

Библиотека профессионала

Java™

Том 1. Основы

Девятое издание

Core Java™

Volume I – Fundamentals

Ninth Edition

Cay S. Horstmann
Gary Cornell



Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Библиотека профессионала

JavaTM

Том 1. Основы

Девятое издание

Кей Хорстманн
Гари Корнелл



Москва • Санкт-Петербург • Киев
2014

ББК 32.973.26-018.2.75

X82

УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция И.В. Берштейна

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

Хорстманн, Кей С., Корнелл, Гари.

X82 Java. Библиотека профессионала, том 1. Основы. 9-е изд. : Пер. с англ. — М. :
ООО "И.Д. Вильямс", 2014. — 864 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1869-7 (рус., том 1)

ISBN 978-5-8459-1886-4 (рус., многотом.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Prentice Hall, Inc.

Authorized translation from the English language edition published by Prentice Hall, Inc., Copyright © 2013 Oracle and/or its affiliates.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2014

Научно-популярное издание

Кей С. Хорстманн, Гари Корнелл

Java. Библиотека профессионала, том 1. Основы 9-е издание

Литературный редактор *И.А. Попова*

Верстка *Л.В. Чернокозинская*

Художественный редактор *В.Г. Павлютин*

Корректор *Л.А. Гордиенко*

Подписано в печать 22.11.2013. Формат 70×100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 69,66. Уч.-изд. л. 47,67.

Тираж 1500 экз. Заказ № 3968.

Первая Академическая типография "Наука"

199034, Санкт-Петербург, 9-я линия, 12/28

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

ISBN 978-5-8459-1869-7 (рус., том 1)

ISBN 978-5-8459-1886-4 (рус., многотом.)

ISBN 978-0-13-708189-9 (англ.)

© Издательский дом "Вильямс", 2014

© Oracle and/or its affiliates, 2013

Оглавление



Предисловие	13
Благодарности	19
Глава 1. Введение в язык Java	21
Глава 2. Среда программирования на Java	37
Глава 3. Основные языковые конструкции Java	57
Глава 4. Объекты и классы	131
Глава 5. Наследование	197
Глава 6. Интерфейсы и внутренние классы	271
Глава 7. Программирование графики	313
Глава 8. Обработка событий	359
Глава 9. Компоненты пользовательского интерфейса в Swing	397
Глава 10. Развертывание приложений и апплетов	525
Глава 11. Исключения, утверждения, протоколирование и отладка	579
Глава 12. Обобщенное программирование	645
Глава 13. Коллекции	683
Глава 14. Многопоточная обработка	753
Приложение А. Ключевые слова Java	849
Предметный указатель	851

Содержание

Предисловие	13
Краткий обзор книги	15
Условные обозначения	17
Примеры исходного кода	17
Благодарности	19
От издательства	20
Глава 1. Введение в язык Java	21
Программная платформа Java	21
Характерные особенности Java	22
Простота	23
Объектно-ориентированный характер	23
Поддержка распределенных вычислений в сети	24
Надежность	24
Безопасность	25
Независимость от архитектуры компьютера	26
Переносимость	26
Интерпретируемость	27
Производительность	27
Многопоточность	27
Динамичность	28
Апплеты и Интернет	28
Краткая история развития Java	30
Распространенные заблуждения относительно Java	32
Глава 2. Среда программирования на Java	37
Установка Java Development Kit	38
Загрузка JDK	38
Задание пути к исполняемым файлам	39
Установка библиотек и документации	41
Установка примеров программ	42
Перемещение по каталогам	43
Выбор среды для разработки программ	44
Использование инструментов командной строки	44
Указания по выявлению ошибок	46
Применение ИСР	47
Выявление ошибок компиляции	50
Выполнение графического приложения	51
Построение и запуск апплетов	53
Глава 3. Основные языковые конструкции Java	57
Простая программа на Java	58
Комментарии	61
Типы данных	62
Целочисленные типы данных	62
Числовые типы данных с плавающей точкой	63
Тип данных char	64
Тип данных boolean	66
Переменные	66
Инициализация переменных	68
Константы	68

Операции	69
Операции инкрементирования и декрементирования	70
Операции отношения и логические операции	71
Поразрядные операции	72
Математические функции и константы	72
Преобразование числовых типов	74
Приведение типов	75
Скобки и иерархия операций	75
Перечислимые типы	76
Символьные строки	77
Подстроки	77
Сцепление	77
Принцип постоянства символьных строк	78
Проверка символьных строк на равенство	79
Пустые и нулевые строки	80
Кодовые точки и кодовые единицы	80
Прикладной интерфейс API класса String	81
Оперативно доступная документация на API	83
Построение символьных строк	86
Ввод и вывод	88
Чтение вводимых данных	88
Форматирование выводимых данных	90
Файловый ввод и вывод	95
Управляющая логика	96
Область действия блоков	97
Условные операторы	97
Неопределенные циклы	100
Определенные циклы	105
Оператор switch для многовариантного выбора	108
Операторы прерывания логики управления программой	111
Большие числа	113
Массивы	116
Цикл в стиле for each	117
Инициализация массивов и анонимные массивы	118
Копирование массивов	119
Параметры командной строки	120
Сортировка массива	121
Многомерные массивы	124
Неровные массивы	127
Глава 4. Объекты и классы	131
Введение в объектно-ориентированное программирование	132
Классы	133
Объекты	134
Идентификация классов	134
Отношения между классами	135
Применение предопределенных классов	137
Объекты и объектные переменные	137
Класс GregorianCalendar из библиотеки Java	140
Модифицирующие методы и методы доступа	142
Определение собственных классов	148
Класс Employee	149
Использование нескольких исходных файлов	151
Анализ класса Employee	152
Первые действия с конструкторами	153
Явные и неявные параметры	154
Преимущества инкапсуляции	155
Привилегии доступа к данным в классе	157
Закрытые методы	158
Неизменяемые поля экземпляра	158
Статические поля и методы	159
Статические поля	159

Статические константы	160
Статические методы	160
Фабричные методы	161
Метод <code>main()</code>	162
Параметры методов	164
Конструирование объектов	170
Перегрузка	170
Инициализация полей по умолчанию	171
Конструктор без аргументов	172
Явная инициализация полей	172
Имена параметров	173
Вызов одного конструктора из другого	174
Блоки инициализации	174
Уничтожение объектов и метод <code>finalize()</code>	179
Пакеты	179
Импорт классов	180
Статический импорт	181
Ввод классов в пакеты	182
Область действия пакетов	185
Путь к классам	186
Указание пути к классам	188
Комментарии и документирование	189
Вставка комментариев	190
Комментарии к классам	190
Комментарии к методам	191
Комментарии к полям	192
Комментарии общего характера	192
Комментарии к пакетам и обзорные	193
Извлечение комментариев	194
Рекомендации по разработке классов	194
Глава 5. Наследование	197
Классы, суперклассы и подклассы	198
Иерархии наследования	204
Полиморфизм	205
Динамическое связывание	206
Предотвращение наследования: конечные классы и методы	209
Приведение типов	210
Абстрактные классы	212
Защищенный доступ	217
Глобальный суперкласс <code>Object</code>	218
Метод <code>equals()</code>	219
Проверка равнозначности объектов и наследование	220
Метод <code>hashCode()</code>	224
Метод <code>toString()</code>	226
Обобщенные списочные массивы	232
Доступ к элементам списочных массивов	235
Совместимость типизированных и базовых списочных массивов	238
Объектные оболочки и автоупаковка	239
Методы с переменным числом параметров	243
Классы перечислений	244
Рефлексия	246
Класс <code>Class</code>	246
Основы обработки исключений	249
Анализ функциональных возможностей классов с помощью рефлексии	250
Анализ объектов во время выполнения с помощью рефлексии	256
Написание кода обобщенного массива с помощью рефлексии	260
Вызов произвольных методов	264
Рекомендации по применению наследования	267

Глава 6. Интерфейсы и внутренние классы	271
Интерфейсы	272
Свойства интерфейсов	278
Интерфейсы и абстрактные классы	279
Клонирование объектов	280
Интерфейсы и обратные вызовы	286
Внутренние классы	289
Доступ к состоянию объекта с помощью внутреннего класса	291
Специальные синтаксические правила для внутренних классов	294
О пользе, необходимости и безопасности внутренних классов	295
Локальные внутренние классы	298
Доступ к конечным переменным из внешних методов	298
Анонимные внутренние классы	301
Статические внутренние классы	304
Прокси-классы	307
Свойства прокси-классов	311
Глава 7. Программирование графики	313
Общие сведения о библиотеке Swing	314
Создание фрейма	319
Расположение фрейма	322
Свойства фрейма	324
Определение подходящих размеров фрейма	324
Отображение данных в компоненте	329
Двухмерные формы	334
Окрашивание цветом	343
Специальное шрифтовое оформление текста	346
Вывод изображений	354
Глава 8. Обработка событий	359
Общее представление об обработке событий	359
Пример обработки событий от щелчков на кнопке	361
Овладение внутренними классами	366
Создание приемников событий с единственным вызовом метода	368
Пример изменения визуального стиля	370
Классы адаптеров	373
Действия	377
События от мыши	385
Иерархия событий в библиотеке AWT	392
Семантические и низкоуровневые события	392
Глава 9. Компоненты пользовательского интерфейса в Swing	397
Библиотека Swing и шаблон проектирования “модель–представление–контроллер”	398
Шаблоны проектирования	398
Шаблон проектирования “модель–представление–контроллер”	399
Анализ кнопок в Swing по шаблону “модель–представление–контроллер”	403
Введение в компоновку пользовательского интерфейса	405
Граничная компоновка	408
Сеточная компоновка	410
Ввод текста	414
Текстовые поля	414
Метки и пометка компонентов	416
Поля для ввода пароля	418
Текстовые области	418
Панели прокрутки	419
Компоненты для выбора разных вариантов	421
Флажки	422
Кнопки-переключатели	424
Рамки	427
Комбинированные списки	432
Регулируемые ползунки	435

