
{
  return View();
}
```

Детали реализации будут вскоре раскрыты.

Версия `HttpPost`

Перегруженная версия метода `Create()`, которая принимает в качестве параметра объект `Inventory` (созданный посредством неявной привязки модели), снабжена двумя атрибутами уровня метода, `[HttpPost]` и `[ValidateAntiForgeryToken]`, и одним атрибутом уровня параметра, `[Bind]`. Эта версия выполняется, когда пользователь щелкнул на кнопке отправки формы `Create` (при условии, что все проверки достоверности на стороне клиента прошли успешно).

Привязка модели

Вспомните из главы 32, что привязка модели берет форму, строку запроса и т.д., а также пары "имя-значение", после чего пробует воссоздать объект указанного типа с применением рефлексии. Различают явную и неявную привязку модели. В каждом случае механизм привязки моделей пытается присвоить значения (из пар "имя-значение" в отправленных данных формы) соответствующим свойствам желаемого типа. Если ему не удается присвоить одно или более значений (скажем, из-за проблем с преобразованиями типов данных или ошибок проверки достоверности), то свойство `ModelState.IsValid` устанавливается в `false`. Если всем соответствующим свойствам успешно присвоены значения, то свойство `ModelState.IsValid` устанавливается в `true`. В дополнение к свойству `IsValid` свойство `ModelState` имеет тип `ModelStateDictionary` и содержит информацию об ошибках для каждого проблемного свойства, а также информацию об ошибках уровня модели. При желании можно добавить специфическое сообщение об ошибке для свойства:

```
ModelState.AddModelError("Name","Name is required");
```

Чтобы добавить сообщение об ошибке для целой модели, вместо имени свойства должно использоваться значение `string.Empty`:

```
ModelState.AddModelError(string.Empty, $"Unable to create record: {ex.Message}");
```

Для явной привязки модели необходимо вызвать метод `TryUpdateModel()`, передав ему экземпляр типа. Если привязка модели не удалась, то вызов `TryUpdateModel()` возвращает `false`. Например, метод `Create()` можно было бы реализовать следующим образом:

```
public async Task<ActionResult> Create()
{
  var inv = new Inventory();
  if (TryUpdateModel(inv))
  {
    // Сохранить данные.
  }
}
```

Для неявной привязки модели методу передается желаемый тип. Механизм привязки моделей выполняет с параметром ту же самую операцию, которая делалась посредством метода `TryUpdateModel()` в предыдущем примере:

```
public async Task<ActionResult> Create(Inventory inventory)
{
  if (ModelState.IsValid)
  {
    // Сохранить данные.
  }
}
```

Сравнение `HttpPost` и `HttpGet`

В то время как инфраструктура ASP.NET Web Forms по большому счету игнорирует разницу между `HttpGet` и `HttpPost`, эти методы HTTP в MVC применяются соответственно их назначению. Протокол HTTP определяет вызов `HttpGet` как запрос данных из сервера, а вызов `HttpPost` — как отправку данных специфическому ресурсу на обработку.

В инфраструктуре MVC любое действие *без* атрибута HTTP (вроде `HttpPost`) будет выполняться как операция `HttpGet`. Чтобы указать операцию `HttpPost` (действие, при котором данные будут отправлены и потенциально обновлены), метод действия потребуется декорировать атрибутом `[HttpPost]`.

Маркеры противодействия подделке

Одним из средств защиты от взлома является значение формы `AntiForgeryToken`, которая добавляется к представлениям. При поступлении запроса `HttpPost` маркер проверяется, если присутствует атрибут `[ValidateAntiForgeryToken]`. Хотя это не единственная мера защиты (тема безопасности выходит за рамки настоящей книги), каждая форма должна добавлять значение `AntiForgeryToken` и каждое действие `HttpPost` должно проверять его.

Атрибут `[Bind]`

Атрибут `[Bind]` в методах действий `Create()` и `Edit()` позволяет помещать свойства в "белый" или "черный" список, а также добавлять к ним префиксы (которые здесь не рассматриваются). Когда поля находятся в *"белом" списке*, только им будут присвоены значения во время привязки модели; это помогает защититься от чрезмерной отправки данных пользователем. Помещение в *"черный" список* исключает свойства из процесса привязки модели. В методе `Create()` все поля находятся в "белом" списке, но требуется отправлять только `Make`, `Color` и `PetName`. Удалите поля `CarId` и `Timestamp` из списка `Include`:

```
public async Task<ActionResult>
       Create([Bind(Include = "Make,Color,PetName")] Inventory inventory)
```

Написание кода

Если состояние модели не является допустимым, то метод отправляет пользователю представление Create с текущими данными, давая ему возможность исправить ошибочные данные. Если состояние модели допустимо и значения успешно записаны в хранилище, то метод действия возвращает результат вызова метода RedirectToAction(), который перенаправляет пользователя на действие Index контроллера Inventory. Перенаправление на представление Index после успешного сохранения предотвращает повторный щелчок пользователя на кнопке Create (Создать), который привел бы к двукратной отправке. Если в процессе сохранения возникла ошибка, то в ModelState добавляется новый экземпляр ModelError, а пользователю отправляется страница Create для повторной попытки. Обратите внимание, что начальный оператор предназначен для улучшения читабельности метода. Финальное изменение связано с использованием метода AddAsync() хранилища. Ниже показан результирующий код:

```
[HttpPost][ValidateAntiForgeryToken]
public async Task<ActionResult> Create([Bind(Include =
"Make,Color,PetName")] Inventory inventory)
{
  if (!ModelState.IsValid) { return View(inventory); }
  try
  {
    await _repo.AddAsync(inventory);
    return RedirectToAction("Index");
  }
  catch (Exception ex)
  {
    ModelState.AddModelError(string.Empty,
      $"Unable to create record: {ex.Message}");
    return View(inventory);
  }
}
```

Действие Edit

Подобно Create() существуют две версии метода действия Edit(): первая обрабатывает запрос HttpGet, а вторая — запрос HttpPost.

Версия HttpGet

Первая версия метода Edit() принимает параметр id и идентична версии HttpGet метода Details(). Удостоверьтесь в том, что код изменен для применения хранилища Inventory вместо работы напрямую с AutoLotEntities.

```
public async Task<ActionResult> Edit(int? id)
{
  if (id == null)
  {
     return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
  }
  Inventory inventory = await _repo.GetOneAsync(id);
  if (inventory == null)
  {
     return HttpNotFound();
  }
  return View(inventory);
}
```

Версия HttpPost

Как и метод действия Create(), вторая версия Edit() выполняется, когда пользователь щелкнул на кнопке отправки формы редактирования (при условии, что проверка достоверности на стороне клиента прошла успешно). Если состояние модели не является допустимым, то метод еще раз возвращает представление Edit, отправляя текущие значения для объекта Inventory. Если состояние модели допустимо, то объект Inventory передается хранилищу для попытки его записи. В дополнение к общей обработке ошибок (вроде используемой в методе Create()) понадобится также добавить проверку на предмет исключения DbUpdateConcurrencyException, которое будет сгенерировано, если другой пользователь обновил запись после того, как текущий пользователь загрузил ее в форму редактирования. Если все прошло успешно, то метод действия возвращает результат RedirectToAction, перенаправляя пользователя на действие Index контроллера Inventory.

Атрибут [Bind] можно оставить без изменений, т.к. нужны все значения из формы, но измените код для вызова метода AddAsync() хранилища:

```
[HttpPost][ValidateAntiForgeryToken]
public async Task<ActionResult> Edit(
  [Bind(Include = "CarId,Make,Color,PetName,Timestamp")] Inventory inventory)
{
  if (!ModelState.IsValid) { return View(inventory); }
  try
  {
    await _repo.SaveAsync(inventory);
    return RedirectToAction("Index");
  }
  catch (DbUpdateConcurrencyException)
  {
    ModelState.AddModelError(string.Empty,
        "Unable to save record. Another user updated the record.");
  }
  catch (Exception ex)
  {
    ModelState.AddModelError(string.Empty, $"Unable to save record: {ex.Message}");
  }
  return View(inventory);
}
```

На заметку! В главе 23 было показано, что класс DbUpdateConcurrencyException предоставляет много информации для разработчика. Из-за ограничений по объему его возможности в этой главе не демонстрируются.

Действие Delete

Имеются две версии метода действия Delete(): первая обрабатывает запрос HttpGet, а вторая — запрос HttpPost.

Версия HttpGet

Первая версия метода Delete() принимает параметр id и идентична версиям HttpGet методов Details() и Edit(). Удостоверьтесь в том, что код изменен для применения хранилища Inventory вместо работы напрямую с AutoLotEntities.

```
public async Task<ActionResult> Delete(int? id)
{
  if (id == null)
  {
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
  }
  Inventory inventory = await _repo.GetOneAsync(id);
  if (inventory == null)
  {
    return HttpNotFound();
  }
  return View(inventory);
}
```

Версия HttpPost

Вторая версия `Delete()` выполняется, когда пользователь щелкнул на кнопке отправки формы удаления. Автоматически сгенерированная версия этого метода принимает только `id` в качестве параметра, т.е. она имеет ту же самую сигнатуру, что и версия `HttpGet` метода. Поскольку существование двух методов с одним и тем же именем и одинаковыми сигнатурами невозможно, генерируемый метод именуется `DeleteConfirmed()` и снабжается атрибутом `[ActionName("Delete")]`. Библиотека `AutoLotDAL` проверяет наличие конфликтов параллелизма и для удаления записи кроме свойства `CarId` требует указания свойства `Timestamp`. Экземпляр `Inventory` также должен отображать ошибки модели. Для удовлетворения этих нужд просто измените параметр `id` типа `int` на параметр `inventory` типа `Inventory`. Такое изменение приведет к использованию неявной привязки модели для получения значений записи `Inventory` из запроса.

Для удаления записи необходимы только свойства `CarId` и `Timestamp`. Добавьте атрибут `[Bind]`, указав в списке `Include` строку `"CarId,Timestamp"`, чтобы поместить значения этих свойств в экземпляр `Inventory` и проигнорировать остальные свойства. Теперь, когда сигнатура метода стала отличаться от версии `HttpGet`, вы можете переименовать метод в `Delete()` и убрать атрибут `[ActionName]`. Наконец, модифицируйте метод для вызова метода `DeleteAsync()` хранилища `Inventory` и добавьте обработку ошибок (как поступали в версии `HttpPost` метода `Edit()`). Вот финальная версия кода:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Delete([Bind(Include="CarId,Timestamp")]
Inventory inventory)
{
  try
  {
    await _repo.DeleteAsync(inventory);
    return RedirectToAction("Index");
  }
  catch (DbUpdateConcurrencyException)
  {
    ModelState.AddModelError(string.Empty,
       "Unable to delete record. Another user updated the record.");
  }
  catch (Exception ex)
  {
    ModelState.AddModelError(string.Empty,
                             $"Unable to create record: {ex.Message}");
  }
  return View(inventory);
}
```

Если вы запустите приложение прямо сейчас и попробуете удалить запись `Inventory`, то оно не заработает, потому что представление не отправляет свойство `Timestamp`, а только `CarId`. Вскоре проблема будет устранена.

Метод `Dispose()`

В методе `Dispose()` понадобится освободить память, занимаемую переменной типа `AutoLotEntities` и переменной уровня класса для `AutoLotEntities`. Код метода `Dispose()` должен выглядеть следующим образом:

```
protected override void Dispose(bool disposing)
{
  if (disposing)
  {
    _repo.Dispose();
  }
  base.Dispose(disposing);
}
```

Заключительное слово по поводу контроллеров

Мы привели довольно большой объем информации, но лишь слегка коснулись поверхности того, что можно делать в контроллерах и методах действий MVC. Тем не менее, суть в том, что контроллеры — это просто классы C#. Они должны следовать соглашению об именовании вида *<Имя>*`Controller` (часть `Controller` имени отбрасывается инфраструктурой). Действия — это методы в классе контроллера, которые возвращают `ActionResult`. Методы действий могут быть декорированы атрибутом, который указывает, является ли метод `HttpPost` или `HttpGet` (по умолчанию), а все методы `HttpPost` должны проверять маркер `AntiForgery`. А теперь перейдем к представлениям.

Представления MVC

Представления в MVC отображают пользовательский интерфейс внутри сайтов MVC. Изначально представления MVC строились с применением механизма представлений Web Forms. В настоящее время у вас есть выбор между использованием механизма представлений Razor и механизма представлений Web Forms, хотя в большинстве сайтов MVC применяется Razor. Представления MVC спроектированы так, чтобы быть очень легковесными, и передают обработку серверной стороны контроллерам, а обработку клиентской стороны — коду JavaScript.

Механизм представлений Razor

Механизм представлений Razor создан как усовершенствование механизма представлений Web Forms и использует Razor в качестве основного языка. Razor — это синтаксис разметки шаблонов, который на стороне сервера интерпретируется в код C# (или VB.NET). Применение Razor в представлениях с HTML и CSS дает в результате более чистую и простую для чтения разметку. Наряду со многими улучшениями, появляющимися в случае использования Razor в представлениях, представления на основе Razor по-прежнему поддерживают все, что можно ожидать от веб-формы.

Синтаксис Razor

Первое отличие между механизмами представлений Web Forms и Razor в том, что код вводится с символом `@`. Интеллектуальные возможности Razor устраняют потребность в добавлении закрывающих символов `@`, тогда как Web Forms требует открывающего и закрывающего дескрипторов (`<% %>`).

Блоки операторов открываются символом @ и заключаются в фигурные скобки, как показано ниже (обратите внимание, что символ @ в качестве терминатора отсутствует):

```
@foreach (var item in Model)
{
}
```

Блоки кода могут содержать смесь разметки и кода. Строки, которые начинаются с дескриптора разметки, интерпретируются как HTML, а строки, начинающиеся с кода, трактуются как код:

```
@foreach (var item in Model)
{
  int x = 0;
  <tr></tr>
}
```

Строки также могут содержать смесь разметки и кода:

```
<h1>Hello, @username</h1>
```

Дескриптор <text> обозначает текст и должен быть визуализирован как часть разметки:

```
@item<text>-<text>
```

Символ @ перед именем переменной эквивалентен вызову Response.Write(), причем по умолчанию все значения в HTML закодированы. Если вы хотите выводить незакодированные данные (т.е. потенциально небезопасные), то должны применять синтаксис @Html.Raw(username).

Вспомогательные методы, функции и делегаты

В Razor разрешена инкапсуляция кода для увеличения продуктивности и сокращения объема повторяющегося кода. Функции можно размещать внутри файла в папке App_Code или делать статическими.