Меню	441
Создание меню	442
Пиктограммы в пунктах меню	444
Пункты меню с флажками и кнопками-переключателями	445
Всплывающие меню	446
Клавиши быстрого доступа и оперативные клавиши	448
Разрешение и запрет доступа к пунктам меню	450
Панели инструментов	454
Всплывающие подсказки	457
Расширенные средства компоновки	458
Диспетчер сеточно-контейнерной компоновки	460
Диспетчер групповой компоновки	470
Компоновка без диспетчера	480
Специальные диспетчеры компоновки	480
Порядок обхода компонентов	484
Диалоговые окна	485
Диалоговые окна для выбора разных вариантов	486
Создание диалоговых окон	496
Обмен данными	501
Диалоговые окна для обращения с файлами	507
Диалоговые окна для выбора цвета	518
Глава 10. Развертывание приложений и апплетов	525
Файлы формата JAR	526
Файл манифеста	527
Исполняемые JAR-файлы	528
Ресурсы	529
Герметизация пакетов	532
Технология Java Web Start	533
"Песочница"	537
Подписанный код	538
Прикладной интерфейс JNLP API	540
Апплеты	549
Простой апплет	549
HTML-дескриптор <applet> и его атрибуты	554
Дескриптор <object>	557
Передача данных апплетам через параметры	557
Обращение к файлам изображений и звуковым файлам	563
Контекст апплета	564
Сохранение глобальных параметров настройки приложений	567
Таблица свойств	567
Прикладной интерфейс API для сохранения глобальных параметров настройки	572
Глава 11. Исключения, утверждения, протоколирование и отладка	579
Обработка ошибок	580
Классификация исключений	582
Объявление проверяемых исключений	583
Порядок генерирования исключений	586
Создание классов исключений	587
Перехват исключений	588
Повторное генерирование исключений	591
Блок finally	592
Оператор try с ресурсами	596
Анализ элементов трассировки стека	597
Рекомендации по обработке исключений	600
Применение утверждений	603
Разрешение и запрет утверждений	604
Проверка параметров с помощью утверждений	605
Документирование предположений с помощью утверждений	606
Протоколирование	608
Элементарное протоколирование	608
Усовершенствованное протоколирование	608

Смена диспетчера протоколирования	611
Интернационализация	612
Обработчики протоколов	613
Фильтры	616
Средства форматирования	616
“Рецепт” протоколирования	617
Рекомендации по отладке программ	626
Рекомендации по отладке программ с ГПИ	631
Применение робота AWT	634
Применение отладчика	639
Глава 12. Обобщенное программирование	645
Назначение обобщенного программирования	646
На кого рассчитано обобщенное программирование	647
Определение простого обобщенного класса	648
Обобщенные методы	650
Ограничения на переменные типа	651
Обобщенный код и виртуальная машина	653
Преобразование обобщенных выражений	655
Преобразование обобщенных методов	655
Вызов унаследованного кода	657
Ограничения и пределы обобщений	658
Параметрам типа нельзя приписывать простые типы	658
Во время выполнения можно запрашивать только базовые типы	658
Массивы параметризованных типов недопустимы	659
Предупреждения о переменном числе аргументов	660
Нельзя создавать экземпляры переменных типа	661
Переменные типа в статическом контексте обобщенных классов недействительны	662
Нельзя генерировать или перехватывать экземпляры обобщенного класса в виде исключений	663
Остерегайтесь конфликтов после стирания типов	665
Правила наследования обобщенных типов	666
Подстановочные типы	668
Ограничения супертипа на подстановки	670
Неограниченные подстановки	672
Захват подстановок	673
Рефлексия и обобщения	675
Сопоставление типов с помощью параметров <code>Class<T></code>	676
Сведения об обобщенных типах в виртуальной машине	676
Глава 13. Коллекции	683
Интерфейсы коллекций	683
Разделение интерфейсов и реализаций коллекций	684
Интерфейсы <code>Collection</code> и <code>Iterator</code> из библиотеки <code>Java</code>	686
Конкретные коллекции	692
Связные списки	693
Списочные массивы	702
Хеш-множества	702
Древовидные множества	706
Сравнение объектов	707
Односторонние и двухсторонние очереди	713
Очереди по приоритету	715
Отображения	716
Классы специализированных множеств и отображений	720
Архитектура коллекций	725
Представления и оболочки	729
Групповые операции	736
Взаимное преобразование коллекций и массивов	737
Алгоритмы	737
Сортировка и перетасовка	739
Двоичный поиск	741
Простые алгоритмы	743

Написание собственных алгоритмов	744
Унаследованные коллекции	746
Класс Hashtable	746
Перечисления	746
Таблицы свойств	747
Стеки	748
Битовые множества	749
Реализация алгоритма "Решето Эратосфена"	750
Глава 14. Многопоточная обработка	753
Назначение потоков	754
Предоставление возможности для выполнения других задач с помощью потоков	759
Прерывание потоков	765
Состояния потоков	767
Новые потоки	768
Исполняемые потоки	768
Блокированные и ожидающие потоки	768
Завершенные потоки	769
Свойства потоков	771
Приоритеты потоков	771
Потоковые демоны	772
Обработчики необрабатываемых исключений	772
Синхронизация	774
Пример состояния гонок	774
Объяснение причин, приводящих к состоянию гонок	778
Объекты блокировки	780
Объекты условий	783
Ключевое слово synchronized	788
Синхронизированные блоки	792
Принцип монитора	793
Поля и переменные типа volatile	794
Поля и переменные типа final	796
Атомарность операций	796
Взаимные блокировки	796
Локальные переменные потоков	799
Проверка блокировок и время ожидания	800
Блокировки чтения/записи	802
Причины, по которым методы stop () и suspend () не рекомендованы к применению	803
Блокирующие очереди	805
Потокобезопасные коллекции	813
Эффективные отображения, множества и очереди	813
Копируемые при записи массивы	815
Устаревшие потокобезопасные коллекции	815
Интерфейсы Callable и Future	816
Исполнители	821
Пулы потоков	821
Плановое выполнение потоков	826
Управление группами задач	826
Архитектура вилочного соединения	828
Синхронизаторы	831
Семафоры	831
Защелки с обратным отсчетом	832
Барьеры	832
Обменники	833
Синхронные очереди	833
Потоки и библиотека Swing	833
Выполнение продолжительных задач	835
Применение класса SwingWorker	839
Правило единственного потока	846
Приложение А. Ключевые слова Java	849
Предметный указатель	851

Предисловие

В конце 1995 года язык программирования Java вырвался на арену Интернета и моментально завоевал популярность. Технология Java обещала стать *универсальным связующим звеном*, соединяющим пользователей с информацией, откуда бы эта информация ни поступала — от веб-серверов, из баз данных, поставщиков информации или любого другого источника, который только можно было себе вообразить. И действительно, у Java есть все, чтобы выполнить эти обещания. Это весьма основательно сконструированный язык, получивший признание всех основных участников рынка за исключением корпорации Microsoft. Его встроенные средства защиты и безопасности оснастили как программистов, так и пользователей программ на Java. Язык Java изначально обладал встроенной поддержкой для решения таких сложных задач, как сетевое программирование, взаимодействие с базами данных и многопоточная обработка.

С 1995 года было выпущено восемь главных версий комплекта Java Development Kit. За последние 17 лет прикладной программный интерфейс (API) увеличился от 200 до 3 тысяч классов. Теперь прикладной интерфейс API охватывает самые разные предметные области, включая конструирование пользовательских интерфейсов, управление базами данных, интернационализация, безопасность и обработка данных в формате XML.

Книга, которую вы держите в руках, является первым томом девятого издания. С выходом каждого издания мы, ее авторы, старались как можно быстрее следовать очередному выпуску Java Development Kit, каждый раз переписывая ее, чтобы вы, читатель, могли воспользоваться преимуществами новейших средств Java. Настоящее издание обновлено с учетом новых языковых средств, появившихся в версии Java Standard Edition (SE) 7.

Как и все предыдущие издания этой книги, мы по-прежнему *адресуем ее серьезным программистам, которые хотели бы пользоваться Java для разработки настоящих проектов*. Мы, авторы этой книги, представляем себе вас, дорогой читатель, как грамотного специалиста с солидным опытом программирования на других языках, кроме Java, и надеемся, что вам не нравятся книги, которые полны игрушечных примеров вроде программ управления тостерами или животными в зоопарке либо “прыгающим текстом”. Ничего подобного вы не найдете в этой книге. Наша цель — помочь вам понять язык Java и его библиотеки в полной мере, а не создать иллюзию такого понимания.

В книге вы найдете массу примеров кода, демонстрирующих почти все обсуждаемые языковые и библиотечные средства. Эти примеры мы намеренно сделали как можно более простыми, чтобы сосредоточиться на основных моментах. Тем не менее они в большинстве своем совсем не игрушечные и не “срезают острых углов”. Все они могут послужить вам неплохой отправной точкой для разработки собственного кода.

Мы предполагаем, что вы стремитесь (и даже жаждете) узнать обо всех расширенных средствах, которые Java предоставляет в ваше распоряжение. В частности, мы предлагаем подробное рассмотрение следующих тем.

- Объектно-ориентированное программирование.
- Рефлексия и прокси-классы.
- Интерфейсы и внутренние классы.
- Модель приемников событий.
- Проектирование графического пользовательского интерфейса инструментальными средствами библиотеки Swing.
- Обработка исключений.
- Обобщенное программирование.
- Архитектура коллекций.
- Многопоточное программирование.

В связи со стремительным ростом библиотеки классов Java одного тома оказалось недостаточно для описания всех языковых средств Java, о которых следует знать серьезным программистам. Поэтому книга была разделена на два тома. В первом томе, который вы держите в руках, главное внимание уделяется фундаментальным понятиям языка Java, а также основам программирования пользовательского интерфейса. А второй том посвящен средствам разработки приложений масштаба предприятия и усовершенствованному программированию пользовательских интерфейсов. В нем вы найдете подробное обсуждение следующих тем.

- Файлы и потоки ввода-вывода.
- Распределенные объекты.
- Базы данных.
- Расширенные компоненты графического пользовательского интерфейса.
- Собственные методы.
- Обработка данных в формате XML.
- Сетевое программирование.
- Усовершенствованная графика.
- Интернационализация.
- Компонент JavaBeans.
- Аннотации.

При написании книги ошибки и неточности неизбежны. И нам очень важно знать о них. Но мы бы, конечно, предпочли узнать о каждой из них только один раз. Поэтому мы разместили перечень часто задаваемых вопросов, исправлений ошибок и обходных приемов по адресу <http://horstmann.com/corejava>, куда вы можете обращаться за справкой.

Краткий обзор книги

В главе 1 дается краткий обзор тех функциональных возможностей языка Java, которыми он отличается от других языков программирования. В ней сначала поясняется, что было задумано разработчиками Java и в какой мере им удалось воплотить задуманное в жизнь. Затем приводится краткая история развития языка Java и показывается, как он стал тем, чем он есть в настоящее время.

В главе 2 сначала поясняется, как загрузить и установить инструментальный JDK, а также примеры программ к этой книге. Затем рассматривается весь процесс компиляции и запуска трех типичных программ на Java (консольного приложения, графического приложения и апплета) только средствами JDK, текстового редактора, специально ориентированного на Java, а также интегрированной среды разработки на Java.

В главе 3 начинается обсуждение языка программирования Java. В этой главе излагаются самые основы: переменные, циклы и простые функции. Если у вас имеется опыт программирования на C или C++, вам нетрудно будет усвоить материал этой главы, поскольку синтаксис этих языковых средств, по существу, ничем не отличается в Java. А если вам приходилось программировать на языках, не похожих на C, например на Visual Basic, прочитайте эту главу с особым вниманием.

Ныне объектно-ориентированное программирование (ООП) — господствующая методика программирования, и ей в полной мере отвечает язык Java. В главе 4 представлено понятие *инкапсуляции* — первой из двух фундаментальных составляющих объектной ориентации, а также механизмы, реализующие ее в Java: классы и методы. В дополнение к правилам языка Java здесь также приводятся рекомендации по правильному объектно-ориентированному проектированию. И, наконец, в этой главе будет представлен замечательный инструмент *javadoc*, форматирующий комментарии из исходного кода в набор веб-страниц с перекрестными ссылками. Если у вас имеется опыт программирования на C++, можете лишь бегло просмотреть эту главу. А тем, кому раньше не приходилось программировать на объектно-ориентированных языках, придется потратить больше времени на усвоение принципов ООП, прежде чем изучать Java дальше.

Классы и инкапсуляция — это лишь часть методики ООП, и поэтому в главе 5 представлен еще один ее краеугольный камень — *наследование*. Наследование позволяет взять существующий класс и модифицировать его в соответствии с конкретными потребностями программирующего. Это — основополагающий прием программирования на Java. Механизм наследования в Java очень похож на аналогичный механизм в C++. Опять же программирующие на C++ могут сосредоточить основное внимание лишь на языковых отличиях в реализации наследования.

В главе 6 поясняется, как пользоваться в Java понятием *интерфейса*. Интерфейсы дают возможность выйти за пределы простого наследования, описанного в главе 5. Овладение интерфейсами позволит в полной мере воспользоваться объектно-ориентированным подходом к программированию на Java. В этой главе рассматривается также удобное языковое средство Java, называемое *внутренними классами*. Внутренние классы помогают сделать код более ясным и кратким.

С главы 7 начинается серьезное прикладное программирование. Всякий программирующий на Java должен знать хотя бы немного о программировании графических пользовательских интерфейсов, и в этом томе вы найдете его основы. В этой главе будет показано, как создаются окна, как в них выполняется раскраска, рисуются

геометрические фигуры, форматируется текст многими шрифтами и как изображения выводятся на экран.

В главе 8 подробно обсуждается модель событий AWT — *абстрактного оконного инструментария*. Здесь будет показано, как писать код, реагирующий на такие события, как щелчки кнопкой мыши или нажатия клавиш. Попутно вам предстоит ознакомиться с тем, как правильно обращаться с базовыми элементами графического пользовательского интерфейса вроде кнопок и панелей.

Глава 9 посвящена более подробному обсуждению инструментальных средств Swing. Набор инструментов Swing позволяет строить межплатформенный графический пользовательский интерфейс. В этой главе вы ознакомитесь с различными видами кнопок, текстовых компонентов, рамок, ползунков, комбинированных списков, меню и диалоговых окон. Но знакомство с некоторыми из более совершенных компонентов будет отложено до второго тома настоящего издания.

В главе 10 будет показано, как разворачивать прикладные программы, будь то приложения или апплеты. В частности, в ней описывается, как пакетировать программы в файлы формата JAR и как доставлять приложения через Интернет с помощью технологии Java Web Start и механизмов апплетов. И, наконец, в этой главе поясняется, каким образом программы на Java способны сохранять и извлекать информацию о конфигурации после своего разворачивания.

Глава 11 посвящена *обработке исключений* — надежному механизму Java, призванному учитывать тот факт, что непредвиденные ситуации могут возникать и в грамотно написанных программах. Исключения обеспечивают эффективный способ отделения кода нормальной обработки от кода обработки ошибок. Но даже после оснащения прикладной программы проверкой всех возможных исключительных ситуаций в ней все-таки может произойти неожиданный сбой. Во второй части этой главы будет представлено немало полезных советов по организации отладки программ. И, наконец, в главе рассматривается весь процесс отладки на конкретном примере.

В главе 12 дается краткий обзор *обобщенного программирования* — главного преимущества версии Java SE 5.0. Обобщенное программирование делает прикладные программы легче читаемыми и более безопасными. В этой главе будет показано, как применяется строгая типизация, исключается потребность в неприглядном и небезопасном приведении типов и как преодолеваются трудности на пути совместимости с предыдущими версиями Java.

Глава 13 посвящена архитектуре коллекций на платформе Java. Всякий раз, когда требуется сначала собрать множество объектов, а в дальнейшем извлечь их, приходится обращаться к коллекции, которая наилучшим образом подходит для конкретных условий, вместо того чтобы сбрасывать их в обычный массив. В этой главе будут продемонстрированы те преимущества, которые дают стандартные, предварительно подготовленные коллекции.

Глава 14 завершает первый том настоящего издания обсуждением многопоточной обработки, которая позволяет программировать выполняемые параллельно задачи. (Под *поток* здесь понимается поток управления командами в программе.) В этой главе будет показано, как создаются потоки и осуществляется их синхронизация. Средства многопоточной обработки существенно изменились после выхода версии Java SE 5.0, и в этой главе рассказывается о ее новых механизмах.

В приложении А перечислены зарезервированные слова языка Java.

Условные обозначения

Как это принято во многих компьютерных книгах, моноширинный шрифт используется для представления исходного кода.



НА ЗАМЕТКУ! Этой пиктограммой выделяются замечания.



СОВЕТ. Этой пиктограммой выделяются советы.



ВНИМАНИЕ! Этой пиктограммой выделяются предупреждения о потенциальной опасности.



НА ЗАМЕТКУ C++! В этой книге имеется немало примечаний к синтаксису C++, где разъясняются отличия между языками Java и C++. Вы можете пропустить их, если у вас нет опыта программирования на C++ или же если вы склонны воспринимать этот опыт как страшный сон, который лучше забыть.

Язык Java сопровождается огромной библиотекой в виде прикладного программного интерфейса (API). При упоминании вызова какого-нибудь метода из прикладного интерфейса API в первый раз в конце соответствующего раздела приводится его краткое описание. Эти описания не слишком информативны, но, как мы надеемся, более содержательны, чем те, что представлены в официальной оперативно доступной документации на прикладной интерфейс API. Имена интерфейсов выделены *полужирным*, как это делается в официальной документации. А число после имени класса, интерфейса или метода обозначает версию JDK, в которой данное средство было внедрено, как показано ниже.

Название прикладного программного интерфейса 1.2

Программы с доступным исходным кодом организованы в виде примеров, как показано ниже.

Листинг 1.1. Исходный код из файла InputTest/InputTest.java

Примеры исходного кода

Все примеры исходного кода, приведенные в этой книге, доступны в архивированном виде на посвященном ей веб-сайте по адресу <http://horstmann.com/corejava>. Их можно извлечь из файла архива любой из распространенных программ разархивирования или с помощью утилиты `jar`, входящей в состав набора Java Development Kit (JDK). Подробнее об установке JDK и примеров кода речь пойдет в главе 2.

Благодарности

Написание книги всегда требует значительных усилий, а ее переписывание не намного легче, особенно если учесть постоянные изменения в технологии Java. Чтобы сделать книгу полезной, необходимы совместные усилия многих преданных делу людей, и мы с удовольствием выражаем признательность всем, кто внес свой посильный вклад в наше общее дело.

Большое число сотрудников издательств Prentice Hall оказали неоценимую помощь, хотя и остались в тени. Я хотел бы выразить им свою признательность за их усилия. Как всегда, самой горячей благодарности заслуживает мой редактор из издательства Prentice Hall Грег Доеич (Greg Doench) — за сопровождение книги на протяжении всего процесса ее написания и издания, а также за то, что он позволил мне пребывать в блаженном неведении относительно многих скрытых деталей этого процесса. Я благодарен Джули Нахил (Julie Nahil) за оказанную помощь к подготовке книги к изданию, а также Дмитрию и Алине Кирсановым — за литературное редактирование и набор рукописи книги. Приношу также свою благодарность моему соавтору по прежним изданиям Гари Корнеллу (Gary Cornell), который с тех пор обратился к другим занятиям.

Выражаю большую признательность многим читателям прежних изданий, которые сообщали о найденных ошибках и внесли массу ценных предложений по улучшению книги. Я особенно благодарен блестящему коллективу рецензентов, которые тщательно просмотрели рукопись книги, устранив в ней немало досадных ошибок.

Среди рецензентов этого и предыдущих изданий хотелось бы отметить Чака Аллисона (Chuck Allison), выпускающего редактора издания *C/C++ Users Journal*, Ланса Андерсона (Lance Anderson, Oracle), Алека Битона (Alec Beaton, PointBase, Inc.), Клиффа Берга (Cliff Berg), Джошуа Блоха (Joshua Bloch), Дэвида Брауна (David Brown), Корки Картрайта (Corky Cartwright), Френка Коена (Frank Cohen, PushToTest), Криса Крейна (Chris Crane, devXsolution), доктора Николаса Дж. Де Лилло (Dr. Nicholas J. De Lillo) из Манхэттенского колледжа, Ракеша Дхупара (Rakesh Dhoopar, Oracle), David Geary (David Geary, Sabreware), Джима Гиша (Oracle), Брайана Гоецца (Brian Goetz) из компании Oracle, Анжелу Гордон (Angela Gordon), Дэна Гордона (Dan Gordon), Роба Гордона (Rob Gordon), Джона Грэя (John Gray) из Хартфордского университета, Камерона Грегори (Cameron Gregory, olabs.com), Марти Холла (Marty Hall) из лаборатории прикладной физики в университете имени Джона Хопкинса, Винсента Харди (Vincent Hardy) из компании Adobe Systems, Дэна Харки (Dan Harkey) из университета штата Калифорния в Сан-Хосе, Вильяма Хиггинса (William Higgins, IBM), Владимира Ивановича (Vladimir Ivanovic) из PointBase, Джерри Джексона (Jerry Jackson) из CA Technologies, Тима Киммета (Tim Kimmet) из Walmart, Криса Лаффра (Chris Laffra), Чарли Лаи (Charlie Lai) из компании Apple, Анжелику Лангер (Angelika Langer), Дуга Лэнгстона (Doug Langston), Ханг Лау (Hang Lau) из университета имени Макгилла, Марка Лоуренса (Mark Lawrence), Дуга Ли (Doug Lea)