Вспомогательные методы HTML

Вспомогательные методы HTML в Razor визуализируют разметку. Существует множество встроенных вспомогательных методов, которые вы будете широко использовать, такие как применяемый ранее @Html.ActionLink(). Вы можете также строить собственные вспомогательные методы HTML, чтобы сократить (или полностью устранить) повторяющийся код. Например, может понадобиться вспомогательный метод, который выводит детали записи Inventory. Поместите следующий код вспомогательного метода HTML в начало представления Index.cshtml (после строки @model):

```
@using AutoLotDAL.Models
@helper ShowInventory(Inventory item)
{
  @item.Make<text>-</text>@item.Color<text>(</text>@item.PetName<text>)
            </text>
}
```

Внутри @foreach добавьте вызов ShowInventory():

```
@foreach (var item in Model)
{
  @ShowInventory(item)
  <!-- Для краткости остальной код не показан -->
}
```

Запустите приложение и перейдите к странице Index для Inventory; вы увидите детали каждой записи в форме одной строки. В реальном вспомогательном методе HTML было бы предусмотрено форматирование и согласование разметки с внешним видом всего сайта. Поскольку это просто демонстрация создания вспомогательного метода HTML, которая для сайта не требуется, закомментируйте строку с использованием комментария Razor, который выглядит как @* ... *@:

```
@*@ShowInventory(item)*@
```

Функции Razor

Функции Razor не возвращают разметку, а применяются для инкапсуляции кода с целью повторного использования. Чтобы увидеть их в действии, поместите показанную ниже функцию SortCars() после вспомогательного метода HTML внутри страницы представления Index.cshtml. Эта функция принимает список элементов Inventory и сортирует их по PetName:

```
@functions
{
  public IList<Inventory> SortCars(IList<Inventory> cars)
  {
    var list = from s in cars orderby s.PetName select s;
    return list.ToList();
  }
}
```

Модифицируйте оператор @foreach для вызова функции SortCars(). Переменная Model представляет реализацию IEnumerable<Inventory>, так что вы должны добавить еще и вызов метода ToList():

```
@foreach (var item in SortCars(Model.ToList()))
{
  <!-- Для краткости остальной код не показан. -->
}
```

Делегаты Razor

В последнем примере демонстрируются делегаты Razor, которые работают точно так же, как делегаты C#. Поместите следующий код делегата сразу после функции SortCars() в файле представления Index.cshtml. Этот делегат помечает символы полужирным.

```
@{
  Func<dynamic, object> b = @<strong>@item</strong>;
}
```

Чтобы посмотреть на него в действии, добавьте непосредственно после блока кода, определяющего делегат, такую строку:

```
This will be bold: @b("Foo")
```

Разумеется, пример тривиален, но в случае более сложного повторяющегося кода можно получить выигрыш от его помещения внутрь делегата. По существу здесь применимы все за и против, характерные для делегатов C#. После запуска приложения и перехода к странице Index для Inventory вы увидите, что слово Foo выделено полужирным. Теперь закомментируйте обращение к делегату, т.к. в оставшихся примерах в нем нет необходимости.

Заключительное слово по поводу Razor

Мы снова должны перейти к рассмотрению новой темы, поскольку для подробного описания всех аспектов Razor в книге недостаточно места. Дополнительные примеры использования Razor вы встретите далее в главе. Выше были даны основы, необходимые для дальнейшего изучения этого механизма представлений.

Компоновки

Подобно тому, как инфраструктура Web Forms поддерживает мастер-страницы, MVC поддерживает компоновки. Представления MVC могут быть основаны на главной компоновке, что придает сайту унифицированный внешний вид и поведение. Вспомните, что в диалоговом окне Add Controller, показанном на рис. 34.12, имеется флажок Use a layout page (Использовать страницу компоновки). Оставьте текстовое поле под флажком пустым, если страница компоновки устанавливается в файле `_ViewStart.cshtml`. Также вспомните, что в папке `Views` присутствует файл по имени `_ViewStart.cshtml`. Откройте его, чтобы просмотреть содержимое:

```
@{
  Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Файл содержит один блок кода Razor, который устанавливает для компоновки специфический файл. Это запасное значение; если компоновка в представлении не указана, то данный файл будет по умолчанию применяться в качестве компоновки для представления.

Перейдите в папку `Views/Shared` и откройте файл `_Layout.cshtml`. Он является полноценным файлом HTML, укомплектованным дескрипторами `<head>` и `<body>`, а также разметкой HTML и вспомогательными методами HTML из Razor. Подобно мастер-страницам Web Forms страница `_Layout.cshtml` — это основа того, что будет отображаться пользователю при визуализации представлений (использующих страницу `_Layout.cshtml`).

При работе с компоновками важно помнить о существовании двух элементов: тело и области. В тело будет вставляться код представления, когда представление и компоновка объединяются. Место размещения содержимого страницы представления в компоновке управляется следующей строкой кода Razor:

```
@RenderBody()
```

Области — это разделы страницы компоновки, которые могут заполняться во время выполнения. Они могут быть обязательными или необязательными и определяются внутри страницы компоновки с помощью `RenderSection()`. В первом параметре указывается имя области, а во втором — признак обязательности реализации области представлением. Приведенная ниже строка кода в `_Layout.cshtml` создает область по имени `scripts`, которая является необязательной для представления:

```
@RenderSection("scripts", required: false)
```

Области могут также помечаться как обязательные путем передачи во втором параметре значения `true`. Например, если нужно создать новую обязательную область по имени `Header`, то должен применяться такой код:

```
@RenderSection("Header",required: true)
```

Для визуализации области в представления используется блок Razor под названием `@section`. Следующие блоки кода внутри страницы `Edit.cshtml`, находящейся в папке `Views/Inventory`, добавляют к визуализируемой странице пакет проверки достоверности jQuery:

```
@section Scripts {
  @Scripts.Render("~/bundles/jqueryval")
}
```

Использование специфической страницы компоновки

В дополнение к применению стандартной страницы компоновки представлениям можно указывать на необходимость использования специфической страницы. Скопируйте файл `_Layout.cshtml` под новым именем `_LayoutNew.cshtml`. Откройте новый файл и поместите сразу после дескриптора `<body>` такие строки:

```
<div class="jumbotron">
  <h1>My MVC Application</h1>
</div>
```

Теперь откройте файл `Index.cshml` из папки `Views/Inventory` и добавьте после строки `ViewBag` код `Layout="~/Views/Shared/_LayoutNew.cshtml"`. Модифицированный блок кода должен выглядеть следующим образом:

```
@{
  ViewBag.Title = "Index";
  Layout = "~/Views/Shared/_LayoutNew.cshtml";
}
```

Здесь представление инструктируется относительно применения нового файла в качестве компоновки. Запустите приложение и перейдите к странице `Index` для `Inventory`; вы увидите экран, показанный на рис. 34.15.

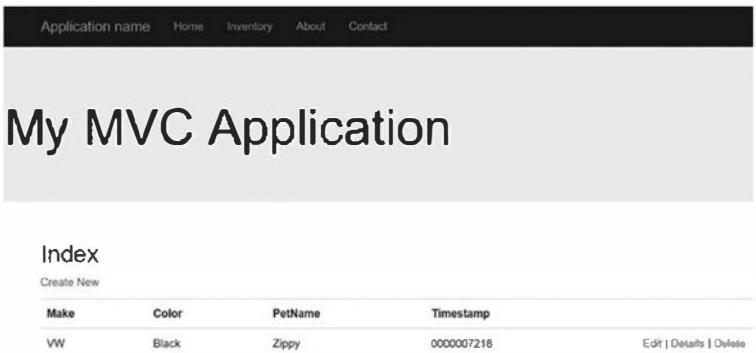


Рис. 34.15. Представление `Index` с новой компоновкой

Частичные представления

Частичные представления полезны для инкапсуляции пользовательского интерфейса, позволяя сократить (или полностью устранить) повторяющийся код. Поскольку представления Razor не унаследованы от класса `System.Web.Page` (и директивы вроде `Page` не существует), единственная *формальная* разница между частичным представлением и нормальным представлением заключается в том, что частичное представление визуализируется из метода действия. Полное представление (возвращаемое из контроллера посредством метода `View()`) будет использовать страницу компоновки, указанную либо в качестве стандартной в файле `_ViewStart.cshtml`, либо заданную с помощью оператора `Layout` в коде Razor. Представление, визуализируемое с применением метода `PartialView()` (или вспомогательного метода HTML по имени `Partial()`), не использует стандартную компоновку, но все же задействует компоновку, если она указана посредством оператора `Layout` в коде Razor.

Чтобы удостовериться в сказанном, откройте файл `InventoryController.cs` и измените метод действия `Index()` для возвращения частичного представления вместо полного:

```
public async Task<ActionResult> Index()
{
  return PartialView(await _repo.GetAllAsync());
}
```

Откройте страницу `Index.cshtml` и либо удалите ранее добавленную строку `Layout=`, либо закомментируйте ее:

```
@{
  ViewBag.Title = "Index";
  // Layout = "~/Views/Shared/_LayoutNew.cshtml";
}
```

Запустите приложение и перейдите на страницу `Index` для `Inventory`. Вы увидите те же самые данные, что и ранее, но без какой-либо компоновки (рис. 34.16).

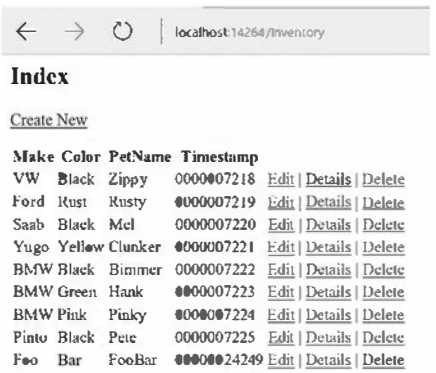


Рис. 34.16. Страница `Index` для `Inventory`, визуализированная как частичное представление

Измените метод действия `Index()`, чтобы снова вызывать метод `View()`, а не `PartialView()`. Можете оставить закомментированной строку `//Layout = "~/Views/Shared/_LayoutNew.cshtml"`, и представление `Index` вернется к применению стандартной компоновки.

Вдобавок к визуализации представления из метода действия с помощью вызова `PartialView()` вы можете поместить частичное представление внутрь другого представления, используя вспомогательный метод HTML, что похоже на загрузку пользовательского элемента управления в Web Forms. Следующий блок кода Razor в файле `_Layout.cshtml` создает пользовательский интерфейс для входа, отображаемый на каждой странице:

```
@Html.Partial("_LoginPartial")
```

Отправка данных представлению

Как обсуждалось ранее в главе, шаблон MVC полагается на определенный уровень разделения ответственности. Контроллер отправляет данные представлению, представление запрашивает действия, а модели передаются в качестве данных для приложения. Вы уже видели, как представления запрашивают действия, но мы еще не обсуждали, каким образом помещать данные (модели) внутрь представлений.

Объекты ViewBag, ViewData и TempData

Объекты `ViewBag`, `ViewData` и `TempData` — это механизмы для отправки небольших объемов данных представлению. Примером может служить строка в начале каждого представления `Inventory`, устанавливающая свойство `ViewBag.Title`; вот как она выглядит в представлении `Index.cshtml`:

```
@{
  ViewBag.Title = "Index";
}
```

Свойство ViewBag.Title применяется для отправки заголовка представления компоновке для использования в следующей строке внутри _Layout.cshtml:

```
<title>@ViewBag.Title - My ASP.NET Application</title>
```

В табл. 34.9 кратко описаны три механизма передачи данных из контроллера представлению (помимо свойства Model, рассматриваемого в следующем разделе) или из представления представлению.

Таблица 34.9. Способы передачи данных представлению

Объект транспортировки данных	Описание
TempData	Недолговечный объект, который работает только во время текущего запроса и запроса, следующего за ним
ViewData	Словарь, позволяющий хранить пары "имя-значение", например, ViewData["Title"] = "Foo"
ViewBag	Динамическая оболочка для словаря ViewData, например, ViewBag.Title = "Foo"

Строго типизированные представления и модели представлений

Для более крупных объемов данных (таких как все записи Inventory, применяемые в представлении Index.cshtml) используется свойство Model. Взгляните на первую строку в файле Index.cshtml, которая отражает, что представление является строго типизированным, а в качестве типа указан IEnumerable<Inventory>:

```
@model IEnumerable<AutoLotDAL.Models.Inventory>
```

Атрибут @model задает тип представления. Для доступа к нему в остальной части представления применяется свойство Model. Обратите внимание на заглавную букву M в свойстве Model и строчную букву m в атрибуте @model. При ссылке на данные, содержащиеся в представлении, используется Model, как в показанной ниже строке, в которой осуществляется проход по всем записям Inventory:

```
@foreach (var item in Model)
{
  // Какие-то интересные действия.
}
```

Представление Index

Наступило время заняться исследованием действительных представлений. Откройте файл Index.cshtml, в котором находится следующий код:

```
@Html.DisplayNameFor(model => model.Make)
```

Здесь применяется вспомогательный метод HTML под названием DisplayNameFor() для вывода отображаемого имени (в виде простого текста) поля модели, указанного в лямбда-выражении. В этом примере получается отображаемое имя для свойства Make объекта Inventory.

Для свойств Make и Color все работает хорошо, но свойство PetName отображается как PetName. Взамен его желательно отображать как Pet Name. Можно было бы изменить код и жестко закодировать метку Pet Name, но это решит проблему только для отдельного представления. В любых других представлениях, нуждающихся в отображении свойства PetName, пришлось бы аналогично жестко кодировать указанную метку.

Более эффективный подход предусматривает использование для установки отображаемого имени аннотаций данных в модели. Вскоре это будет сделано.

Так как пользователи не желают видеть значения отметок времени, удалите заголовок для свойства `Timestamp`, включая разметку и код Razor. Также удалите код для этого свойства из цикла `foreach`.

Внутри цикла `foreach` значения для каждого элемента отображаются с применением еще одного вспомогательного метода HTML, `DisplayFor()`. Он выясняет тип данных и представляет значение на основе стандартного шаблона для этого типа данных. В настоящем примере поля данных являются строковыми, поэтому вспомогательный метод HTML просто отображает значения.

Аннотации данных `Display`

В дополнение к аннотациям данных, используемым для определения модели, по которой инфраструктура Entity Framework может создать базу данных (см. главу 23), существуют аннотации данных, предназначенные для определения отображаемых свойств. Хотя вы можете самостоятельно добавить их к классам моделей, в случае применения EF для создания классов моделей из имеющейся базы данных любые внесенные вами изменения будут перезаписаны, если понадобится повторно сгенерировать модели. Чтобы предотвратить это, аннотации данных можно поместить в другой файл.

Перейдите в папку `Models` проекта `AutoLotDAL` и добавьте новую папку по имени `MetaData`. Поместите в эту папку новый файл класса `InventoryMetaData.cs`. Сделайте класс открытым и добавьте свойство `PetName` типа `string`. Снабдите это свойство атрибутом `[Display(Name="Pet Name")]`. Код класса должен выглядеть следующим образом:

```
public class InventoryMetaData
{
  [Display(Name="Pet Name")]
  public string PetName;
}
```

В данном файле не находится полное определение класса; он будет использоваться только для загрузки назначенных вами атрибутов. Следовательно, вам не придется добавлять к данному свойству синтаксис `get/set`; на самом деле вы и не должны делать это. Может возникнуть вопрос: как инфраструктура узнает, что данный класс предоставляет атрибуты для класса `Inventory`? В настоящий момент никак. Вам необходимо добавить к классу `Inventory` атрибут уровня класса, чтобы инфраструктуре было известно, что этот класс содержит дополнительные атрибуты для нее. Ниже вы внесете нужное изменение.

Откройте файл `InventoryPartial.cs` из папки `Models/Partials` и добавьте к классу атрибут `[MetadataType]`:

```
[MetadataType(typeof(InventoryMetaData))]
public partial class Inventory
{
  public override string ToString() =>
    $"{this.PetName ?? "**No Name**"} is a {this.Color} {this.Make}
        with ID {this.CarId}.";
}
```

Запустив приложение и перейдя к странице `Index` для `Inventory`, вы увидите, что метка `PetName` отображается как Pet Name безо всякого изменения кода в представлении.

Обновление представления возможностями Bootstrap

На следующем шаге мы несколько оживим представление за счет применения Bootstrap.

Обновление заголовка

Первым делом обновите заголовок страницы. Сгенерированное представление имеет заголовок Index, что не особо информативно. Удалите строку `<h2>Index</h2>` и вместо нее поместите показанную ниже разметку, которая создает небольшую затененную область, содержащую заголовок страницы:

```
<div class="well well-sm"><h1>Available Inventory</h1></div>
```

Результат представлен на рис. 34.17.

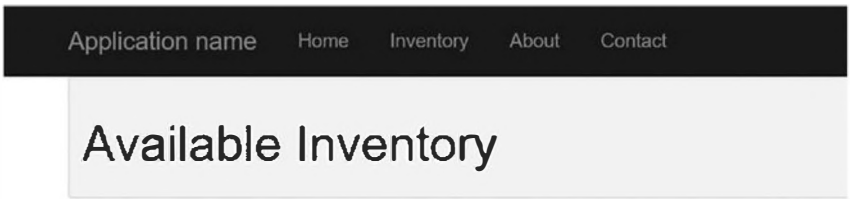


Рис. 34.17. Новый заголовок страницы

Обновление таблицы

Далее будет обновлена таблица. Стандартный класс таблицы в Bootstrap добавляет ряд простейших элементов дизайна, в том числе разделительные линии. В табл. 34.10 описаны дополнительные встроенные стили, которые можно добавлять.

Таблица 34.10. Стили для таблицы

Стиль	Описание
.table	Это базовый стиль таблицы. Он добавляет разделительные линии и небольшие отступы
.table-striped	Добавляет выделение строк таблицы полосами. Отсутствует в Internet Explorer 8
.table-bordered	Добавляет границы вокруг каждой ячейки в таблице
.table-hover	Добавляет легкую подсветку к таблице
.table-condensed	Вдвое сокращает отступы в ячейках
.table-responsive	Делает таблицу более быстрореагирующей на мобильных устройствах

Добавьте в атрибут class дескриптора `<table>` все стили кроме table-condensed, а в следующую строку разметки поместите дескриптор `<caption>` с текстом Vehicle List:

```
<table class="table table-striped table-responsive table-hover table-bordered">
  <caption>Vehicle List</caption>
```

Результирующий пользовательский интерфейс показан на рис. 34.18.

Vehicle List

Make	Color	Pet Name	
VW	Black	Zippy	Edit | Details | Delete
Ford	Rust	Rusty	Edit | Details | Delete
Saab	Black	Mel	Edit | Details | Delete
Yugo	Yellow	Clunker	Edit | Details | Delete
BMW	Black	Bimmer	Edit | Details | Delete
BMW	Green	Hank	Edit | Details | Delete
BMW	Pink	Pinky	Edit | Details | Delete
Pinto	Black	Pete	Edit | Details | Delete
Foo	Bar	FooBar	Edit | Details | Delete

Рис. 34.18. Обновленный пользовательский интерфейс таблицы

Использование библиотеки GlyphIcons

Поставляемая вместе с Bootstrap библиотека `GlyphIcons` содержит набор значков и удобна для добавления визуальных подсказок к ссылкам и кнопкам. Мы собираемся добавить значки ко всем ссылкам в представлении `Index`, но сначала полезно исследовать еще один вспомогательный метод HTML, который называется `@Url.Action()`. Вспомогательный метод `@Url.Action()` возвращает *только* порцию URL, тогда как `Html.ActionLink()` создает полную разметку для дескриптора `<a>`.

Замените вызов `@Html.ActionLink("Create")` такой строкой:

```
<a href="@Url.Action("Create")">Create a new Car</a>
```

В итоге создается тот же самый URL, как и при вызове `ActionLink`, но теперь вы имеете полный контроль над разметкой. Добавьте значок "плюс" из `GlyphIcons` к содержимому дескриптора `<a>`:

```
<a href="@Url.Action("Create")">
  <span class="glyphicon glyphicon-plus"></span> Create a new Car
</a>
```

Обновленная ссылка Create a new Car (Создать новую запись об автомобиле) приведена на рис. 34.19.

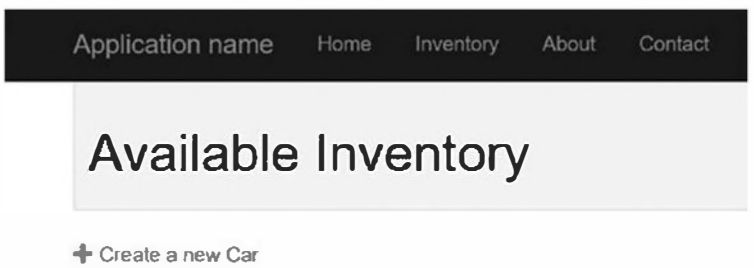


Рис. 34.19. Ссылка Create a new Car со значком "плюс" из `GlyphIcons`

Модифицируйте ссылки Edit (Редактировать), Details (Подробности) и Delete (Удалить) в таблице, добавив значки `Edit`, `List-Alt` и `Trash` из `GlyphIcons`. Ниже показан код:

```
<a href="@Url.Action("Edit", new {id = item.CarId})">
  <span class="glyphicon glyphicon-edit"></span> Edit
</a>
 | 
<a href="@Url.Action("Details", new {id = item.CarId})">
  <span class="glyphicon glyphicon-list-alt"></span> Details
</a>
 | 
<a href="@Url.Action("Delete", new {id = item.CarId})">
  <span class="glyphicon glyphicon-trash"></span> Delete
</a>
```

Финальная страница представлена на рис. 34.20.

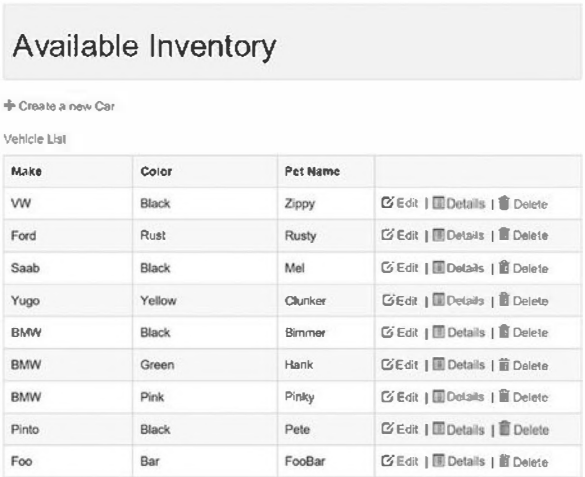


Рис. 34.20. Финальное представление `Index`

Представление `Details`

Представление `Details.cshtml` требует не особенно много изменений. Подобно `Index.cshtml` в представлении `Details.cshtml` применяются вспомогательные методы `DisplayNameFor()` и `DisplayFor()`. Так как библиотека `AutoLotDAL` была модифицирована, измененная метка `Pet Name` будет в нем актуальна. Таким образом, понадобится лишь удалить колонку `Timestamp`.

Обновление представления с использованием Bootstrap

Начните с удаления строк `<h2>Details</h2>` и `<h4>Inventory</h4>` и замены их следующей разметкой:

```
<div class="well well-sm"><h1>Inventory Details</h1></div>
```

Затем модифицируйте ссылки Edit и Back To List (Назад в список) для применения подходящих значков `GlyphIcons` и добавьте ссылку Delete. Вот обновленная разметка (результирующая страница показана на рис. 34.21):

```
<a href="@Url.Action("Edit", new {id = Model.CarId})">
  <span class="glyphicon glyphicon-edit"></span> Edit
</a>
 | 
```

```
<a href="@Url.Action("Delete", new {id = Model.CarId})">
  <span class="glyphicon glyphicon-trash"></span> Delete
</a>
 | 
<a href="@Url.Action("Index", new {id = Model.CarId})">
  <span class="glyphicon glyphicon-list"></span> Back to List
</a>
```

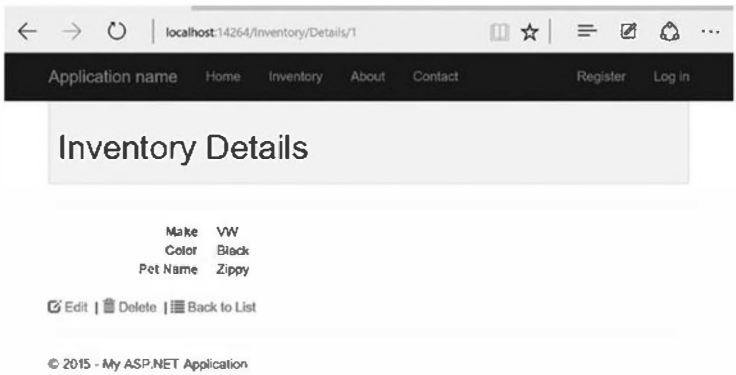


Рис. 34.21. Обновленное представление `Details`

Представление `Create`

Исследуя код в этом представлении, вы заметите, что в нем используются еще два вспомогательных метода HTML: `Html.LabelFor()` и `Html.EditorFor()`. Вспомогательный метод `EditorFor()` создает поле ввода на основе типа данных свойства, указанного в лямбда-выражении. Например, следующая строка:

```
@Html.EditorFor(model =>
    model.Make, new { htmlAttributes = new { @class = "form-control" } })
```

приводит к созданию такой разметки:

```
<input name="Make" class="form-control text-box single-line" id="Make"
type="text" value="" data-val-length-max="50" data-val-length="The field
Make must be a string with a maximum length of 50." data-val="true">
```

Прежде чем двигаться дальше, давайте разберемся с приведенной разметкой. В атрибутах `name` и `id` элемента HTML указано имя свойства, в атрибуте `type` — тип данных свойства, а в атрибуте `class` — комбинация результата выполнения вспомогательного метода HTML и дополнительных атрибутов HTML, добавленных посредством этого вспомогательного метода. Атрибут `value` устанавливается в значение свойства, которым здесь является пустая строка, поскольку это новый экземпляр `Inventory`.

Вспомогательный метод `LabelFor` создает элемент управления `Label`. Например, взгляните на показанную ниже строку:

```
@Html.LabelFor(model => model.Make, htmlAttributes:
  new { @class = "control-label col-md-2" })
```

Она создает следующую разметку, включая автоматически добавленный атрибут `for` (с именем свойства):

```
<label class="control-label col-md-2" for="Make">Make</label>
```

Вспомогательный метод HTML по имени BeginForm()

Вспомогательный метод HTML по имени BeginForm() создает дескриптор <form> в выводе HTML. По умолчанию свойство action формы установлено в текущий URL, а свойство method — в post (оба свойства допускают настройку за счет применения разных перегруженных версий метода BeginForm()). Блок using кода Razor будет инкапсулировать все, что указано между фигурными скобками, внутри открывающего и закрывающего дескрипторов HTML. Например, поместите в представление такой блок кода Razor:

```
@using (Html.BeginForm())
{
  <input name="foo" id="foo" type="text"/>
}
```

Он создаст в разметке HTML дескриптор <form> с атрибутом action, установленным в тот же самый URL, который привел пользователя к данному представлению. Например, если URL для запроса HttpGet был Inventory/Create, то вспомогательный метод Html.BeginForm() сгенерирует следующую разметку:

```
<form action="/Inventory/Create" method="post">
  <input name="foo" id="foo" type="text"/>
</form>
```

Маркер противодействия подделке

Вспомните, что атрибут [ValidateAntiForgeryToken] добавляется ко всем версиям HttpPost методов действий. Он обеспечивает проверку маркера противодействия подделке, отправляемого как часть значений формы, поэтому в блок кода Razor, где вызывается метод BeginForm(), необходимо добавить такой маркер. Это делается посредством вспомогательного метода HTML по имени AntiForgeryToken(), вызов которого уже присутствует в шаблонных формах, нуждающихся в маркере противодействия подделке. Добавить его самостоятельно можно с помощью простого синтаксиса:

```
@Html.AntiForgeryToken()
```

Обновление представления с использованием Bootstrap

Завершите изменения Create.cshtml, удалив строки <h2>Create</h2> и <h4>Inventory</h4> и заменив их показанной ниже разметкой:

```
<div class="well well-sm"><h1>Add Inventory</h1></div>
```

Добавьте к кнопке Create значок "плюс" из GlyphIcons:

```
<button type="submit" class="btn btn-default">
  <span class="glyphicon glyphicon-plus"></span> Create
</button>
```

Поместите ссылку Back To List рядом с кнопкой Create (вместо ее нахождения в нижнем элементе , как было сформировано шаблоном) и снабдите ссылку значком GlyphIcons с изображением списка. Вот модифицированная разметка:

```
 | 
<a href="@Url.Action("Index")">
  <span class="glyphicon glyphicon-list"></span> Back to list
</a>
```

Окончательная разметка для раздела выглядит следующим образом:

```
<div class="form-group">
  <div class="col-md-offset-2 col-md-10">
```

```
    <button type="submit" class="btn btn-default">
      <span class="glyphicon glyphicon-plus"></span> Create
    </button>
     | 
    <a href="@Url.Action("Index")">
      <span class="glyphicon glyphicon-list"></span> Back to list</a>
  </div>
</div>
```

Внешний вид представления показан на рис. 34.22.

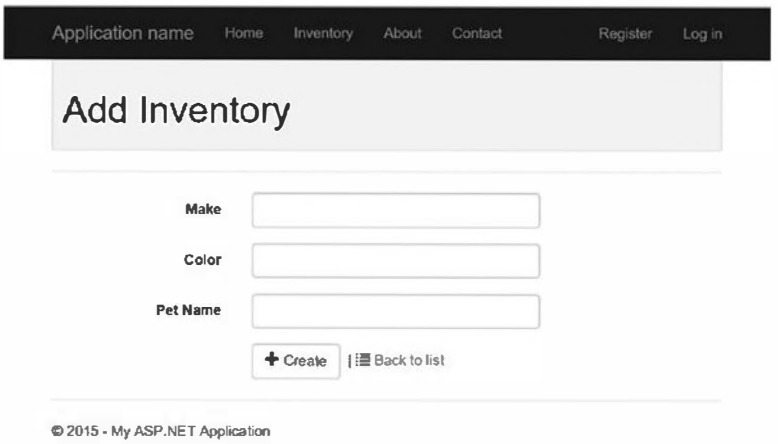


Рис. 34.22. Представление для добавления записи об автомобиле

Представление `Delete`

Шаблонное представление `Delete` **отображает поле** `Timestamp`, **которое имеет мало смысла для пользователя (и потенциально запутывает). Удалите дескрипторы** `<dt>` **и** `<dd>` **для поля** `Timestamp` **(поле** `Timestamp` **будет добавлено как скрытое значение).**

Скрытые значения

В дополнение к значению `CarId` **методы** `Delete()/DeleteAsync()` **объекта** `InventoryRepo` **требуют передачи значения** `Timestamp` **или объекта** `Inventory` **с заполненными полями** `CarId` **и** `Timestamp`. **Для отправки** `CarId` **предусмотрен URL (например,** `/Inventory/Delete/46`**), но такие значения лучше передавать через данные формы в теле запроса HTTP, а не в виде значений строки запроса в URL.**

Для этого внутри блока `BeginForm()` **кода Razor будет применяться еще один вспомогательный метод HTML:** `HiddenFor()`. **Он создает скрытое значение формы для свойства, указанного в лямбда-выражении. Вот код:**

```
@Html.HiddenFor(x => x.CarId)
@Html.HiddenFor(x => x.Timestamp)
```

В результате создается следующая разметка HTML:

```
<input name="CarId" id="CarId" type="hidden"
       value="46" data-val-required="The CarId field is required."
       data-val-number="The field CarId must be a number." data-val="true">
<input name="Timestamp" id="Timestamp" type="hidden" value="AAAAAAABAdE=">
```

Сводка по проверке достоверности

Несмотря на то что проверка достоверности пока не затрагивалась, добавьте в блок `BeginForm()` кода Razor вызов вспомогательного метода HTML по имени `ValidationSummary()`:

```
@Html.ValidationSummary(true, "", new { @class = "text-danger" })
```

Обновление представления с использованием Bootstrap

Удалите строки `<h2>Delete</h2>` и `<h4>Inventory</h4>` и замените их такой разметкой:

```
<div class="well well-sm"><h1>Delete</h1></div>
```

Модифицируйте кнопку Delete, добавив значок `Trash` из `GlyphIcons`:

```
<button type="submit" class="btn btn-default">
  <span class="glyphicon glyphicon-trash"></span> Delete
</button>
```

Обновите ссылку Back To List для применения в ней значка `GlyphIcons` с изображением списка:

```
 | 
<a href="@Url.Action("Index")">
  <span class="glyphicon glyphicon-list"></span> Back to list
</a>
```

Завершенная разметка для раздела `<div>` выглядит следующим образом:

```
<div class="form-actions no-color">
  <button type="submit" class="btn btn-default">
    <span class="glyphicon glyphicon-trash"></span> Delete
  </button>
   | 
  <a href="@Url.Action("Index")"><span class="glyphicon glyphicon-list">
  </span> Back to list</a>
</div>
```

Обновленное представление показано на рис. 34.23.

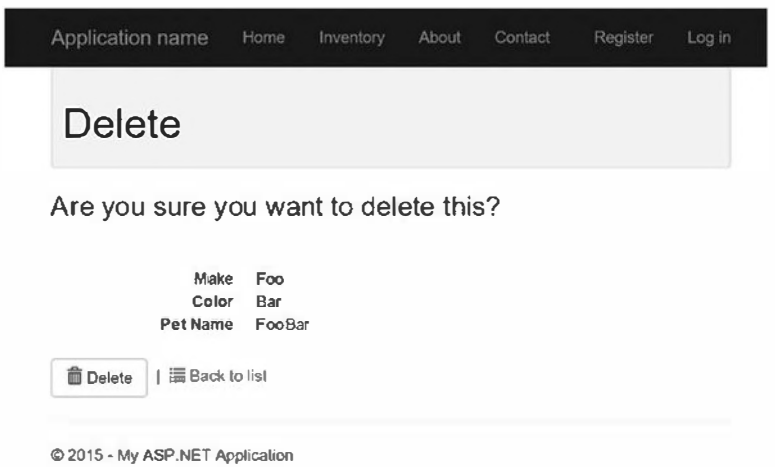


Рис. 34.23. Обновленное представление `Delete`

Представление Edit

Шаблонное представление Edit.schtml отображает поле Timestamp, которое имеет мало смысла для пользователя (и потенциально запутывает). Удалите дескриптор для поля Timestamp и добавьте вызов HiddenFor() сразу после строки HiddenFor(model=>model.CarId):

```
@Html.HiddenFor(model => model.Timestamp)
```

Обновление представления с использованием Bootstrap

Удалите строки <h2>Edit</h2> и <h4>Inventory</h4> и замените их такой разметкой:

```
<div class="well well-sm"><h1>Edit</h1></div>
```

Модифицируйте кнопку Save (Сохранить), добавив значок Save из GlyphIcons:

```
<button type="submit" class="btn btn-default">
  <span class="glyphicon glyphicon-save"></span> Save</button>
```

Обновите ссылку Back To List для применения в ней значка GlyphIcons с изображением списка и поместите ее рядом с кнопкой Save:

```
 | 
<a href="@Url.Action("Index")">
  <span class="glyphicon glyphicon-list"></span> Back to list</a>
```

Завершенная разметка для раздела выглядит следующим образом:

```
<div class="form-group">
  <div class="col-md-offset-2 col-md-10">
    <button type="submit" class="btn btn-default">
      <span class="glyphicon glyphicon-save"></span> Save
    </button>
     | 
    <a href="@Url.Action("Index")">
      <span class="glyphicon glyphicon-list"></span> Back to list</a>
  </div>
</div>
```

Финальный пользовательский интерфейс представления Edit приведен на рис. 34.24.

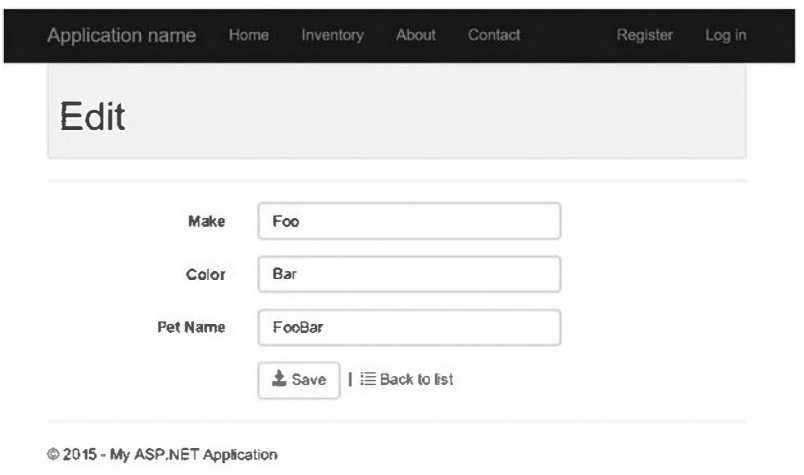


Рис. 34.24. Представление Edit

Проверка достоверности

Приложения MVC поддерживают два уровня проверки достоверности: серверной стороны и клиентской стороны. Проверка достоверности серверной стороны осуществлялась ранее в главе, когда производилось добавление ошибок в `ModelState` как дополнение к ошибкам, которые возникали в процессе привязки модели (из-за преобразований типов данных, отказов, связанных с аннотациями данных, или каких-то других причин). Проверка клиентской стороны реализуется с помощью кода JavaScript, что вскоре будет показано.

Отображение ошибок

Ошибки `ModelState` отображаются в пользовательском интерфейсе посредством вспомогательных методов HTML с названиями `ValidationMessageFor()` и `ValidationSummary()`. Метод `ValidationSummary()` будет показывать ошибки `ModelState`, которые не связаны со свойствами, а также ошибки, относящиеся к свойствам (при условии, что для параметра `excludePropertyErrors` передано значение `false`). Обычно ошибки, связанные со свойствами, отображаются рядом со свойствами, а ошибки, не имеющие отношения к свойствам, выводятся с использованием `ValidationSummary()`. Например, следующая строка кода (в представлениях `Create`, `Update` и `Delete`) обеспечит отображение всех ошибок модели шрифтом красного цвета, но не ошибок, связанных со свойствами:

```
@Html.ValidationSummary(true, "", new { @class = "text-danger" })
```

Для отображения ошибок, относящихся к свойствам, применяется вызов вспомогательного метода `ValidationMessageFor()` рядом со свойствами в представлении:

```
@Html.ValidationMessageFor(model => model.Make, "", new { @class = "text-danger" })
```

Вот результирующая разметка:

```
<span class="field-validation-valid text-danger"
      data-valmsg-replace="true" data-valmsg-for="Make"></span>
```

Чтобы увидеть это в действии, сначала потребуется отключить проверку достоверности клиентской стороны, добавленную к странице стандартным шаблоном. Откройте файл `Create.cshtml` и закомментируйте вызов метода `Render()` для пакета проверки достоверности jQuery, находящийся в конце файла:

```
@section Scripts {
  @*@Scripts.Render("~/bundles/jqueryval")*@
}
```

Откройте файл `InventoryController.cs` и модифицируйте первую часть версии `HttpPost` метода действия `Create()`, как показано ниже:

```
if (!ModelState.IsValid)
{
  ModelState.AddModelError(string.Empty,
    "An error occurred in the data.  Please check all values and try again.");
  return View(inventory);
}
```

Запустите проект, перейдите на страницу `Create` и введите в поле Make строку, содержащую более 50 символов. После щелчка на кнопке Save значения формы отправляются методу `Create()`. Во время привязки модели производится проверка достоверности и возникает ошибка, т.к. свойство `Make` снабжено атрибутом `[StringLength(50)]`. Результирующий экран должен выглядеть примерно так, как на рис. 34.25.

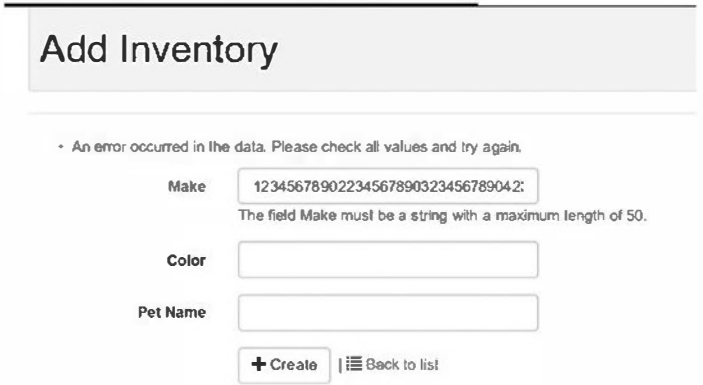


Рис. 34.25. Отображение результатов проверки достоверности на стороне сервера

Проверка достоверности клиентской стороны

Проверка достоверности клиентской стороны обрабатывается посредством библиотек jQuery (`jquery-2.1.4.min.js`), проверки достоверности jQuery (`jquery.validate.min.js`) и ненавязчивой проверки достоверности jQuery (`jquery.validate.unobtrusive.min.js`). Библиотеки проверки достоверности jQuery добавляют атрибуты данных HTML 5.0, предназначенные для проверки пользовательского ввода. Инфраструктура MVC работает с jQuery, анализируя атрибуты модели, чтобы определить, какие проверки должны быть добавлены. В результате анализа вспомогательный метод HTML по имени `EditorFor()` сгенерирует следующую разметку для свойства `Make`:

```
<input name="Make" class="form-control text-box single-line input-
validation-error" id="Make" aria-invalid="true" aria-describedby="Make-
error" type="text" value="" data-val-length-max="50" data-val-length="The
field Make must be a string with a maximum length of 50." data-val="true">
```

Атрибуты данных также поддерживают специальные сообщения об ошибках. Откройте файл `Inventory.cs` в проекте `AutoLotDAL` и модифицируйте атрибут `StringLength` для свойства `Make`, включив в него присваивание `ErrorMessage`:

```
[StringLength(50,ErrorMessage="Please enter a value less than 50 characters long.")]
public string Make { get; set; }
```

Запустив приложение и повторив тест, вы получите указанное сообщение об ошибке (рис. 34.26).

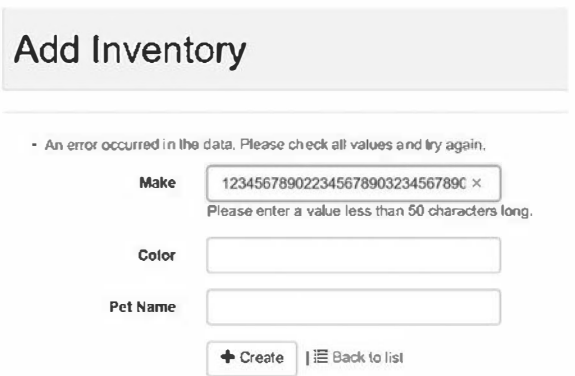


Рис. 34.26. Обновленное сообщение об ошибке, поступившее из аннотаций данных

Завершение пользовательского интерфейса

В завершение раздела главы, посвященного MVC, необходимо навести порядок в оставшихся элементах пользовательского интерфейса.

Обновление представления компоновки

Откройте файл _Layout.cshtml в папке Views/Shared. В верхнюю часть страницы добавьте блок кода Razor для объявления строковой переменной и присваивания ей значения "Car Lot MVC". Она предназначена для замены всех жестко закодированных вхождений имени приложения. Вот этот блок кода:

```
@{
  var appName = "Car Lot MVC";
}
```

Замените жестко закодированные строки "My ASP.NET Application" и "Application Name" конструкцией @appName. Вы найдете три места для внесения изменений: в HTML-дескрипторе <title>, в вызове вспомогательного метода ActionLink() панели навигации и в разделе <footer>. Ниже показан модифицированная разметка:

```
<!-- В разделе <head> -->
<title>@ViewBag.Title - @appName</title>
<head>
   <title>@ViewBag.Title - @appName</title>
   <!-- Для краткости остальной код не показан -->
</head>

<!-- В элементе <div> панели навигации -->
<div class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle" data-toggle="collapse"
              data-target=".navbar-collapse">
        <!-- Для краткости остальной код не показан -->
      </button>
      @Html.ActionLink(@appName, "Index", "Home", new { area = "" },
                       new { @class = "navbar-brand" })
    </div>
    <!-- Для краткости остальной код не показан -->
  </div>
</div>
<!-- В разделе <footer> -->
<footer>
  <p>&copy; @DateTime.Now.Year - @appName</p>
</footer>
```

Наконец, поместите в панель меню изображение автомобиля. На реальном веб-сайте вы вряд ли будете размещать рекламу в таком месте; однако настоящий пример демонстрирует возможность помещения в панель меню любых элементов, и когда размеры окна просмотра изменяются, быстрореагирующая природа Bootstrap обеспечит их сокрытие. Создайте папку по имени Images внутри папки Content в проекте CarLotMVC. Добавьте в папку файл изображения; его можно найти в подкаталоге Chapter_34 загружаемого кода примеров. Теперь добавьте изображение в панель меню прямо перед вызовом для загрузки частичного представления _LoginPartial:

```
<img src="~/Content/Images/CAR.gif" />
@Html.Partial("_LoginPartial")
```

После запуска приложения вы увидите в панели меню изображение автомобиля (рис. 34.27). Когда размеры окна просмотра уменьшаются, изображение исчезает вместе со ссылками в меню.

Рис. 34.27. Добавление изображения в панель меню

Обновление домашней страницы

Осталось изменить домашнюю страницу, которой, как вы знаете, является представление `Index` контроллера `Home`. Для начала модифицируйте блок `JumboTron`, как показано ниже:

```
<div class="jumbotron">
  <h1>Car Lot MVC</h1>
  <p class="lead">A site for viewing and updating vehicles in the dealership.</p>
</div>
```

Затем удалите все содержимое после блока `JumboTron` и добавьте следующую разметку:

```
<h2>View the Inventory</h2>
<p>
  Autolot has the car you are looking for!  Check out our expansive
inventory on the @Html.ActionLink("Inventory", "Index", "Inventory") page.
</p>
```

Окончательный вид домашней страницы приведен на рис. 34.28.

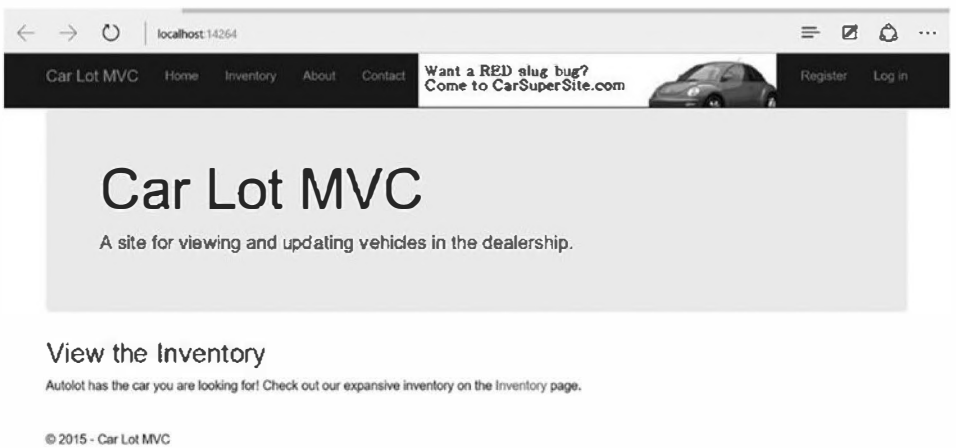


Рис. 34.28. Финальная версия домашней страницы

Заключительное слово по поводу ASP.NET MVC

Часто возникает вопрос: что выбрать — Web Forms или MVC? Ответить на него не так-то просто. Если разработчикам более комфортно иметь дело с операциями перетаскивания при создании пользовательского интерфейса или у них вызывает трудно-

сти лишенная состояния природа HTTP, то, скорее всего, наилучшим вариантом будет инфраструктура Web Forms. Если же разработчики предпочитают располагать полным контролем над пользовательским интерфейсом (что также означает меньший объем "магии", выполняемой автоматически) и строить приложения, лишенные состояния, которые задействуют методы HTTP (такие как `HttpGet` и `HttpPost`), то вероятно наилучшим вариантом окажется инфраструктура MVC. Конечно, при поиске решения придется учитывать намного больше факторов. Были упомянуты лишь несколько из них.

Хорошая новость в том, что вам не придется делать выбор между MVC и Web Forms. Как упоминалось в начале главы, обе инфраструктуры основаны на пространстве имен `System.Web` (включая MVC 5) и всегда могут использоваться вместе. После того, как в Visual Studio 2013 была введена концепция "One ASP.NET", смешивать две инфраструктуры в единственном проекте стало намного легче.

Следуем отметить, что в главе мы только слегка коснулись поверхности ASP.NET MVC. Тема MVC слишком обширна, чтобы ее можно было раскрыть в одной главе. Исчерпывающее описание всего того, что инфраструктура MVC способна предложить, можно найти в книге *ASP.NET MVC 5 с примерами на C# для профессионалов, 5-е изд.* (ИД "Вильямс", 2015 г.).

Исходный код. Решение `CarLotMVC` доступно в подкаталоге `Chapter_34`.

Введение в ASP.NET Web API

В главе 25 вы узнали, что Windows Communication Foundation (WCF) является полноценной инфраструктурой для создания основанных на .NET служб, которые могут взаимодействовать с широким диапазоном клиентов. Наряду с тем, что инфраструктура WCF отличается исключительной мощью, если требуются простые основанные на HTTP службы, то разработка службы с применением WCF может оказаться сложнее, чем необходимо или желательно. И здесь в игру вступает ASP.NET Web API — еще одна инфраструктура для построения API-интерфейсов веб в .NET, к которым возможен доступ из любого клиента, поддерживающего HTTP. С точки зрения разработки MVC инфраструктура Web API является логическим дополнением инструментального набора .NET. Она построена на основе MVC и задействует многие из тех же самых концепций, таких как модели, контроллеры и маршрутизация. Инфраструктура Web API была впервые выпущена в составе Visual Studio 2012 как часть MVC 4 и обновлена до версии 2.2 в Visual Studio 2013 Update 1.

Добавление проекта Web API

Начните с добавления проекта Web API к имеющемуся решению. Щелкните правой кнопкой мыши на имени решения, выберите в контекстном меню пункт Add⇨New Project (Добавить⇨Новый проект) и укажите вариант ASP.NET Web Application (Веб-приложение ASP.NET), как показано на рис. 34.29. Назначьте проекту имя `CarLotWebAPI`.

К этому времени вы уже должны быть знакомы с диалоговым окном, которое открывается следующим. Выберите шаблон Empty (Пустой) и в области Add folders and core references for (Добавить папки и основные ссылки для) отметьте флажок Web API (рис. 34.30). В случае выбора шаблона Web API к проекту добавляется большой объем стереотипного кода (включая контроллеры и представления MVC), но вас интересует только базовый связующий код Web API. В открывшемся далее диалоговом окне на выбор предлагаются те же самые варианты, что и при создании проекта приложения Web Forms или MVC; коллективно на них ссылаются как на концепцию "One ASP.NET".

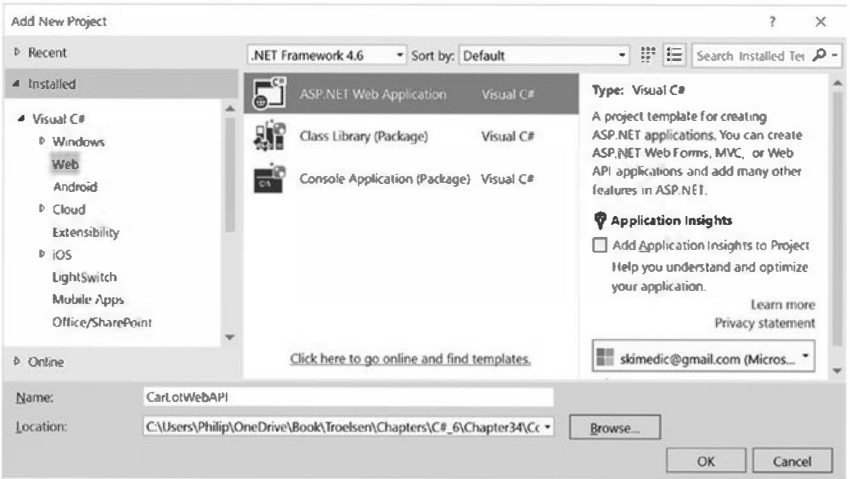


Рис. 34.29. Добавление нового проекта веб-приложения ASP.NET

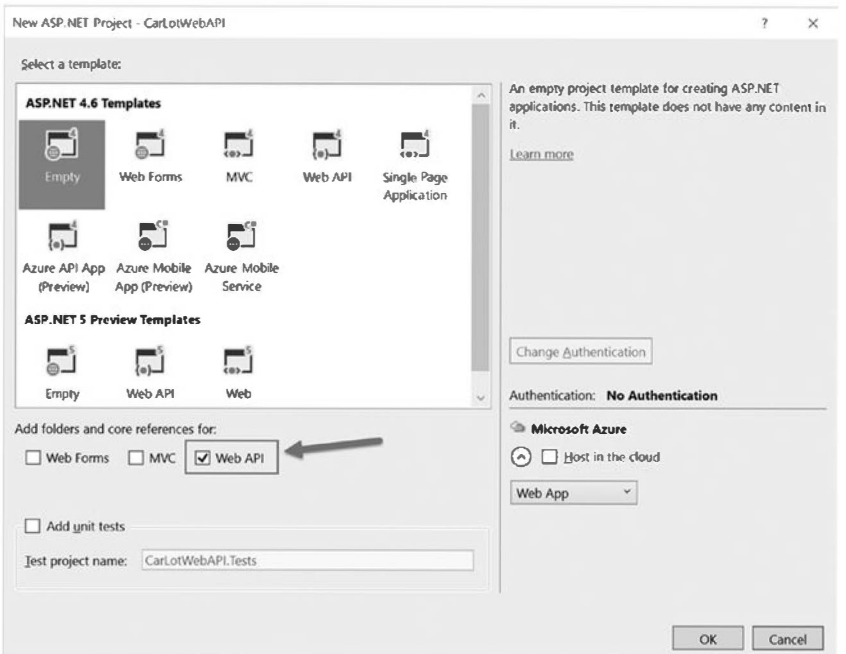


Рис. 34.30. Добавление шаблона проекта Empty с поддержкой Web API

Щелкните на кнопке OK для добавления проекта к решению. Как и в ситуации с проектами Web Forms и MVC, многие включенные пакеты NuGet устарели к моменту создания проекта. Щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer, выберите в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet) и в поле со списком Filter (Фильтр) выберите Upgrade Available (Доступно обновление). Обновите все пакеты, которые можно обновить. Верните в поле Filter выбор All (Все) и установите Entity Framework (как делали это ранее). Вам понадобится установить один дополнительный пакет: AutoMapper (он будет использоваться позже в главе). Чтобы найти этот пакет, введите в поле поиска строку `AutoMapper` (рис. 34.31).

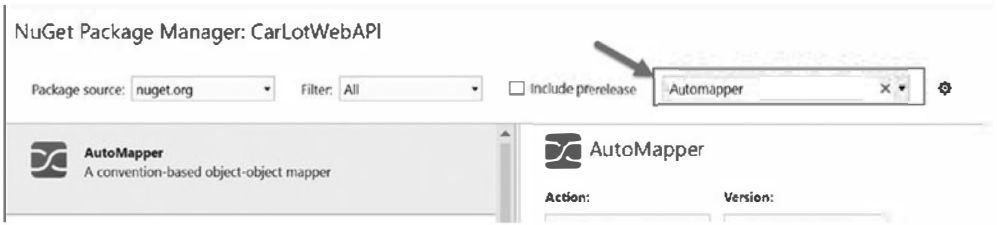


Рис. 34.31. Установка пакета AutoMapper с помощью NuGet

Наконец, добавьте ссылку на проект `AutoLotDAL` (щелкнув правой кнопкой мыши на узле References (Ссылки) в окне Solution Explorer для проекта `CarLotWebAPI` и выбрав `AutoLotDAL` в списке проектов и решений). Добавьте строку подключения в файл `Web.config` (она может отличаться в зависимости от пути установки):

```
<connectionStrings>
  <add name="AutoLotConnection" connectionString="data source=localhost\
SQLEXPRESS2014;initial catalog=AutoLot;integrated security=True;
MultipleActiveResultSets=True;App=EntityFramework"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

Исследование проекта Web API

Этот проект содержит намного меньше файлов, чем проект MVC, созданный в начале главы. Давайте посмотрим, какие файлы были созданы. Откройте файл `WebApiConfig.cs` из папки `App_Start`. Код (приведен ниже) должен быть вам знаком. В первой строке включается маршрутизация с помощью атрибутов (здесь не рассматривается). Во второй строке определяется стандартный маршрут со стандартными значениями. Стандартный маршрут кое в чем отличается о того, что вы видели в MVC. Первая крупная разница — отсутствие действия в маршруте. Причина в том, что (как вы узнаете позже в главе) маршрутизация за пределами контроллера основана на методе HTTP, применяемом в запросе. Наконец, подобно MVC параметр `id` устанавливается в `RouteParameter.Optional`.

```
public static void Register(HttpConfiguration config)
{
  // Конфигурация и службы Web API

  // Маршруты Web API
  config.MapHttpAttributeRoutes();

  config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
  );
}
```

Затем откройте файл `Global.asax.cs` (содержимое показано ниже). Это усеченная версия того, что вы видели в MVC. В файле присутствует единственная строка, которая добавляет конфигурацию для веб-маршрута (веб-маршрутов).

```
protected void Application_Start()
{
  GlobalConfiguration.Configure(WebApiConfig.Register);
}
```

Конфигурирование проекта

Поскольку проект `CarLotWebAPI` связан с построением автоматической службы (т.е. без пользовательского интерфейса), его необходимо настроить на запуск и ожидание обращения к одному из сконфигурированных маршрутов. Для этого откройте окно свойств проекта, щелкнув правой кнопкой мыши на имени проекта в окне Solution Explorer и выбрав в контекстном меню пункт Properties (Свойства). Щелкните на элементе Web в панели слева и выберите переключатель Don't open a page. Wait for a request from an external application (Не открывать страницу. Ожидать запроса из внешнего приложения). Кроме того, обратите внимание на поле Project URL (URL проекта), которым в данном примере является `http://localhost:46024/`. Окно свойств показано на рис. 34.32.

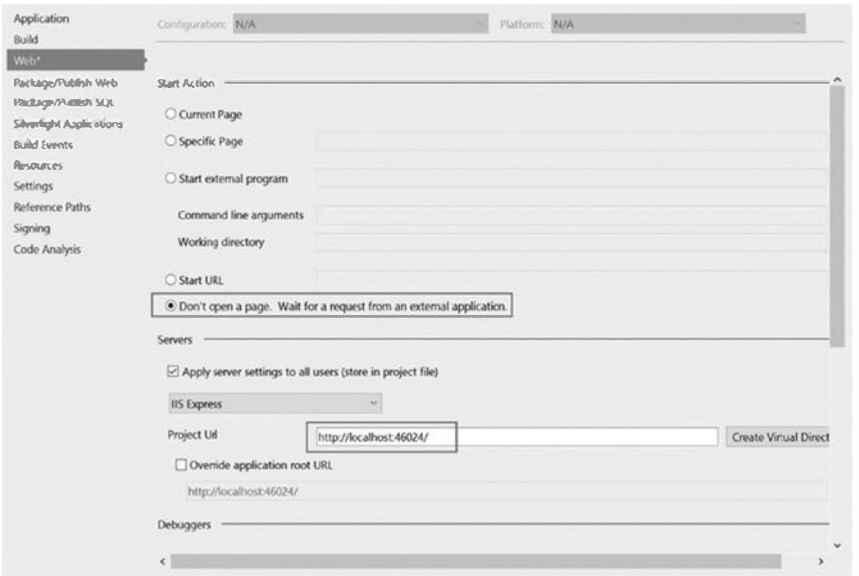


Рис. 34.32. Конфигурирование запускаемого проекта `CarLotWebAPI`

Установите `CarLotWebAPI` в качестве запускаемого проекта в решении, для чего щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer и выберите в контекстном меню пункт Set as StartUp Project (Установить как запускаемый проект).