из SUNY Oswego, Грегори Лонгшора (Gregory Longshore), Боба Линча (Bob Lynch) из Lynch Associates, Филиппа Милна (Philip Milne, консультанта), Марка Моррисси (Mark Morrissey) из научно-исследовательского института штата Орегон, Махеш Нилаканта (Mahesh Neelakanta) из Атлантического университета штата Флорида, Хао Фам (Hao Pham), Пола Филиона (Paul Philion), Блейка Рагсдейла (Blake Ragsdell), Стюарта Реджеса (Stuart Reges) из университета штата Аризона, Рича Росена (Rich Rosen) из Interactive Data Corporation, Питера Сандерса (Peter Sanders) из университета ЭССИ (ESSI), г. Ницца, Франция, доктора Пола Сангеру (Dr. Paul Sanghera) из университета штата Калифорния в Сан-Хосе и колледжа имени Брукса, Поля Сэвинка (Paul Sevinc) из Teamup AG, Деванг Ша (Devang Shah) из Oracle, Бредли А. Смита (Bradley A. Smith), Стивена Стелтинга (Steven Stelting) из Oracle, Кристофера Тэйлора (Christopher Taylor), Люка Тэйлора (Luke Taylor) из Valtech, Джорджа Тхирува-тукала (George Thiruvathukal), Кима Топли (Kim Topley) из StreamingEdge, Жанет Трауб (Janet Traub), Пола Тиму (Paul Тума, консультанта), Питера Ван Дер Линдена (Peter van der Linden) из Motorola Mobile Devices, Берта Уолша (Burt Walsh), Дана Ксю (Dan Xu) из Oracle и Джона Завгрена (John Zavgren) из Oracle.

*Кей Хорстманн,
Сан-Франциско, сентябрь 2012 г.*

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 электронные адреса:

E-mail: info@williamspublishing.com
WWW: <http://www.williamspublishing.com>

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д.43, стр. 1
в Украине: 03150, Киев, а/я 152

Введение в язык Java

В этой главе...

- ▶ Программная платформа Java
- ▶ Характерные особенности Java
- ▶ Апплеты Java и Интернет
- ▶ Краткая история развития Java
- ▶ Распространенные заблуждения относительно Java

На появление первой версии Java в 1996 году откликнулись не только специализирующиеся на вычислительной технике газеты и журналы, но даже такие солидные издания, как *The New York Times*, *The Washington Post* и *Business Week*. Java — единственный язык программирования, удостоившийся десятиминутного репортажа на Национальном общественном радио в США. Для разработки и сопровождения программных продуктов *только* на этом языке программирования был учрежден венчурный фонд в 100 млн. долларов. Это было удивительное время. Тем временам и последующей истории развития языка Java посвящена эта глава.

Программная платформа Java

В первом издании этой книги о Java было сказано следующее: “Как язык программирования, Java перевыполнил рекламные обещания. Определенно Java — хороший язык программирования. Несомненно, это один из лучших языков, доступных серьезным программистам. Потенциально Java имел все предпосылки, чтобы стать великим языком программирования, но теперь время для этого уже, вероятно, упущено. Как только появляется новый язык программирования, сразу же возникает неприятная проблема его совместимости с созданным раньше программным обеспечением”.

По поводу этого абзаца наш редактор долго спорил с одним из руководителей компании Sun Microsystems. Но и сейчас, по прошествии длительного времени, такая

оценка кажется нам правильной. Действительно, Java обладает целым рядом преимуществ, о которых мы поговорим чуть позже. Но более поздние дополнения далеко не так изящны, как исходный вариант этого языка программирования, и виной тому пресловутые требования совместимости.

Как уже отмечалось в первом издании книги, Java никогда не был только языком. Хорошие языки — не редкость, а появление некоторых из них вызвало в свое время настоящую сенсацию в области вычислительной техники. В отличие от них, Java — это программная *платформа*, включающая в себя мощную библиотеку, большой объем кода, пригодного для повторного использования, а также среду для выполнения программ, которая обеспечивает безопасность, независимость от операционной системы и автоматическую “сборку мусора”.

Программистам нужны языки с четкими синтаксическими правилами и понятной семантикой (т.е. определено не C++). Такому требованию, помимо Java, отвечают десятки языков. Некоторые из них даже обеспечивают переносимость и “сборку мусора”, но их библиотеки оставляют желать много лучшего. В итоге программисты вынуждены самостоятельно реализовывать графические операции, доступ к сети и базе данных и другие часто встречающиеся процедуры. Java объединяет в себе прекрасный язык, высококачественную среду выполнения программ и обширную библиотеку. В результате многие программисты остановили свой выбор именно на Java.

Характерные особенности Java

Создатели Java написали руководство, в котором объяснялись цели и достоинства нового языка. В этом документе приведено одиннадцать характерных особенностей Java. Этот язык

- простой;
- объектно-ориентированный;
- распределенный;
- надежный;
- безопасный;
- не зависящий от архитектуры компьютера;
- переносимый;
- интерпретируемый;
- высокопроизводительный;
- многопоточный;
- динамичный.

В данном разделе мы

- приведем цитаты из руководства по Java, раскрывающие особенности этого языка программирования;
- поделимся с читателями соображениями по поводу отдельных характеристик языка Java, основываясь на собственном опыте работы с его текущей версией.



НА ЗАМЕТКУ! Руководство, о котором идет речь, можно найти по адресу www.oracle.com/technetwork/java/langenv-140151.html. Там же описаны характерные особенности Java.

Простота

“Мы хотели создать систему, которая позволяла бы легко писать программы, не требовала дополнительного обучения и учитывала сложившуюся практику и стандарты программирования. Мы считали C++ не совсем подходящим для этих целей, но чтобы сделать систему более доступной, язык Java был разработан как можно более похожим на C++. А исключили мы лишь редко используемые, малопонятные и невразумительные средства C++, которые, по нашему мнению, приносят больше вреда, чем пользы”.

Синтаксис Java, по существу, представляет собой упрощенный вариант синтаксиса C++. В этом языке отсутствует потребность в файлах заголовков, арифметике (и даже в синтаксисе) указателей, структурах, объединениях, перегрузке операций, виртуальных базовых классах и т.п. (Отличия Java от C++ упоминаются на протяжении всей книги в специальных врезках.) Но создатели Java не стремились исправить все недостатки языка C++. Например, синтаксис оператора `switch` в Java остался неизменным. Зная C++, нетрудно перейти к Java.

Если вы привыкли к визуальной среде программирования (например, Visual Basic), язык Java может показаться вам сложным. Его синтаксис некоторые считают странным, хотя понять его не составляет большого труда. Но важнее другое: пользуясь Java, приходится больше программировать. Начинающие программисты нередко предпочитают язык Visual Basic, поскольку его визуальная среда программирования позволяет почти автоматически создавать инфраструктуру приложения. Чтобы достичь того же результата средствами Java, придется вручную написать довольно большой объем кода. Существуют, однако, интегрированные среды разработки (ИСР) от независимых производителей, которые позволяют программировать в стиле перетаскивания.

“Другой аспект простоты — краткость. Одна из целей языка Java — обеспечить разработку независимых программ, способных выполняться на машинах с ограниченным объемом ресурсов. Размер основного интерпретатора и средств поддержки классов составляет около 40 Кбайт; стандартные библиотеки и средства поддержки потоков, в том числе автономное микроядро, занимают еще 175 Кбайт”.

На то время это было впечатляющим достижением. Разумеется, с тех пор библиотеки разрослись до гигантских размеров. Но теперь существует отдельная платформа Java Micro Edition с компактными библиотеками, более подходящая для встроенных устройств.

Объектно-ориентированный характер

“По существу, объектно-ориентированное проектирование — это методика программирования, в центре внимания которой находятся данные (т.е. объекты) и интерфейсы этих объектов. Проводя аналогию со столярным делом, можно сказать, что “объектно-ориентированный” мастер сосредоточен в первую очередь на стуле, который он изготавливает, и лишь во вторую очередь его интересуют необходимые для этого инструменты; в то же время “не объектно-ориентированный” столяр думает в первую очередь о своих инструментах. Объектно-ориентированные средства Java и C++ по существу совпадают”.

За прошедшие тридцать лет объектно-ориентированный подход уже доказал свое право на жизнь, и без него невозможно представить себе современный язык программирования. Действительно, особенности Java, связанные с объектами, сравнимы с языком C++. Основное отличие между ними заключается в механизме множественного наследования, который в языке Java заменен более простым понятием интерфейсов, а также в модели метаклассов Java (о которых речь пойдет в главе 5).



НА ЗАМЕТКУ! Если у вас нет опыта программирования на объектно-ориентированных языках, внимательно изучите материал глав 4–6. В них излагаются основы объектно-ориентированного подхода и демонстрируются его преимущества над традиционными, процедурно-ориентированными средствами программирования на таких языках, как C или Basic. Эти преимущества наглядно проявляются при разработке сложных проектов.

Поддержка распределенных вычислений в сети

“Язык Java предоставляет разработчику обширную библиотеку программ для передачи данных по протоколу TCP/IP, HTTP и FTP. Приложения на Java способны открывать объекты и получать к ним доступ по сети с такой же легкостью, как и в локальной файловой системе, используя URL для адресации”.

Язык Java предоставляет эффективные и удобные средства для работы в сети. Всякий, когда-либо пытавшийся писать программы на других языках для работы в Интернете, будет приятно удивлен тем, насколько просто на Java решаются самые трудные задачи, например установление сокетных соединений. (Работа в сети будет подробно обсуждаться во втором томе данной книги.) Связь между распределенными объектами в Java обеспечивается механизмом вызова удаленных методов (эта тема также рассматривается во втором томе).

Надежность

“Язык Java предназначен для написания программ, которые должны надежно работать в любых условиях. Основное внимание в этом языке уделяется раннему обнаружению возможных ошибок, контролю в процессе выполнения программы, а также устранению ситуаций, которые могут вызвать ошибки... Единственное существенное отличие языка Java от C++ кроется в модели указателей, принятой в Java, которая исключает возможность записи в произвольно выбранную область памяти и повреждения данных”.

И эта характеристика языка очень полезна. Компилятор Java выявляет такие ошибки, которые в других языках обнаруживаются только на этапе выполнения программы. Кроме того, программисты, потратившие многие часы на поиски ошибки, вызвавшей нарушения данных в памяти из-за неверного указателя, будут обрадованы тем, что в работе с Java подобные осложнения не могут даже в принципе возникнуть.