Замечание относительно JSON

Формат JSON (JavaScript Object Notation — запись объектов JavaScript) является одним из способов транспортировки данных между службами. Это простое текстовое представление в виде пар "ключ-значение" объектов и классов. Например, взгляните на следующее представление JSON элемента `Inventory`:

```
{"CarId":1,"Make":"VW","Color":"Black","PetName":"Zippy",
 "Timestamp":"AAAAAAAAB9o=","Orders":[]}
```

Каждый объект JSON начинается и заканчивается фигурными скобками, а имена свойств и строковые значения помещены в двойные кавычки. Объекты JSON могут быть также вложенными. Если свойство `Make` было бы не строкой, а объектом (со свойствами `Builder` и `Year`), то представление JSON могло бы выглядеть так:

```
{"CarId":1,"Make":{"Builder":"VW","Year":2015},"Color":"Black",
 "PetName":"Zippy","Timestamp":"AAAAAAAAB9o=","Orders":[]}
```

Как видно по свойству `Orders`, списки обозначаются квадратными скобками (`[]`). Если служба отправляет список объектов `Inventory`, то представление JSON может напоминать следующее:

```
[{"CarId":1,"Make":"VW","Color":"Black","PetName":"Zippy",
 "Timestamp":"AAAAAAAAB9o=","Orders":[]},{"CarId":2,"Make":"Ford",
 "Color":"Rust","PetName":"Rusty","Timestamp":"AAAAAAAAB9s=","Orders":[]}]
```

На заметку! Шаблон проекта Web API включает бесплатную инфраструктуру с открытым кодом под названием JSON.NET. Это надежная инфраструктура для создания представлений JSON из объектов, а также для создания объектов из представлений JSON. Вы будете использовать JSON.NET позже в главе, а дополнительная информация (включая документацию и примеры) доступна по адресу `www.newtonsoft.com/json`.

Добавление контроллера

Как и MVC, код Web API вращается вокруг контроллеров и действий. Щелкните правой кнопкой мыши на папке `Controllers` и выберите в контекстном меню пункт Add⇨Controller (Добавить⇨Контроллер). В открывшемся диалоговом окне выберите вариант Web API 2 Controller with actions, using Entity Framework (Контроллер Web API 2 с действиями, использующими Entity Framework), как показано на рис. 34.33, и щелкните на кнопке Add (Добавить).

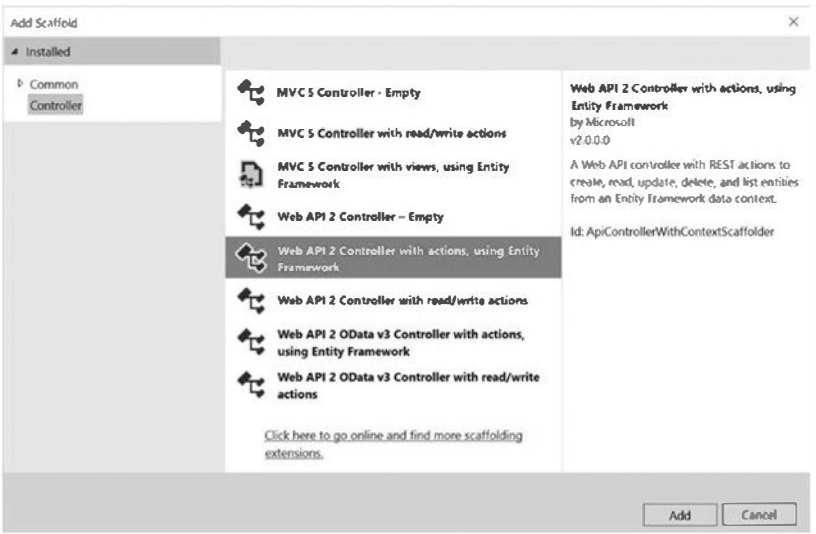


Рис. 34.33. Добавление нового контроллера Web API 2

В диалоговом окне Add Controller (Добавление контроллера) выберите класс `Inventory` для модели и класс `AutoLotEntities` для контекста данных. Отметьте флажок Use async controller actions (Использовать асинхронные действия контроллера), измените имя контроллера на `InventoryController` (рис. 34.34) и щелкните на кнопке Add.

Исследование методов контроллера

Откройте созданный файл `InventoryController.cs`, добавьте переменную-член типа `InventoryRepo` и создайте экземпляр класса `InventoryRepo`. Освободите его в методе `Dispose()` класса контроллера.

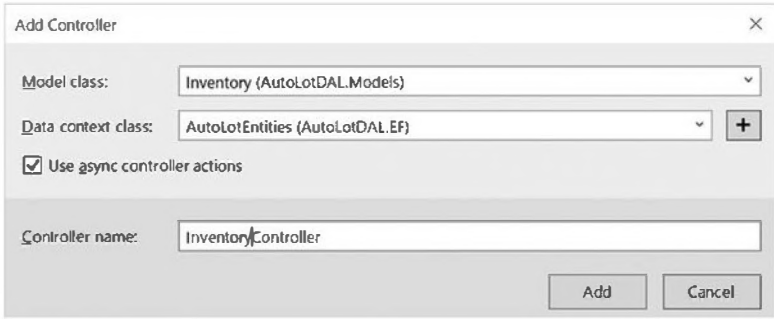


Рис. 34.34. Добавление классов модели и контекста данных для контроллера

Оба фрагмента кода показаны ниже:

```
private readonly InventoryRepo _repo = new InventoryRepo();
protected override void Dispose(bool disposing)
{
  if (disposing)
  {
    db.Dispose();
    _repo.Dispose();
  }
  base.Dispose(disposing);
}
```

Взгляните на сигнатуры методов действий. Хотя они напоминают действия из класса `InventoryController` в проекте MVC, есть несколько бросающихся в глаза отличий. Вместо маршрутизации запроса на основе содержимого URL многие действия принимают тот же самый URL! Вспомните, что в MVC для действий `Add`, `Update` и `Delete` предусмотрено по два метода, и инфраструктура решает, какой из них вызывать, основываясь на методе HTTP запроса (`HttpGet` или `Httpost`). Инфраструктура Web API следует такому же шаблону, но для проведения различий между вызовами применяет дополнительные методы HTTP. Помимо методов `HttpGet` и `HttpPost`, используемых в MVC, в Web API также задействованы методы `HttpPut` и `HttpDelete`. Еще одно заметное отличие связано с отсутствием атрибутов, отражающих методы HTTP, которыми в MVC декорированы методы действий. Это снова соглашение по конфигурации. Инфраструктура Web API ищет нечувствительное к регистру символов совпадение начала имени действия с методом HTTP. Например, действие по имени `DeleteInventory` будет обрабатывать запрос `HttpDelete`. Безусловно, можно вызывать любой желаемый метод, но такой метод должен быть декорирован корректным атрибутом. В последующих разделах вы ознакомитесь с действиями более детально по мере того, как будете модифицировать их для применения класса `InventoryRepo` из библиотеки `AutoLotDAL`.

Получение всех записей склада

Существуют два метода `HttpGet` — `GetInventory()` и `GetInventory(int id)`. Первый получает все записи склада (вспомните, что параметр `id` в маршруте необязателен). Это довольно стандартный метод, подобный тем, что вы видели в контроллере MVC, но только он возвращает не `ActionResult`, а просто данные. В действительности инфраструктура Web API помещает данные в оболочку `HttpResponseMessage` (с кодом HTTP 200), добавляя их как тело сообщения. Здесь понадобится вызвать метод `InventoryRepo.GetAll()` и изменить возвращаемый тип на `IEnumerable<Inventory>`:

```
// GET: api/Inventory
```

```
public IEnumerable<Inventory> GetInventory()
{
  return _repo.GetAll();
}
```

Пришло время протестировать приложение. После запуска решения все выглядит так, будто ничего не происходит за исключением того, что значок выполнения в Visual Studio заменяется значком паузы/останова. Причина в том, что вы сконфигурировали проект на автоматический запуск и ожидание внешнего обращения. Когда что-то загрузилось (и значки отладки изменились), откройте новое окно браузера. Введите URL для службы (отображаемый в окне свойств проекта) и добавьте маршрут `api/Inventory`. Вот как выглядит URL (номер порта может отличаться):

```
http://localhost:46024/api/Inventory
```

Введя этот URL в поле адреса в браузере, вы получите показанное ниже сообщение об ошибке в виде простого текста (здесь приведена только основная его часть; действительное сообщение намного длиннее). В инфраструктуре Web API не предусмотрен "желтый экран смерти", который вы привыкли видеть в Web Forms и MVC.

```
"Message":"An error has occurred.","ExceptionMessage":"Self referencing
loop detected for property 'Car' with type 'System.Data.Entity.
DynamicProxies.Inventory_4F2216023579E149E169D586253289F35987B42694292AD3
BF08836508A419F5'. Path '[0].Orders[0]'"
```

"Message":"Произошла ошибка.","ExceptionMessage":"Обнаружен цикл с рекурсивной ссылкой для свойства Car с типом System.Data.Entity. DynamicProxies.Inventory_4F2216023579E149E169D586253289F35987B42694292AD3 BF08836508A419F5'. Path '[0].Orders[0]"

Отсутствие традиционной страницы ошибки, отображаемой в MVC (или Web Forms), объясняется тем, что в инфраструктуре Web API все возвращается в формате JSON, если только явно не указано иное. Следовательно, вызывающему приложению (браузеру в этом случае) ничего не известно о том, что возникла ошибка; оно просто отображает возвращаемый текст. Обратите внимание, что Web API может также возвращать стандартные коды ошибок HTTP (как будет продемонстрировано позже). Когда Web API возвращает ошибку, вызывающее приложение отвечает за ее интерпретацию и соответствующую обработку. Чтобы увидеть это, откройте инструменты разработчика в браузере (в большинстве браузеров нужно нажать <F12>), перейдите на вкладку Network (Сеть) и обновите страницу. Инструменты разработчика отобразят подробные сведения о вызове, возвращаемое сообщение HTTP и любые возникшие ошибки (рис. 34.35).

Ошибка произошла из-за того, что инфраструктура EF по умолчанию загружает сущности ленивым образом. Как вы наверняка помните из главы 23, ленивая загрузка означает, что EF будет обращаться к информации в базе данных, когда запрашиваются свойства. Процесс сериализации объектов .NET обходит каждое свойство, так что в этом случае он достигает свойства `Orders` со списком объектов `Order`, а класс `Order` имеет ссылку на класс `Inventory`. Такая циклическая ссылка приводит к отказу в работе процесса сериализации. Чтобы решить проблему, понадобится либо отключить ленивую загрузку, либо скопировать все относящиеся к делу свойства в новый класс, игнорируя те из них, которые вызывают проблемы с сериализацией. Для этого будет использоваться пакет AutoMapper.

Создание моделей представлений с помощью AutoMapper

Пакет AutoMapper (установленный ранее в главе) является бесплатным инструментом с открытым кодом для создания нового экземпляра типа из экземпляра другого типа. Он также может применяться для создания нового экземпляра того же самого типа, что и будет делаться здесь.

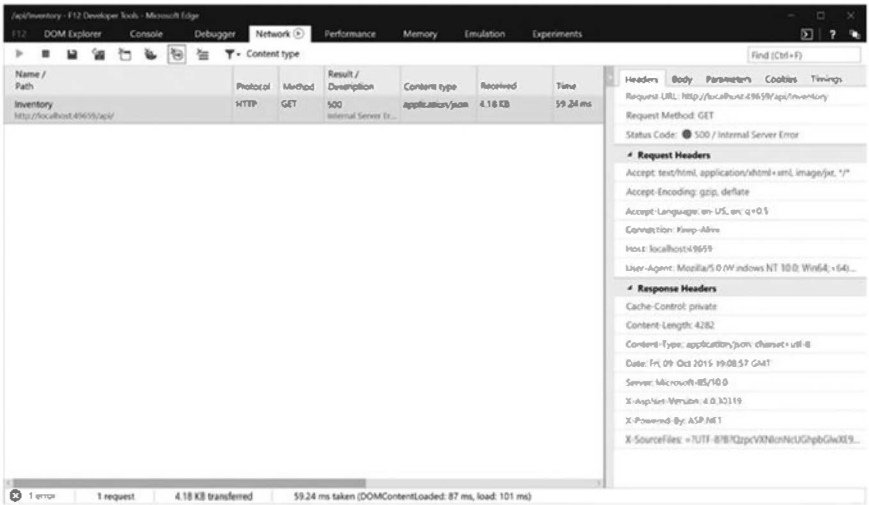


Рис. 34.35. Отображение ошибок HTTP в браузере Microsoft Edge

Добавьте в класс `InventoryController` **новый конструктор со следующим кодом:**

```
public InventoryController()
{
  Mapper.Initialize(
    cfg =>
    {
      cfg.CreateMap<Inventory, Inventory>()
          .ForMember(x => x.Orders, opt => opt.Ignore());
    });
}
```

В коде создается отображение между типом `Inventory` **и им же самим с игнорированием свойства** `Orders`. **Инструмент AutoMapper использует рефлексию для выяснения, какие свойства сопоставляются между двумя типами, и копирует все значения из исходного экземпляра в новый экземпляр целевого типа за исключением значений игнорируемых свойств (навигационного свойства** `Orders` **в этом примере). Как будет показано далее, он также работает с коллекциями.**

Модифицируйте метод `Inventory()`, **чтобы преобразовать список записей** `Inventory` **в новый список записей** `Inventory`, **не содержащий свойство** `Orders`:

```
// GET: api/Inventory
public IEnumerable<Inventory> GetInventory()
{
  var inventories = _repo.GetAll();
  return Mapper.Map<List<Inventory>, List<Inventory>>(inventories);
}
```

На заметку! Хотя в главе недостаточно места для глубоких исследований AutoMapper, следует отметить, что он является инструментом, широко применяемым разработчиками приложений .NET. Вы должны добавить его в свой арсенал средств разработки. Дополнительные сведения, включая документацию и примеры, можно найти на домашней странице проекта по адресу `http://automapper.org`.

Снова запустите приложение и в браузере Internet Explorer или Microsoft Edge (браузеры Chrome и Firefox по умолчанию возвращают разметку XML) введите URI для

Inventory (http://localhost:46024/api/Inventory). Вы увидите такой вывод JSON (действительные данные могут варьироваться):

```
[{"CarId":1,"Make":"VW","Color":"Black","PetName":"Zippy","Timestamp":"AAAAA
AAB9o=","Orders":[]},{"CarId":2,"Make":"Ford","Color":"Rust","PetName":
"Rusty","Timestamp":"AAAAAAAAB9s=","Orders":[]},{"CarId":3,"Make":"Saab",
"Color":"Black","PetName":"Mel","Timestamp":"AAAAAAAAB9w=","Orders":[]},
{"CarId":4,"Make":"Yugo","Color":"Yellow","PetName":"Clunker","Timestamp":
"AAAAAAAAB90=","Orders":[]},{"CarId":5,"Make":"BMW","Color":"Black",
"PetName":"Bimmer","Timestamp":"AAAAAAAAB94=","Orders":[]},{"CarId":6,
"Make":"BMW","Color":"Green","PetName":"Hank","Timestamp":"AAAAAAAAB98=",
"Orders":[]},{"CarId":7,"Make":"BMW","Color":"Pink","PetName":"Pinky",
"Timestamp":"AAAAAAAAB+A=","Orders":[]},{"CarId":13,"Make":"Pinto",
"Color":"Black","PetName":"Pete","Timestamp":"AAAAAAAAB+E=","Orders":[]},
{"CarId":54,"Make":"Yugo","Color":"Brown","PetName":"Brownie","Timestamp":
"AAAAAAABX5E=","Orders":[]},{"CarId":55,"Make":"Yugo","Color":"Brown",
"PetName":"Brownie","Timestamp":"AAAAAAABbzE=","Orders":[]},{"CarId":56,
"Make":"Yugo","Color":"Brown","PetName":"Brownie","Timestamp":"AAAAAAABftE=",
"Orders":[]},{"CarId":57,"Make":"Yugo","Color":"Brown","PetName":"Brownie",
"Timestamp":"AAAAAAABjnE=","Orders":[]},{"CarId":58,"Make":"Yugo","Color":
"Brown","PetName":"Brownie","Timestamp":"AAAAAAABnhE=","Orders":[]},{"CarId":
59,"Make":"Yugo","Color":"Brown","PetName":"Brownie","Timestamp":"AAAAAAABrbE=",
"Orders":[]},{"CarId":60,"Make":"Yugo","Color":"Brown","PetName":"Brownie",
"Timestamp":"AAAAAAABvVE=","Orders":[]},{"CarId":61,"Make":"Yugo","Color":
"Brown","PetName":"Brownie","Timestamp":"AAAAAAABzPE=","Orders":[]},{"CarId":62,
"Make":"Yugo","Color":"Brown","PetName":"Brownie","Timestamp":"AAAAAAAB3JE=",
"Orders":[]},{"CarId":63,"Make":"Yugo","Color":"Brown","PetName":"Brownie",
"Timestamp":"AAAAAAAB7DE=","Orders":[]},{"CarId":64,"Make":"Yugo","Color":
"Brown","PetName":"Brownie","Timestamp":"AAAAAAAB+9E=","Orders":[]}]
```

Легко заметить, что свойство Orders по-прежнему присутствует, но все записи пусты во избежание ошибки циклической ссылки. Проблема решена!

Получение одной записи склада

Второй метод HttpGet, GetInventory(int id), возвращает одиночную запись склада, соответствующую переданному идентификатору. Модифицируйте его для вызова метода GetOneAsync() хранилища и посредством AutoMapper создайте новый экземпляр класса Inventory:

```
// GET: api/Inventory/5
[ResponseType(typeof(Inventory))]
public async Task<IHttpActionResult> GetInventory(int id)
{
  Inventory inventory = await _repo.GetOneAsync(id);
  if (inventory == null)
  {
    return NotFound();
  }
  return Ok(Mapper.Map<Inventory,Inventory>(inventory));
 }
```

В этом методе действия появились четыре новых элемента/метода: атрибут ResponseType, интерфейс IHttpActionResult, а также методы NotFound() и Ok(). Атрибут ResponseType используется для указания типа сущности, возвращаемой в теле HttpResponseMessage. Это нужно, чтобы сериализировать запись Inventory для HttpActionResult (версия Web API класса ActionResult из MVC). Метод NotFound() возвращает объект NotFoundResult, который транслируется в сообщение об ошибке

404. Метод `Ok()` возвращает код состояния 200 и добавляет объект или объекты, переданные методу (в формате JSON или XML, что зависит от браузера), в тело сообщения.

Чтобы протестировать код, введите в поле адреса Internet Explorer/Edge следующий URI (номер порта и значение `CarId` могут отличаться):

```
http://localhost:46024/api/Inventory/5
```

Обновление записи склада

Обновление записи в HTTP достигается вызовом метода `HttpPut` с передачей идентификатора записи, подлежащей обновлению, и самого обновляемого объекта. Этот метод применяет привязку модели (подобно Web Forms и MVC) для создания экземпляра класса `Inventory` со значениями, отправленными из клиента в теле сообщения. Модифицируйте метод `PutInventory()` для использования `InventoryRepo`. Ниже показан код:

```
// PUT: api/Inventory/5
[ResponseType(typeof(void))]
public async Task<IHttpActionResult> PutInventory(int id, Inventory
inventory)
{
  if (!ModelState.IsValid)
  {
    return BadRequest(ModelState);
  }
  if (id != inventory.CarId)
  {
    return BadRequest();
  }
  try
  {
    await _repo.SaveAsync(inventory);
  }
  catch (Exception ex)
  {
    // В производственном приложении здесь должны быть дополнительные действия
    throw;
  }
  return StatusCode(HttpStatusCode.NoContent);
 }
```

Внутри метода действия (как с привязкой модели в MVC и Web Forms) сначала проверяется допустимость состояния модели (`ModelState`). Если состояние модели не является допустимым, то возвращается `HttpBadRequest` (400). Если же оно допустимо, то затем проверяется соответствие значения идентификатора, переданного в URL, значению `CarId` записи `Inventory` (из тела сообщения). Это помогает сократить (но не устранить) возможность взлома URL злоумышленниками. Далее предпринимается попытка сохранить запись, и если она завершается успешно, то возвращается код состояния HTTP 204 (содержимое отсутствует). В случае если возникло исключение, оно просто повторно генерируется для клиента. В приложении производственного уровня понадобится предусмотреть более развитую обработку исключений.

Добавление записей склада

Добавление записи в HTTP производится посредством вызова метода `HttpPost` с передачей обновляемого объекта в теле сообщения. В MVC метод `Httpost` применяется для всего того, что не является запросами `HttpGet`, но инфраструктура Web API более

совершенна и использует методы HTTP корректно. Метод `PostInventory()` также применяет привязку модели для создания экземпляра класса `Inventory` со значениями, отправленными из клиента в теле сообщения. Модифицируйте метод для использования `InventoryRepo`. Код выглядит следующим образом:

```
// POST: api/Inventory
[ResponseType(typeof(Inventory))]
public async Task<IHttpActionResult> PostInventory(Inventory inventory)
{
  if (!ModelState.IsValid)
  {
    return BadRequest(ModelState);
  }
  try
  {
    await _repo.AddAsync(inventory);
  }
  catch (Exception ex)
  {
    // В производственном приложении здесь должны быть дополнительные действия
    throw;
  }
  return CreatedAtRoute("DefaultApi", new { id = inventory.CarId }, inventory);
}
```

Метод действия `PostInventory()` возвращает добавленную запись `Inventory`, дополненную сгенерированными сервером значениями, поэтому `ResponseType` имеет тип `Inventory`. Кроме того, данный метод действия применяет привязку модели для получения значений из тела сообщения, проверяет `ModelState` и возвращает `HttpBadRequest` (400), если во время привязки модели возникли проблемы. В случае успешной привязки модели метод попытается добавить новую запись. Если добавление прошло успешно, возвращается `HttpCreated` (201) с новой записью `Inventory` в теле сообщения.

Удаление записей склада

Последним подлежащим обновлению методом действия является `DeleteInventory()`. Шаблон контроллера Web API создал метод, который принимает `id`, извлекает запись, если она существует, и удаляет ее. Проблема здесь (как объяснялось при реализации действия `Delete` в MVC) в том, что библиотека `AutoLotDAL` использует проверку параллелизма, которая гарантирует, что никто другой не изменил запись до того, как текущий пользователь отправил запрос на ее удаление. Таким образом, вы должны модифицировать сигнатуру метода, для приема значения `id` и объекта `Inventory`, заполненного из тела сообщения. Измените метод следующим образом:

```
// DELETE: api/Inventory/5
[ResponseType(typeof(void))]
public async Task<IHttpActionResult> DeleteInventory(int id,
                                                     Inventory inventory)
{
  if (id != inventory.CarId)
  {
    return BadRequest();
  }
  try
  {
    await _repo.DeleteAsync(inventory);
  }
```

```
    catch (Exception ex)
    {
      // В производственном приложении здесь должны быть дополнительные действия
      throw;
    }
    return Ok();
  }
```

В атрибуте ResponeType теперь указан тип void. В дополнение к параметру id метод принимает тип Inventory. Внутри метода производится проверка соответствия параметра id значению CarId записи Inventory из тела сообщения. Если они совпадают, то предпринимается попытка удалить запись Inventory. В случае успешного удаления возвращается код состояния 200.

Удаление переменной AutoLotEntities

Финальная очистка класса InventoryController предусматривает удаление переменной AutoLotEntities, находящейся в начале определения, и освобождение хранилища в методе Dispose() класса контроллера. Ниже показан модифицированный код метода Dispose():

```
protected override void Dispose(bool disposing)
{
  if (disposing)
  {
    _repo.Dispose();
  }
  base.Dispose(disposing);
}
```

Обновление проекта CarLotMVC для использования CarLotWebAPI

В настоящее время в проекте CarLotMVC для выполнения всех операций CRUD применяется библиотека AutoLotDAL. В этом разделе для всех операций доступа к данным будет использоваться CarLotWebAPI.

Обновление действия Index

Откройте файл InventoryController.cs из проекта CarLotMVC и перейдите к методу действия Index(). Вместо применения класса InventoryRepo из библиотеки AutoLotDAL будут использоваться службы в CarLotWebAPI. Создайте новый экземпляр HttpClient и вызовите метод GetAsync(), передав ему URL метода действия GetInventory(). Этот метод Web API возвращает тип IHTTPActionResult. Результат имеет два свойства, в которых вы заинтересованы: IsSuccessStatusCode и Content. Свойство IsSuccessStatusCode равно true, если вызов сработал. Данное свойство устраняет необходимость в проверке всех возможных возвращаемых кодов, т.к. многие из них считаются успешным завершением. Свойство Content предоставляет доступ к телу сообщения. В случае метода действия Index(), если записи Inventory были возвращены, то Web API возвратит код состояния 200.

Если работа выполнена успешно, то все записи Inventory содержатся в свойстве Content в формате JSON. Именно здесь в игру вступает инфраструктура JSON.NET (упомянутая ранее в главе). Метод JsonConvert.DeserializeObject<T>() вызывается с указанием в качестве обобщенного параметра либо одиночного типа (Inventory), либо реализации IEnumerable какого-то типа (List<Inventory>). Передайте методу строку JSON, после чего он попытается преобразовать ее в указанный тип. Если методу удалось успешно преобразовать текст в объекты, то он возвращает представление Index.

Номер порта для обращения к Web API может отличаться.

```
// GET: Inventory
public async Task<ActionResult> Index()
{
  var client = new HttpClient();
  var response = await client.GetAsync("http://localhost:46024/api/Inventory");
  if (response.IsSuccessStatusCode)
  {
    var items = JsonConvert.DeserializeObject<List<Inventory>>(
       await response.Content.ReadAsStringAsync());
    return View(items);
  }
  return HttpNotFound();
}
```

На заметку! Ни в одном из приведенных примеров не обеспечивается обработка ошибок, необходимая для приложения производственного уровня. Это сделано намеренно, чтобы примеры были ясными и краткими. В главе 7 объяснялось, как элегантно обрабатывать любые исключения.

Удостоверьтесь в том, что приложение `CarLotMVC` установлено в качестве стартового проекта, щелкнув правой кнопкой мыши на имени проекта `CarLotMVC` в окне Solution Explorer и выбрав в контекстном меню пункт Set as StartUp Project (Установить как запускаемый проект). Запустите приложение и щелкните на ссылке Inventory в меню; вы увидите ту же самую страницу, которая отображалась при работе метода `Index()` с библиотекой `AutoLotDAL`. Все очень просто.

На заметку! Вас может интересовать, нужна ли в дальнейшем ссылка на `AutoLotDAL` в `CarLotMVC`. Да, она необходима, во всяком случае, при такой структуре решения. Библиотека `AutoLotDAL` содержит определения моделей, а в `CarLotMVC` требуется доступ к классам моделей. Распространенный подход (для простоты в главе он не рассматривается) предусматривает помещение определений моделей в отдельную сборку и ссылку на нее из любого проекта, который нуждается в определениях типов для моделей.

Обновление действия `Details`

Теперь следует обновить метод действия `Details()`. Модифицируйте его код, как показано ниже (указав действительный номер порта):

```
// GET: Inventory/Details/5
public async Task<ActionResult> Details(int? id)
{
  if (id == null)
  {
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
  }
  var client = new HttpClient();
  var response =
await client.GetAsync($"http://localhost:46024/api/Inventory/{id.Value}");
  if (response.IsSuccessStatusCode)
  {
    var inventory = JsonConvert.DeserializeObject<Inventory>(
       await response.Content.ReadAsStringAsync());
    return View(inventory);
  }
  return HttpNotFound();
}
```

Главным здесь (подобно действию Index) является замена обращения к CarLotMVC для получения записи применением нового объекта HttpClient. В коде проверяется успешность получения ответа, и если это так, то с помощью JSON.NET содержимое сообщения десериализируется в объект Inventory. Затем возвращается представление.

Обновление действия Add

Существуют два метода действий Create(), но версия HttpGet в обновлении не нуждается, поскольку она загружает представление безо всякого взаимодействия с базой данных. Версия HttpPost требует модификации. К счастью, класс HttpClient позаботится о выполнении основной работы (почти как метод действия Index()). Тем не менее, прежде чем обновлять метод, понадобится добавить оператор using для пространства имен System.Net.Http и ссылку на сборку System.Net.Http.Formatting. Класс System.Net.Http.Formatting имеет расширяющие методы, которые будут использоваться в проекте повсеместно, в том числе метод PostAsJsonAsync(), задействованный в следующем коде. Вот полный код метода действия:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult>
   Create([Bind(Include = "Make,Color,PetName"] Inventory inventory)
{
  if (!ModelState.IsValid)
  {
    ModelState.AddModelError(string.Empty,
     "An error occurred in the data.  Please check all values and try again.");
    return View(inventory);
  }
  try
  {
    var client = new HttpClient();
    var response = await client.PostAsJsonAsync(
                  "http://localhost:46024/api/Inventory", inventory);
    if (response.IsSuccessStatusCode)
    {
      return RedirectToAction("Index");
    }
  }
  catch (Exception ex)
  {
    ModelState.AddModelError(string.Empty,
         $"Unable to create record: {ex.Message}");
  }
  return View(inventory);
}
```

Главное изменение связано с тем, что после создания экземпляра HttpClient производится вызов расширяющего метода PostAsJsonAsync(). Он принимает два параметра: URI службы (например, http:/localhost/46024/api/Inventory) и данные, подлежащие отправке (inventory). Метод PostAsJsonAsync() позаботится о создании строки в формате JSON, вызове метода HttpPost и вставке ваших данных в тело сообщения. Если запрос прошел успешно, то свойство IsSuccessStatusCode будет установлено в true.

Обновление действия Edit

Оба метода действия Edit() нуждаются в обновлении. Версия HttpGet должна обращаться к CarLotWebAPI для получения отображаемой записи; изменения будут теми же, которые вносились в метод действия Details(), чтобы получать данные от веб-службы.