Если вам приходилось раньше программировать на Visual Basic или другом языке, где указатели явным образом не применяются, то у вас может возникнуть недоумение, почему этот вопрос настолько важен. Программирующим на C нужны указатели для доступа к символьным строкам, массивам, объектам и даже файлам.

При программировании на Visual Basic ничего подобного не требуется, и разработчик может не беспокоиться о распределении памяти для хранения объектов. С другой стороны, многие структуры данных в языке, не имеющем указателей, реализовать очень трудно. В Java выбрана золотая середина. Для простых конструкций, вроде символьных строк и массивов, указатели не нужны. Но в то же время указателями можно в полной мере воспользоваться там, где без них нельзя обойтись, например, при составлении связных списков. Программирующий на Java полностью защищен, поскольку он никогда не обратится к неправильному указателю, не допустит ошибок выделения памяти, и ему не придется искать причины утечек памяти.

Безопасность

“Язык Java предназначен для использования в сетевой или распределенной среде. По этой причине большое внимание было уделено безопасности. Java позволяет создавать системы, защищенные от вирусов и несанкционированного доступа”.

В первом издании мы предостерегали: “Никогда не говорите никогда”, и оказались правы. Вскоре после выхода первой версии Java Development Kit группа экспертов по вопросам безопасности из Принстонского университета обнаружила первые ошибки в системе защиты Java 1.0. Компания Sun Microsystems развернула исследования в области безопасности программ на Java. В частности, она обнародовала спецификацию и код виртуальной машины и библиотек, ответственных за защиту. Это ускорило выявление и устранение ошибок. Так или иначе, создатели Java сильно затруднили задачу обхода ее механизмов безопасности. Обнаруженные с тех пор ошибки носили лишь технический характер и были совсем не многочисленны.

Ниже перечислены некоторые виды нарушения защиты, которые с самого начала предотвращает система безопасности Java.

- Намеренное переполнение стека выполняемой программы — один из распространенных способов нарушения защиты, используемых вирусами и “червями”.
- Повреждение данных на участках памяти, находящихся за пределами пространства, выделенного процессу.
- Несанкционированное чтение файлов и их модификация.

Со временем в Java был добавлен ряд новых средств защиты. Так, в версии Java 1.1 появилось понятие классов с цифровой подписью (подробнее об этом — во втором томе). Пользуясь таким классом, вы получаете сведения об его авторе. Если вы доверяете автору класса, то можете предоставить этому классу все необходимые полномочия на своей машине.



НА ЗАМЕТКУ! Альтернативный механизм доставки кода, предложенный корпорацией Microsoft, опирается на технологию ActiveX, где ради безопасности используются только цифровые подписи. Очевидно, что этого недостаточно — любой пользователь программного обеспечения корпорации Microsoft может подтвердить, что даже программы широко известных производителей часто завершаются аварийно, создавая тем самым опасность повреждения данных. Система безопасности в языке Java намного надежнее, чем технология ActiveX, поскольку она контролирует приложение в процессе его выполнения и не позволяет ему произвести какие-либо разрушительные действия.

Независимость от архитектуры компьютера

“Компилятор генерирует объектный файл, формат которого не зависит от архитектуры компьютера. Скомпилированная программа может выполняться на любых процессорах, а для ее работы требуется лишь исполняющая система Java. Код, генерируемый компилятором Java, называется байт-кодом. Он разработан таким образом, чтобы его можно было легко интерпретировать на любой машине или оперативно преобразовать в собственный машинный код”.

Эта идея не нова. Более тридцати лет назад она была предложена Никлаусом Виртом (Niclaus Wirth) для языка Pascal. Эта же технология была реализована в системе UCSD Pascal.

Очевидно, что байт-код, интерпретируемый с помощью виртуальной машины, всегда будет работать медленнее, чем машинный код, поэтому целесообразность такого подхода у многих вызывает сомнение. Но эффективность байт-кода можно существенно повысить за счет динамической компиляции во время выполнения программы. Этот механизм доказал свою эффективность и даже использовался при создании в корпорации Microsoft платформы .NET, также опирающейся на виртуальную машину.

Следует отметить и ряд других преимуществ виртуальной машины по сравнению с непосредственным выполнением программы. Она существенно повышает безопасность, поскольку в процессе работы можно оценить последствия выполнения каждой конкретной команды. Некоторые программы способны даже генерировать байт-код по ходу выполнения, динамически расширяя свои функциональные возможности.

Переносимость

“В отличие от C и C++, ни один из аспектов спецификации Java не зависит от реализации. Разрядность примитивных типов данных и арифметические операции над ними строго определены”.

Например, тип `int` в Java всегда означает 32-разрядное целое число. А в C и C++ тип `int` может означать как 16-, так и 32-разрядное целое число. Единственное ограничение состоит в том, что разрядность типа `int` не может быть меньше разрядности типа `short int` и больше разрядности типа `long int`. Фиксированная разрядность числовых типов данных позволяет избежать многих неприятностей, связанных с выполнением программ на разных компьютерах. Двоичные данные хранятся и передаются в неизменном формате, что также позволяет избежать недоразумений, связанных с разным порядком следования байтов на различных платформах. Строки сохраняются в стандартном формате Unicode.

“Библиотеки, являющиеся частью системы, предоставляют переносимые интерфейсы. Например, в Java предусмотрен абстрактный класс `Window` и его реализации для операционных систем `Unix`, `Windows` и `Macintosh`”.

Всякий, когда-либо пытавшийся написать программу, которая одинаково хорошо работала бы под управлением операционных систем `Windows`, `Macintosh` и десятка разновидностей ОС `Unix`, знает, что это очень трудная задача. Разработчики версии Java 1.0 предприняли героическую попытку решить эту задачу, предоставив простой набор инструментальных средств, приспособляющий обычные элементы пользовательского интерфейса к различным платформам. К сожалению, библиотека, на

которую было затрачено немало труда, не позволила достичь приемлемых результатов в разных системах. (В реализациях графических программ на разных платформах нередко проявлялись характерные ошибки.) Но это было лишь началом. Во многих приложениях машинная независимость оказалась намного важнее изысканности графического пользовательского интерфейса. Именно эти приложения выиграли от появления версии Java 1.0. А ныне инструментальный набор для создания графического пользовательского интерфейса (ГПИ) полностью переработан и больше не зависит от интерфейсных средств, используемых на конкретном компьютере. Теперь он выглядит вполне согласованным и, по нашему мнению, намного более привлекательным для пользователя, чем в прежних версиях.

Интерпретируемость

“Интерпретатор Java может выполнять байт-код непосредственно на любой машине, на которую перенесен интерпретатор. А поскольку процесс компоновки носит в большей степени пошаговый и относительно простой характер, то процесс разработки программ может быть заметно ускорен, став более творческим”.

Пошаговая компоновка имеет свои преимущества, но ее выгоды для процесса разработки, очевидно, переоцениваются. Первоначально инструментальные средства разработки на Java были довольно медленными. А ныне байт-код транслируется в машинный код динамическим компилятором.

Производительность

“Обычно интерпретируемый байт-код имеет достаточную производительность, но бывают ситуации, когда требуется еще более высокая производительность. Байт-код можно транслировать во время выполнения программы в машинный код для того процессора, на котором выполняется данное приложение”.

На ранней стадии развития Java многие пользователи были не согласны с утверждением, что производительности “более чем достаточно”. Но теперь динамические компиляторы (называемые иначе *JIT-компиляторами*) настолько усовершенствованы, что могут конкурировать с традиционными компиляторами, а в некоторых случаях они даже дают выигрыш в производительности, поскольку имеют больше доступной информации. Так, например, динамический компилятор может отслеживать код, который выполняется чаще, и оптимизировать по быстрдействию только эту часть кода. Динамическому компилятору известно, какие именно классы были загружены. Он может сначала применить встраивание, когда определенная функция вообще не переопределяется на основании загруженной коллекции классов, а затем отменить, если потребуется, такую оптимизацию.

Многопоточность

“К преимуществам многопоточности относится более высокая интерактивная реакция и поведение программ в реальном масштабе времени”.

Если вы когда-либо пытались организовать многопоточную обработку на каком-нибудь из языков программирования, то будете приятно удивлены тем, насколько это легко сделать на Java. В потоках, организуемых на Java, могут быть использованы преимущества многопроцессорных систем, если операционная система допускает

такую возможность. К сожалению, реализации потоков в большинстве систем сильно отличаются друг от друга, а разработчики Java не предпринимают должных усилий, чтобы достичь независимости от платформы и в этом вопросе. Только код для вызова потоков остается одинаковым для всех машин, а ответственность за их поддержку возлагается на операционную систему или библиотеку потоков. Несмотря на это, именно простота организации многопоточной обработки делает язык Java таким привлекательным для разработки серверного программного обеспечения.

Динамичность

“Во многих отношениях язык Java является более динамичным, чем языки C и C++. Он был разработан таким образом, чтобы легко адаптироваться к постоянно изменяющейся среде. В библиотеки можно свободно включать новые методы и объекты, ни коим образом не затрагивая приложения, пользующиеся библиотеками. В Java совсем не трудно получить информацию о ходе выполнения программы”.

Это очень важно в тех случаях, когда требуется добавить код в уже выполняющуюся программу. Ярким тому примером служит код, загружаемый из Интернета для выполнения браузером. В версии Java 1.0 получить информацию о работающей программе было непросто, но в последних версиях Java программирующий получает полное представление как о структуре, так и о поведении объектов. Это весьма ценно для систем, которые должны анализировать объекты в ходе выполнения программы. К таким системам относятся средства построения графического пользовательского интерфейса, интеллектуальные отладчики, подключаемые компоненты и объектные базы данных.



НА ЗАМЕТКУ! После первых успехов Java корпорация Microsoft выпустила программный продукт под названием J++, включавший в себя язык программирования, виртуальную машину и очень похожий на Java. В настоящее время корпорация Microsoft прекратила поддержку J++ и сосредоточила свои усилия на другом языке, который называется C# и чем-то напоминает Java, хотя в нем используется другая виртуальная машина. Создан даже язык J#, предназначенный для переноса приложений J++ на виртуальную машину, используемую в C#. Языки J++, C# и J# в этой книге не рассматриваются.

Аплеты и Интернет

Первоначальная идея была проста — пользователи загружают байт-коды Java по сети и выполняют их на своих машинах. Программы Java, работающие под управлением веб-браузеров, называются *аплетами*. Для использования аплета требуется веб-браузер, поддерживающий язык Java и способный интерпретировать байт-код. Лицензия на исходные коды языка Java принадлежит компании Oracle, настаивающей на неизменности как самого языка, так и структуры его основных библиотек, и поэтому аплеты Java должны запускаться в любом браузере, который поддерживает Java. Посещая всякий раз веб-страницу, содержащую аplet, вы получаете последнюю версию этой прикладной программы. Но важнее другое: благодаря средствам безопасности виртуальной машины вы избавляетесь от необходимости беспокоиться об угрозах, исходящих от вредоносного кода.

Ввод апплета на веб-странице осуществляется практически так же, как и встраивание изображения. Апплет становится частью страницы, а текст обтекает занимаемое им пространство. Изображение, реализуемое апплетом, становится *активным*. Оно реагирует на действия пользователя, изменяет в зависимости от них свой внешний вид и выполняет обмен данными между компьютером, на котором выполняется апплет, и компьютером, где этот апплет постоянно хранится.

На рис. 1.1 приведен характерный пример динамической веб-страницы, выполняющей сложные вычисления и применяющей апплет для отображения моделей молекул. Чтобы лучше понять структуру молекулы, ее можно повернуть или изменить масштаб изображения, пользуясь мышью. Подобные эффекты нельзя реализовать на статических веб-страницах, тогда как апплеты делают это возможным. (Этот апплет можно найти по адресу <http://jmol.sourceforge.net/>.)

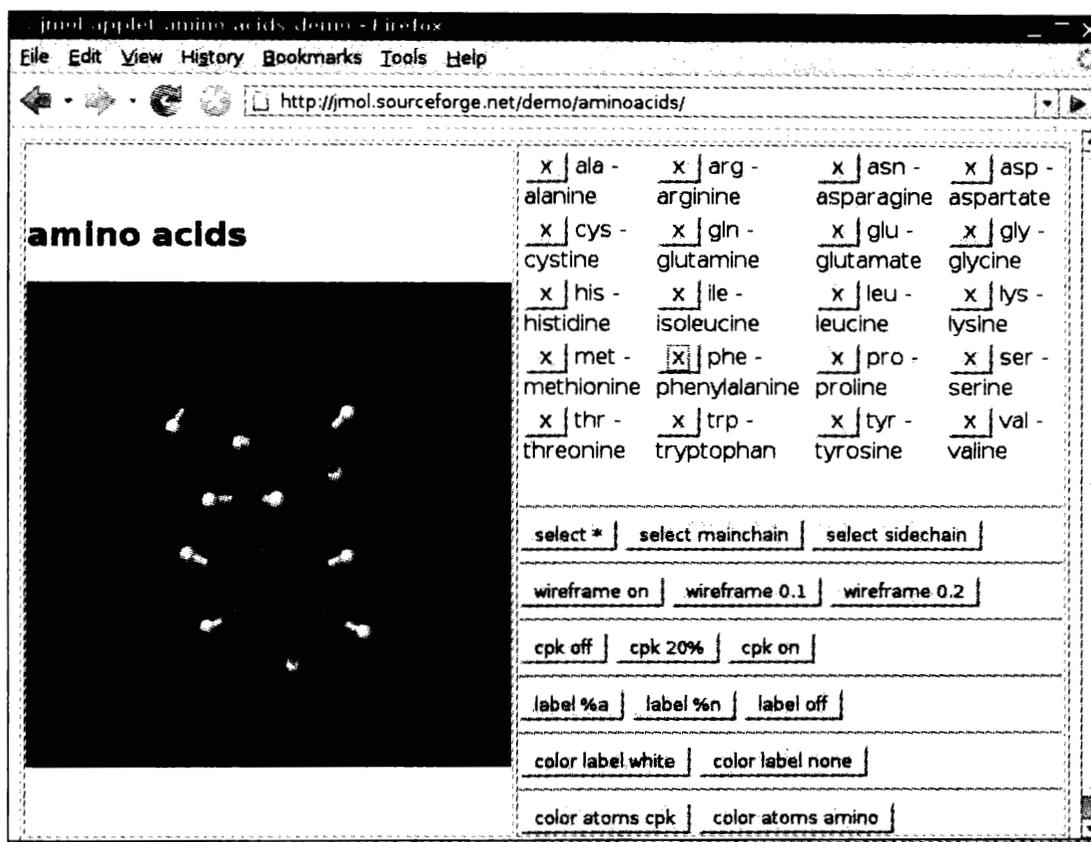


Рис. 1.1. Апплет Jmol

Когда апплеты только появились, они наделали немало шума. Многие считают, что именно привлекательность апплетов стала причиной ошеломляющего успеха Java. Но первоначальный энтузиазм быстро сменился разочарованием. В разных версиях браузеров Netscape и Internet Explorer поддерживались разные версии языка Java, причем некоторые из этих версий заметно устарели. Эта неприятная ситуация создавала все больше препятствий при разработке апплетов, в которых можно было бы воспользоваться преимуществами последней версии Java. Ныне на большинстве веб-страниц вместо апплетов применяются сценарии JavaScript или графические средства Flash, когда требуется получить динамические эффекты в браузере. С другой стороны, Java стал самым распространенным языком для разработки серверных приложений, которые формируют веб-страницы и поддерживают логику работы серверной части.

Краткая история развития Java

В этом разделе кратко изложена история развития языка Java. В основу этого материала положены различные опубликованные первоисточники (в частности, интервью с создателями языка Java в июльском выпуске электронного журнала *SunWorld* за 1995 г.).

История Java восходит к 1991 году, когда группа инженеров из компании Sun Microsystems под руководством Патрика Нотона (Patrick Naughton) и члена совета директоров (и разностороннего специалиста) Джеймса Гослинга (James Gosling) занялась разработкой языка, который можно было бы использовать для программирования бытовых устройств, например, контроллеров для переключения каналов кабельного телевидения. Подобные устройства не обладают большими вычислительными мощностями и объемом оперативной памяти, и поэтому новый язык должен был быть простым и способным генерировать очень компактный код. Кроме того, разные производители могут выбирать разные процессоры для контроллеров, и поэтому было очень важно не привязываться к конкретной их архитектуре. Проект создания нового языка получил кодовое название "Green".

Стремясь получить компактный и независимый от платформы код, разработчики нового языка возродили модель, использовавшуюся при реализации первых версий языка Pascal на заре эры персональных компьютеров. Никлаус Вирт, создатель языка Pascal, в свое время разработал переносимый язык, генерирующий промежуточный код для некоей гипотетической машины. Такие машины называются виртуальными, отсюда термин — *виртуальная машина Java (JVM)*. Этот промежуточный код можно выполнять на любой машине, имеющей соответствующий интерпретатор. Инженеры, работавшие над проектом "Green", также использовали виртуальную машину, что решило основную проблему переносимости кода.

Большинство сотрудников компании Sun Microsystems имели опыт работы с операционной системой Unix, поэтому в основу разрабатываемого ими языка был положен язык C++, а не Pascal. В частности, они сделали язык объектно-, а не процедурно-ориентированным. Как сказал Гослинг в своем интервью: "Язык — это всегда средство, а не цель". Сначала Гослинг решил назвать его Oak (Дуб). (Возможно потому, что он любил смотреть на дуб, росший прямо под окнами его рабочего кабинета в компании Sun Microsystems.) Затем сотрудники компании узнали, что слово "Oak" уже используется в качестве имени ранее созданного языка программирования, и изменили название на Java. Этот выбор был сделан по наитию.

В 1992 г. в рамках проекта "Green" была выпущена первая продукция под названием "47". Это было устройство интеллектуального дистанционного управления. (Умещааясь в корпусе 6×4×4 дюйма, оно имело мощность рабочей станции SPARC.) К сожалению, ни одна из компаний-производителей электронной техники не заинтересовалась этой разработкой. Затем группа стала заниматься созданием устройства для кабельного телевидения, которое могло бы осуществлять новые виды услуг, например, включать видеосистему по требованию. И снова они не получили ни одного контракта. Примечательно, что одной из компаний, согласившихся все-таки с ними сотрудничать, руководил Джим Кларк (Jim Clark) — основатель компании Netscape, впоследствии сделавшей очень много для развития языка Java.

Весь 1993 год и половину 1994 года продолжались безрезультатные поиски покупателей продукции, разработанной в рамках проекта "Green", получившего новое название — "First Person, Inc.". Патрик Нотон, один из основателей группы, в основном

занимавшийся маркетингом, налетал в общей сложности более 300 тысяч миль, пытаясь продать разработанную технологию. Работа над проектом "First Person, Inc." была прекращена в 1994 г.

Тем временем в рамках Интернета начала развиваться система под названием World Wide Web (Всемирная паутина). Ключевым элементом этой системы является браузер, превращающий гипертекстовые данные в изображение на экране. В 1994 году большинство пользователей применяли некоммерческий веб-браузер Mosaic, разработанный в суперкомпьютерном центре университета штата Иллинойс в 1993 г. Частично этот браузер был написан Марком Андреессеном (Mark Andreessen), подрядившимся работать за 6,85 доллара в час. В то время Марк заканчивал университет, и браузер был его дипломной работой. (Затем он достиг известности и успеха как один из основателей и ведущих специалистов компании Netscape.)

В своем интервью журналу *Sun World* Гослинг сказал, что в середине 1994 г. разработчики нового языка поняли: "Нам нужно создать высококачественный браузер. Такой браузер должен представлять собой приложение, соответствующее технологии "клиент-сервер", в которой жизненно важным является именно то, что мы сделали: архитектурная независимость, выполнение в реальном времени, надежность, безопасность — вопросы, которые были не так уж важны для рабочих станций. И мы создали такой браузер".

Сам браузер был разработан Патриком Нотоном и Джонатаном Пэйном (Johnatan Payne). Позднее он был доработан и получил имя HotJava. Чтобы продемонстрировать все возможности Java, браузер был написан на этом языке. Но разработчики не забывали о таких средствах, которые теперь называются апплетами, наделив свой продукт способностью выполнять код на веб-страницах. Программный продукт, подтверждавший действенность новой технологии, был представлен 23 мая 1995 года на выставке SunWorld '95 и вызвал всеобщий интерес к Java, сохраняющийся и по сей день.

Компания Sun Microsystems выпустила первую версию Java в начале 1996 г. Пользователи быстро поняли, что версия Java 1.0 не подходит для разработки серьезных приложений. Конечно, эту версию можно применять для реализации визуальных эффектов на веб-страницах, например, написать апплет, выводящий на страницу случайно "прыгающий" текст, но версия Java 1.0 была еще сырой. В ней даже отсутствовали средства вывода на печать. Грубо говоря, версия Java 1.0 еще не была готова. В следующей версии, Java 1.1, были устранены наиболее очевидные недостатки, улучшены средства рефлексии и реализована новая модель событий для программирования ГПИ. Но несмотря на это, ее возможности были все еще ограничены.