```
// GET: Inventory/Edit/5
public async Task<ActionResult> Edit(int? id)
{
  if (id == null)
  {
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
  }
  var client = new HttpClient();
  var response =
await client.GetAsync($"http://localhost:46024/api/Inventory/{id.Value}");
  if (response.IsSuccessStatusCode)
  {
    var inventory = JsonConvert.DeserializeObject<Inventory>(
      await response.Content.ReadAsStringAsync());
    return View(inventory);
  }
  return new HttpNotFoundResult();
}
```

В версии HttpPost применяется похожий расширяющий метод как метод действия Add(). Расширяющий метод PutAsJsonAsync() создает сообщение HttpPut по указанному URL и добавляет объекты в тело сообщения в формате JSON. Ниже показан модифицированный код:

```
// POST: Inventory/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit(
 [Bind(Include = "CarId,Make,Color,PetName,Timestamp")] Inventory inventory)
{
  if (!ModelState.IsValid) { return View(inventory); }
  var client = new HttpClient();
  var response = await client.PutAsJsonAsync(
    $"http://localhost:46024/api/Inventory/{inventory.CarId}", inventory);
  if (response.IsSuccessStatusCode)
  {
    return RedirectToAction("Index");
  }
  return View(inventory);
}
```

На заметку! Вас может интересовать, почему метод действия в контроллере MVC помечался с помощью атрибута HttpPost, тогда как обращение к веб-службе представляет собой метод HttpPut. Важный момент заключается в том, что метод HTTP, используемый для вызова действий MVC, не обязан совпадать с методом HTTP, который применяется для обращения к методам действий Web API. Они являются разрозненными операциями.

Обновление действия Delete

Существуют два метода действий Delete(), и подобно версии HttpGet метода Edit() единственное изменение касается обращения к веб-службе для получения данных. Изменения вносятся тем же способом, как это делалось для методов действий Delete() и Details().

```
// GET: Inventory/Delete/5
public async Task<ActionResult> Delete(int? id)
{
  if (id == null)
  {
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
  }
  var client = new HttpClient();
  var response =
await client.GetAsync($"http://localhost:46024/api/Inventory/{id.Value}");
  if (response.IsSuccessStatusCode)
  {
    var inventory = JsonConvert.DeserializeObject<Inventory>(
        await response.Content.ReadAsStringAsync());
    return View(inventory);
  }
  return new HttpNotFoundResult();
}
```

Версия HttpGet метода действия Delete() требует дополнительной работы. Как и можно было предположить, в действительности есть расширяющий метод DeleteAsync() класса HttpClient, но он не принимает какого-либо параметра для содержимого из тела сообщения. Использование этого метода приведет к отказу запроса на удаление, потому что должно быть передано значение отметки времени для проверки параллелизма. Взамен потребуется вручную создать экземпляр HttpRequestMessage. Конструктор класса HttpRequestMessage принимает экземпляр HttpMethod в первом параметре и URL во втором. Создайте новый экземпляр, передав в качестве параметров метод HttpDelete и URL метода действия Delete():

```
HttpRequestMessage request = new HttpRequestMessage(
    HttpMethod.Delete,
    $"http://localhost:46024/api/Inventory/{inventory.CarId}");
```

Далее посредством JSON.NET сериализируйте объект inventory и добавьте результат к содержимому. Вызов сериализации очень прост:

```
JsonConvert.SerializeObject(inventory)
```

Во время присваивания содержимого объекта HttpRequestMessage вы должны установить кодировку и тип, которым является application.json:

```
Content = new StringContent(JsonConvert.SerializeObject(inventory),
Encoding.UTF8, "application/json")
```

Собрав все вместе, вы получите следующий код:

```
HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Delete,
  $"http://localhost:46024/api/Inventory/{inventory.CarId}")
  {
    Content =
      new StringContent(JsonConvert.SerializeObject(inventory),
      Encoding.UTF8, "application/json")
  };
```

Наконец, отправьте сообщение, вызвав метод `SendAsync()` экземпляра `HttpClient`; это приведет к отправке только что созданного запроса. Ниже показан полный код метода `Delete()`:

```
// POST: Inventory/Delete/5
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Delete([Bind(Include = "CarId,Timestamp")]
   Inventory inventory)
{
  try
  {
    var client = new HttpClient();

    HttpRequestMessage request = new HttpRequestMessage(
      HttpMethod.Delete,
      $"http://localhost:46024/api/Inventory/{inventory.CarId}")
      {
        Content =
          new StringContent(JsonConvert.SerializeObject(inventory),
                            Encoding.UTF8, "application/json")
      };
sponse = await client.SendAsync(request);
    return RedirectToAction("Index");
  }
  catch (DbUpdateConcurrencyException)
  {
    ModelState.AddModelError(string.Empty,
              "Unable to delete record. Another user updated the record.");
  }
  catch (Exception ex)
  {
    ModelState.AddModelError(string.Empty,
                             $"Unable to create record: {ex.Message}");
  }
  return View(inventory);
}
```

Тестирование приложения

Чтобы запустить приложение, необходимо сконфигурировать `CarLotMVC` и `CarLotWebAPI` на запуск, когда начинается отладка. Для этого щелкните правой кнопкой мыши на имени решения в окне Solution Explorer и выберите в контекстном меню пункт Set StartUp Projects (Установить запускаемые проекты). В открывшемся диалоговом окне выберите переключатель Multiple startup projects (Несколько запускаемых проектов) и в раскрывающихся списках в колонке Action (Действие) для проектов `CarLotMVC` и `CarLotWebAPI` выберите вариант Start (Запустить), как показано на рис. 34.36.

Если теперь вы нажмете <F5> для запуска решения, то проект `CarLotMVC` загрузится в браузер, отобразив домашнюю страницу, а проект `CarLotWebAPI` будет выполняться как автоматическое приложение, ожидая внешнего вызова. Щелкните на ссылке Inventory в меню и затем пощелкайте на страницах. С точки зрения пользователя приложение работает тем же самым образом.

Исходный код. Решение `CarLotMVC_WebAPI` доступно в подкаталоге `Chapter_34`.

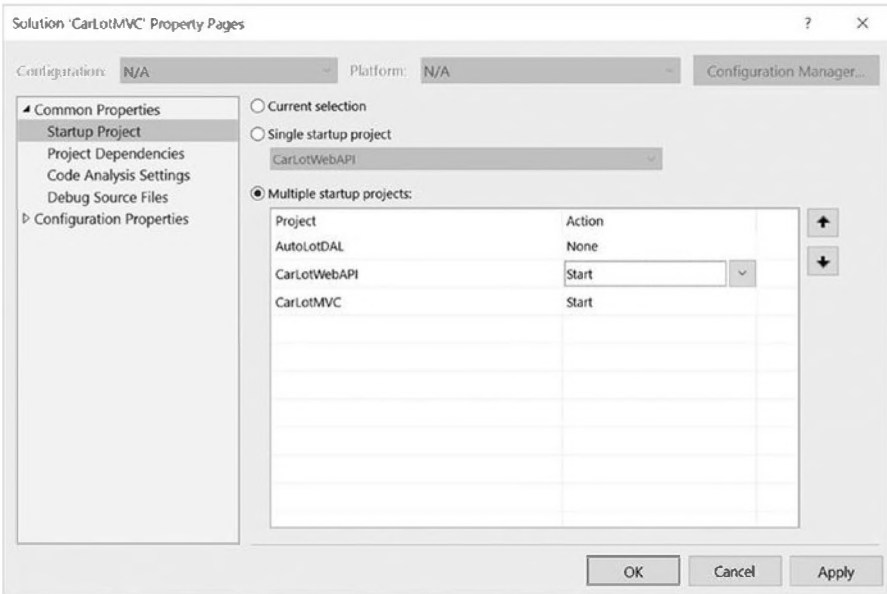


Рис. 34.36. Установка нескольких запускаемых проектов

Резюме

В этой главе были исследованы многие аспекты ASP.NET MVC и Web API. Для начала вы ознакомились с шаблоном Model-View-Controller и построили свой первый сайт MVC. Вы узнали о соглашении по конфигурации для инфраструктуры MVC, обо всех шаблонных файлах, создаваемых как часть нового шаблона проекта, а также о папках и их предназначении. Были рассмотрены все классы из папки `App_Start` и показано, каким образом они помогают в создании приложений MVC. Вы освоили объединение в пакеты и минификацию и научились при необходимости отключать данные функции.

Затем объяснялась маршрутизация и способы направления запросов контроллерам и действиям. Вы создали новые маршруты для страниц `About` и `Contact` и научились перенаправлять пользователей на другие ресурсы внутри сайта с применением маршрутизации вместо жестко закодированных URL.

Вы создали контроллер для страниц `Inventory` и узнали, каким образом формирование шаблонов, встроенное в Visual Studio, создает базовые методы действий и представления. Вы ознакомилось с запросами `HttpGet` и `HttpPost`, а также их взаимодействием с механизмом маршрутизации для обеспечения еще большей степени контроля над тем, какой метод действия будет вызван. Затем вы модифицировали методы действий для использования библиотеки `AutoLotDAL`, а также обновили сигнатуры и код для удовлетворения бизнес-требованиям.

Вы узнали о механизме представлений Razor и увидели синтаксис, вспомогательные методы, функции и делегаты Razor. Также были приведены дополнительные сведения о строго типизированных представлениях, частичных представлениях и компоновках. Вы научились отправлять данные представлению с применением объектов `ViewBag`, `ViewData` и `TempData`.

После этого вы изменили каждое шаблонное представление, обновили действия `InventoryController`, добавили проверку достоверности и посредством Bootstrap улучшили пользовательский интерфейс.

Далее вы узнали об инфраструктуре Web API, создали новый проект Web API и ознакомились с назначением и содержимым шаблонных файлов и папок. Затем вы использовали формирование шаблонов Visual Studio для добавления контроллера и его методов действий. Вы узнали о дополнительных методах HTTP и о том, как они применяются в маршрутизации Web API. Вы обновили каждый метод действия для использования библиотеки `AutoLotDAL` и инфраструктуры Entity Framework и с помощью AutoMapper устранили проблему циклической ссылки, возникающую из-за ленивой загрузки EF и сериализации.

После модификации всех действий `InventoryController` в `CarLotWebAPI` вы обновили проект `CarLotMVC` с целью обращения к URL службы `CarLotWebAPI`, применяя инфраструктуру JSON.NET для сериализации и десериализации записей. Вы также узнали, каким образом делать вызовы с использованием дополнительных методов HTTP, применяемых Web API.

Предметный указатель

К

П

Р

С

Т

У

Ф

Х

Ц

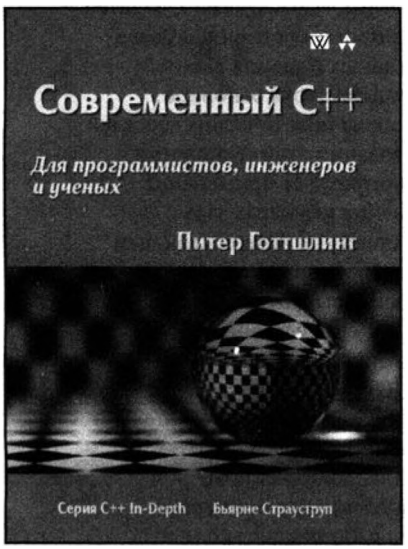
Э

Я

СОВРЕМЕННЫЙ C++

Для программистов, инженеров и ученых

Питер Готтшлинг



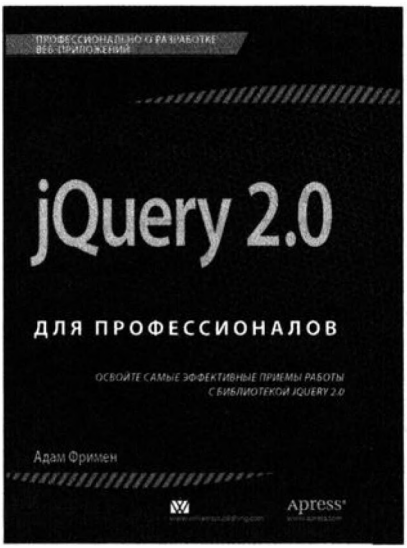
www.williamspublishing.com

Перед вами книга для тех, кто нуждается в быстром освоении передовых возможностей C++. В ней описаны мощные возможности стандарта C++14, наиболее полезные для научных и инженерных приложений. Книга не предполагает у читателя наличия опыта программирования на C++ или иных языках программирования. Читатели узнают, как воспользоваться преимуществами мощных библиотек, доступных для программистов C++: стандартной библиотеки шаблонов (STL) и научных библиотек для решения задач линейной алгебры, арифметики, дифференциальных уравнений и построения графиков. На протяжении всей книги автор демонстрирует, как писать программы ясно и выразительно, используя объектно-ориентированное, обобщенное и метапрограммирование, параллелизм и процедурные технологии.

ISBN 978-5-8459-2095-9 **в продаже**

JQUERY 2.0
ДЛЯ ПРОФЕССИОНАЛОВ

Адам Фримен



www.williamspublishing.com

Автор книги, Адам Фримен, делится с читателями секретами наиболее эффективных приемов работы с jQuery, фокусируя основное внимание на практических аспектах использования этой технологии и демонстрируя ее применение для решения реальных задач.
В этом поистине исчерпывающем руководстве вы найдете ответы на все вопросы, которые могут возникать у вас в процессе разработки веб-приложений на основе jQuery.
Благодаря подробному и тщательно продуманному изложению материала, дополненному многочисленными примерами готового работающего кода, демонстрирующими мощь и гибкость jQuery, эта книга поможет быстро приобрести знания и навыки, необходимые профессионалам в области веб-разработки.

ISBN 978-5-8459-1919-9 в продаже

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

Язык программирования C# 6.0 и платформа .NET 4.6

Новое 7-е издание этой книги было полностью пересмотрено и переписано с учетом последних изменений спецификации языка C# и новых достижений платформы .NET Framework. Отдельные главы посвящены важным новым средствам, которые делают .NET Framework 4.6 самым передовым выпуском, в том числе:

- усовершенствованная модель программирования ADO.NET Entity Framework
- многочисленные улучшения IDE-среды и архитектуры MVVM для разработки настольных приложений WPF
- многочисленные обновления в ASP.NET Web API

Помимо этого, предлагается исчерпывающее рассмотрение всех ключевых возможностей языка C#, как старых, так и новых, что позволило обрести популярность предыдущим изданиям этой книги. Читатели получат основательные знания приемов объектно-ориентированной разработки, атрибутов и рефлексии, обобщений и коллекций, а также обретут понимание многих сложных тем, которые не раскрываются в других источниках (таких, как коды операций CIL и выпуск динамических сборок).

Основная задача книги заключается в том, чтобы служить исчерпывающим руководством по языку программирования C# и ключевым аспектам платформы .NET, а также предоставлять обзорные сведения о технологиях, построенных на основе C# и .NET (ADO.NET и Entity Framework, Windows Communication Foundation (WCF), Windows Presentation Foundation (WPF) и ASP.NET (Web Forms, MVC, Web API)). Благодаря приведенной в книге информации у читателей появится возможность применять полученные знания при решении специфичных задач программирования и готовность к дальнейшим исследованиям мира .NET.

НА ВЕБ-САЙТЕ

Исходные коды всех примеров, рассмотренных в книге, можно загрузить с веб-сайта издательства по адресу: http://www.williamspublishing.com/Books/978-5-8459-2099-7.html

Категория: языки программирования
Предмет рассмотрения: язык C# 6.0
Уровень: для программистов средней и высокой квалификации

ISBN= 978-5-8459-2099-7

www.williamspublishing.com

Apress®
www.apress.com