Выпуск версии Java 1.2 стал основной новостью на конференции JavaOne в 1998 г. В новой версии слабые средства для создания графического пользовательского интерфейса и графических приложений были заменены мощным инструментарием. Это был шаг вперед, к реализации лозунга "Write Once, Run Anywhere" ("Однажды написано — везде выполняется"), выдвинутого при разработке предыдущих версий. В декабре 1998 года через три дня (!) после выхода в свет название новой версии было изменено на громоздкое *Java 2 Standard Edition Software Development Kit Version 1.2* (Стандартная редакция набора инструментальных средств для разработки программного обеспечения на Java 2, версия 1.2).

Кроме Standard Edition, были предложены еще два варианта: Micro Edition ("микроредакция") для портативных устройств, например для мобильных телефонов, и Enterprise Edition (редакция для корпоративных приложений). В этой книге основное внимание уделяется редакции Standard Edition.

Версии 1.3 и 1.4 набора инструментальных средств Standard Edition являются результатами поэтапного усовершенствования первоначально выпущенной версии Java 2. Они обладают новыми возможностями, повышенной производительностью и, разумеется, содержат много меньше ошибок. В процессе развития Java многие взгляды на апплеты и клиентские приложения были пересмотрены. В частности, оказалось, что на Java удобно разрабатывать высококачественные серверные приложения.

В версии 5.0 язык Java подвергся наиболее существенной модификации с момента выпуска версии 1.1. (Первоначально версия 5.0 имела номер 1.5, но на конференции JavaOne в 2004 г. была принята новая нумерация версий.) После многолетних исследований были добавлены обобщенные типы (которые приблизительно соответствуют шаблонам C++), хотя при этом не были выдвинуты требования модификации виртуальной машины. Ряд других языковых элементов, например, циклы в стиле `for each`, автоупаковка и метаданные, были явно “навеяны” языком C#.

Версия 6 (без суффикса .0) была выпущена в конце 2006 г. Опять же сам язык не претерпел существенных изменений, но были внесены усовершенствования, связанные с производительностью, а также произведены расширения библиотек.

По мере того как в центрах обработки данных все чаще стали применяться аппаратные средства широкого потребления вместо специализированных серверов, для компании Sun Microsystems наступили тяжелые времена, и в конечном итоге она была приобретена компанией Oracle в 2009 г. Разработка последующих версий Java приостановилась на долгое время. И только в 2011 году компания Oracle выпустила новую версию Java 7 с простыми усовершенствованиями. А более серьезные изменения было решено отложить до версии Java 8, выпуск которой ожидается в 2013 г.

В табл. 1.1 сведены данные об этапах развития языка и библиотек Java. Как видите, размеры прикладного программного интерфейса API значительно увеличились.

Таблица 1.1. Этапы развития языка Java

Версия	Год выпуска	Новые языковые средства	Количество классов и интерфейсов
1.0	1996	Выпуск самого языка	211
1.1	1997	Внутренние классы	477
1.2	1998	Отсутствуют	1524
1.3	2000	Отсутствуют	1840
1.4	2002	Утверждения	2723
5.0	2004	Обобщенные классы, цикл в стиле <code>for each</code> , автоупаковка, аргументы переменной длины, метаданные, перечисления, статический импорт	3279
6	2006	Отсутствуют	3793
7	2011	Оператор <code>switch</code> со строковыми метками ветвей, ромбовидный оператор, двоичные литералы, усовершенствованная обработка исключений	4024

Распространенные заблуждения относительно Java

Мы завершим эту главу перечислением некоторых распространенных заблуждений, касающихся языка Java.

Язык Java — это расширение языка HTML.

Java — это язык программирования, а HTML — способ описания структуры веб-страниц. У них нет ничего общего, за исключением дескрипторов HTML-разметки, позволяющих размещать на веб-страницах апплеты, написанные на Java.

Я пользуюсь XML, и поэтому мне не нужен язык Java.

Java — это язык программирования, а XML — средство описания данных. Данные, представленные в формате XML, можно обрабатывать программными средствами, написанными на любом языке программирования, но лишь в прикладном интерфейсе API языка Java содержатся превосходные средства для обработки таких данных. Кроме того, на Java реализованы многие программы независимых производителей, предназначенные для работы с XML-документами. Более подробно этот вопрос обсуждается во втором томе данной книги.

Язык Java легко выучить.

Нет ни одного языка программирования, сопоставимого по функциональным возможностям с Java, который можно было бы легко выучить. Простейшие программы написать несложно, но намного труднее выполнить серьезные задания. Обратите также внимание на то, что в этой книге обсуждению самого языка Java посвящены только четыре главы. А в остальных главах рассматривается работа с библиотеками, содержащими тысячи классов и интерфейсов, а также многие тысячи методов. Правда, вам совсем не обязательно помнить каждый из них, но все же нужно знать достаточно много, чтобы применять Java на практике.

Java со временем станет универсальным языком программирования для всех платформ.

Теоретически это возможно. Именно об этом мечтают все производители программного обеспечения, кроме корпорации Microsoft. Но есть немало приложений, которые отлично работают на настольных компьютерах, и вряд ли они будут так же надежно работать на других устройствах или в окне браузера. Кроме того, эти приложения написаны таким образом, чтобы в максимальной степени использовать возможности процессоров и платформенно-ориентированных библиотек. В любом случае они уже перенесены на все основные платформы. К таким приложениям относятся текстовые и графические редакторы, а также веб-браузеры. Как правило, эти приложения написаны на C или C++, и пользователь ничего не выиграет, если переписать их на Java.

Java — это всего лишь очередной язык программирования.

Java — прекрасный язык программирования. Многие программисты отдадут предпочтение именно ему, а не языку C, C++ или C#. Но в мире существуют сотни великолепных языков программирования, так и не получивших широкого распространения, в то время как языки с очевидными недостатками, как, например C++ и Visual Basic, достигли поразительных успехов.

Почему так происходит? Успех любого языка программирования определяется в большей степени его *системой поддержки*, чем изяществом его синтаксиса. Существуют ли стандартные библиотеки, позволяющие сделать именно то, что требуется программисту? Разработана ли удобная среда для создания и отладки программ? Интегрирован ли язык и его инструментарий в остальную инфраструктуру компьютера? Язык Java достиг успехов в области серверных приложений, поскольку

его библиотеки классов позволяют легко сделать то, что раньше было трудно реализовать, например, поддерживать работу в сети и организовать многопоточную обработку. Тот факт, что язык Java способствовал сокращению количества ошибок, связанных с указателями, также говорит в его пользу. Благодаря этому производительность труда программистов повысилась. Но не в этом кроется причина его успеха.

С появлением C# язык Java устарел.

Создатели языка C# использовали многие удачные технические решения, применявшиеся в Java, в том числе язык с ясно определенным синтаксисом, виртуальную машину и систему “сборки мусора”. Но по ряду характеристик C# уступает Java, включая вопросы безопасности и переносимости программ. Самое очевидное преимущество C# — это прекрасная среда разработки. Если вы привязаны к ОС Windows, в таком случае вам, вероятно, стоит пользоваться C#. Если же оценивать реальное положение дел в программировании, то на сегодняшний день Java по-прежнему отдают предпочтение очень многие программисты.

Java является патентованным средством, и поэтому пользоваться им не рекомендуется.

Сразу же после создания Java компания Sun Microsystems предоставляла бесплатные лицензии распространителям и конечным пользователям. Несмотря на то что компания Sun Microsystems полностью контролировала распространение Java, в работу над этим языком и его библиотеками были вовлечены многие другие компании. Исходный код виртуальной машины и библиотек доступен всем желающим, но его можно использовать лишь для ознакомления, а вносить в него изменения запрещено. До настоящего момента язык Java имел “закрытый исходный код, но прекрасно работал”.

Ситуация изменилась кардинально в 2007 году, когда компания Sun Microsystems объявила, что последующие версии Java будут доступны на условиях General Public License (GPL) — той же лицензии открытого кода, по которой распространяется Linux. Компания Oracle проявила свою приверженность к сохранению открытости кода Java, но с одной важной оговоркой — патентованием. Всякий может получить патент на использование и видоизменение кода Java по лицензии GPL, но только на настольных и серверных платформах. Так, если вы желаете использовать Java во встроенных системах, для этой цели вам потребуется другая лицензия, которая, скорее всего, повлечет за собой определенные отчисления. Впрочем, срок действия таких патентов истекает через десять лет, после чего язык Java станет совершенно бесплатным.

Программы на Java работают под управлением интерпретатора, а следовательно, серьезные приложения будут выполняться слишком медленно.

В начале развития Java программы на этом языке действительно интерпретировались. Теперь в состав всех виртуальных машин, за исключением “микроредакции” для таких устройств, как мобильные телефоны, входит динамический компилятор. Основные элементы кода будут выполняться не медленнее, чем если бы они были написаны на C++, а в некоторых случаях даже быстрее.

Для Java характерны дополнительные накладные расходы по сравнению с языком C. Запуск виртуальной машины требует времени, и поэтому ГПИ, написанный на Java, действует медленнее, чем аналогичный интерфейс в собственном коде, поскольку он воспроизводится платформенно-независимым способом.

Пользователи уже давно жалуются на низкое быстродействие Java. Но с того времени, когда начались эти жалобы, вычислительные мощности компьютеров многократно возросли. И теперь медленные программы на Java работают быстрее, чем несколько лет назад выполнялись даже самые “молниеносные” программы на C++. Ныне эти жалобы выглядят надуманными, и потому некоторые скептики вместо этого стали жаловаться на “уродливый” пользовательский интерфейс Java.

Все программы на Java выполняются на веб-страницах.

Все апплеты, написанные на Java, действительно выполняются в окне веб-браузера. Но большинство программ на Java являются независимыми приложениями, которые никак не связаны с веб-браузером. Фактически многие программы на Java выполняются на веб-серверах и генерируют код для веб-страниц.

Программы на Java представляют большую опасность для системы защиты.

В свое время было опубликовано несколько отчетов об ошибках в системе защиты Java. Большинство из них касалось реализаций языка Java в конкретных браузерах. Исследователи восприняли это как вызов и принялись искать глобальные недостатки в системе защиты Java, чтобы доказать недееспособность модели безопасности апплетов. Их усилия не увенчались успехом. Обнаруженные ошибки в конкретных реализациях вскоре были исправлены, и, после этого, насколько нам известно, ни одна реальная система не была взломана посредством апплета. Чтобы оценить значение этого факта, вспомните о миллионах вирусных атак на исполняемые файлы операционной системы Windows и макросы редактора Word, действительно вызвавшие немало хлопот. При этом критика недостатков платформы была удивительно беззубой. Кроме того, механизм ActiveX в браузере Internet Explorer мог бы вызвать много нареканий, но способы его взлома настолько очевидны, что лишь немногие специалисты потрудились опубликовать результаты своих исследований.

Некоторые системные администраторы даже стали отключать системы защиты языка Java в своих браузерах, чтобы пользователи могли, как и прежде, загружать исполняемые файлы, элементы управления ActiveX и документы, созданные с помощью текстового процессора Word, что было намного более рискованно. Но даже через 15 лет после своего создания Java по-прежнему остается намного более безопасной платформой, чем любая другая исполняющая платформа из всех имеющихся.

Язык JavaScript — упрощенная версия Java.

JavaScript — это язык сценариев, которые можно использовать на веб-страницах. Он был разработан компанией Netscape и сначала назывался LiveScript. Синтаксис JavaScript напоминает синтаксис Java, но на этом их сходство и заканчивается (за исключением названия, конечно). Подмножество JavaScript было нормировано по стандарту ECMA-262. Сценарии JavaScript более тесно связаны с браузерами, чем апплеты Java. В частности, сценарий JavaScript может видоизменить отображаемый документ, в то время как апплет контролирует только свою ограниченную область отображения.

Пользуясь Java, можно заменить компьютер недорогим устройством для доступа к Интернету.

Когда появился язык Java, некоторые были уверены, что так и случится. И хотя в продаже появились сетевые компьютеры, оснащенные средствами Java, пользователи не спешат заменить свои мощные и удобные персональные компьютеры устрой-

ствами без жестких дисков с ограниченными возможностями. Казалось бы, сетевой компьютер, оснащенный средствами Java, давал вполне реальную возможность реализовать “нулевую администраторскую инициативу”, направленную на уменьшение стоимости компьютеров, применяемых в деловой сфере. Но это направление не получило большого развития. А в современных планшетных компьютерах Java вообще не применяется.

Среда программирования на Java

В этой главе...

- ▶ Установка Java Development Kit
- ▶ Выбор среды для разработки программ
- ▶ Использование инструментов командной строки
- ▶ Применение интегрированной среды разработки
- ▶ Выполнение графического приложения
- ▶ Построение и запуск апплетов

Из этой главы вы узнаете, как устанавливать комплект инструментальных средств разработки Java Development Kit (JDK), а также компилировать и запускать на выполнение разнотипные программы: консольные программы, графические приложения и апплеты. Мы будем пользоваться инструментальными средствами JDK, набирая команды в окне командной оболочки. Но многие программисты предпочитают удобства, предоставляемые интегрированной средой разработки (ИСР). В этой главе будет показано, как пользоваться бесплатно доступной ИСР для компиляции и выполнения программ, написанных на Java. Освоить ИСР и пользоваться ими нетрудно, но они долго загружаются и предъявляют большие требования к вычислительным ресурсам компьютера, так что применять их для разработки небольших программ не имеет смысла. Овладев приемами, рассмотренными в этой главе, и выбрав подходящие инструментальные средства для разработки программ, вы можете перейти к главе 3, с которой, собственно, начинается изучение языка Java.

Установка Java Development Kit

Наиболее полные и современные версии Java Development Kit (JDK) от компании Oracle доступны для операционных систем Solaris, Linux, Mac OS X и Windows. Версии, находящиеся на разных стадиях разработки для многих других платформ, лицензированы и поставляются производителями соответствующих платформ.

Загрузка JDK

Для загрузки Java Development Kit на свой компьютер вам нужно обратиться на веб-страницу по адресу www.oracle.com/technetwork/java/javase/downloads, приложив немного усилий, чтобы разобраться в обозначениях и сокращениях и найти нужное программное обеспечение. И в этом вам помогут сведения, приведенные в табл. 2.1.

Таблица 2.1. Обозначения и сокращения программных средств Java

Наименование	Сокращение	Пояснение
Java Development Kit	JDK	Программное обеспечение для тех, кто желает писать программы на Java
Java Runtime Environment	JRE	Программное обеспечение для потребителей, желающих выполнять программы на Java
Standard Edition	SE	Платформа Java для применения в настольных системах и простых серверных приложениях
Enterprise Edition	EE	Платформа Java для сложных серверных приложений
Micro Edition	ME	Платформа Java для применения в мобильных телефонах и других компактных устройствах
Java 2	J2	Устаревшее обозначение версий Java, выпущенных в 1998–2006 гг.
Software Development Kit	SDK	Устаревшее обозначение версий JDK, выпущенных в 1998–2006 гг.
Update	u	Обозначение, принятое в компании Oracle для выпусков с исправлениями ошибок
NetBeans	—	Интегрированная среда разработки от компании Oracle

Сокращение JDK вам уже знакомо. Оно, как нетрудно догадаться, означает Java Development Kit, т.е. комплект инструментальных средств разработки программ на Java. Некоторые трудности может вызвать тот факт, что в версиях 1.2–1.4 этот пакет называется Java SDK (Software Development Kit). Иногда вы встретите ссылки на старый термин. Существует также среда Java Runtime Environment (JRE), включающая в себя виртуальную машину, но без компилятора. Но вам, как разработчику, она не подходит. Комплект JRE предназначен для конечных пользователей программ на Java, которым компилятор ни к чему. Далее вам будет встречаться обозначение Java SE. Оно означает Java Standard Edition, т.е. стандартную редакцию Java, в отличие от редакций Java EE (Enterprise Edition) для предприятий и Java ME (Micro Edition) для встроенных устройств.

Иногда вам может встретиться обозначение Java 2, которое было введено в 1998 г., когда специалисты компании Sun Microsystems по маркетингу поняли, что очередной дробный номер выпуска никак не отражает глобальных отличий между JDK 1.2 и

предыдущими версиями. Но поскольку такое мнение сформировалось уже после выхода в свет JDK, было решено, что номер версии 1.2 останется за *пакетом разработки*. Последующие выпуски JDK имеют номера 1.3, 1.4 и 5.0. Платформа же была переименована с Java на Java 2. Таким образом, последний комплект разработки назывался Java 2 Standard Edition Software Development Kit Version 5.0, или J2SE SDK 5.0.

Для нас, инженеров, эти манипуляции с обозначениями версий выглядят, по меньшей мере, странно. Именно поэтому мы никогда не сможем достичь успеха в маркетинге. Наконец, в 2006 г. здравый смысл возобладал. Неудобное обозначение Java 2 было отменено, и текущая версия Java Standard Edition была названа Java SE 6. Иногда вам могут попасться случайные ссылки на версии 1.5 и 1.6 — это просто синонимы версий 5.0 и 6.

И, наконец, когда компания Oracle вносит небольшие изменения, исправляя неотложные погрешности, она называет такие изменения обновлениями. Например, первое обновление пакета разработки для версии Java SE 7 официально называется JDK 7u1, где *u* — означает обновление, и имеет внутренний номер версии 1.7.0_01. Обновление совсем не обязательно устанавливать поверх предыдущей версии. Ведь оно содержит самую последнюю версию всего комплекта JDK в целом.

Время от времени компания Oracle выпускает комплекты, содержащие ИСР, помимо JDK. Эта интегрированная среда разработки в разные периоды времени называлась Forte, Sun ONE Studio, Sun Java Studio и Netbeans. Трудно предугадать, какое имя для нее выберут напряженно работающие маркетологи на тот момент, когда вы обратитесь на веб-сайт компании Oracle. На данном этапе вам нужен только комплект JDK. Если впоследствии вы решите поработать с ИСР, то сможете загрузить ее по адресу <http://netbeans.org>.



НА ЗАМЕТКУ! Процедура установки предлагает по умолчанию имя каталога для установки JDK. В имя каталога входит номер версии, например `jdk1.7.0`. Это может показаться не столь важным, но такой подход упрощает работу с новыми версиями JDK, установленными для тестирования.

Если вы работаете в системе Windows, то рекомендуется не принимать предлагаемый каталог. По умолчанию это каталог `c:\Program Files\jdk1.7.0`. Удалите Program Files из пути для установки данного пакета.

В этой книге делаются ссылки на каталог `jdk`, содержащий комплект JDK. Так, если в тексте указана ссылка `jdk/bin`, она обозначает обращение к каталогу `/usr/local/jdk1.7.0/bin` или `c:\jdk1.7.0\bin`.

Задание пути к исполняемым файлам

После установки JDK вам нужно сделать еще один шаг: добавить имя каталога `jdk/bin` в список путей, по которым операционная система ищет исполняемые файлы. В различных системах это действие выполняется по-разному.

- В ОС Unix (включая Solaris, Mac OS X и Linux) процедура редактирования путей к исполняемым файлам зависит от используемой командной оболочки. Так, если вы пользуетесь оболочкой Bourne Again (в ОС Linux — по умолчанию), добавьте приведенную ниже строку в конце файла `~/.bashrc` или `~/.bash_profile`.

```
export PATH=jdk/bin:$PATH
```

- Под Windows войдите в систему как администратор. Запустите панель управления, выберите пиктограмму System (Система). В Windows XP сразу же откроется диалоговое окно свойств системы. А в Vista и Windows 7 вам нужно будет выбрать ссылку Advanced System Settings (Дополнительные параметры системы), как показано на рис. 2.1. В диалоговом окне свойств системы перейдите на вкладку Advanced (Дополнительно), затем щелкните на кнопке Environment (Переменные среды). Прокручивайте содержимое окна System Variables (Системные переменные) до тех пор, пока не найдете переменную окружения Path. Щелкните на кнопке Edit (Изменить), как показано на рис. 2.2. Введите имя каталога `jdk\bin` в начале списка. Новый элемент списка отделяется от уже существующих точкой с запятой, например:

`jdk\bin; остальное`

- Будьте внимательны, заменяя `jdk` на конкретный путь для установки Java, например `c:\jdk1.7.0_02`. Если вы пренебрегли упомянутой выше рекомендацией удалить имя каталога Program Files из этого пути, непременно заключите весь путь в двойные кавычки: `"c:\Program Files\jdk1.7.0_02\bin"`; остальное.

Сохраните сделанные установки. В любом новом консольном окне заданный путь будет использоваться для поиска исполняемых файлов.

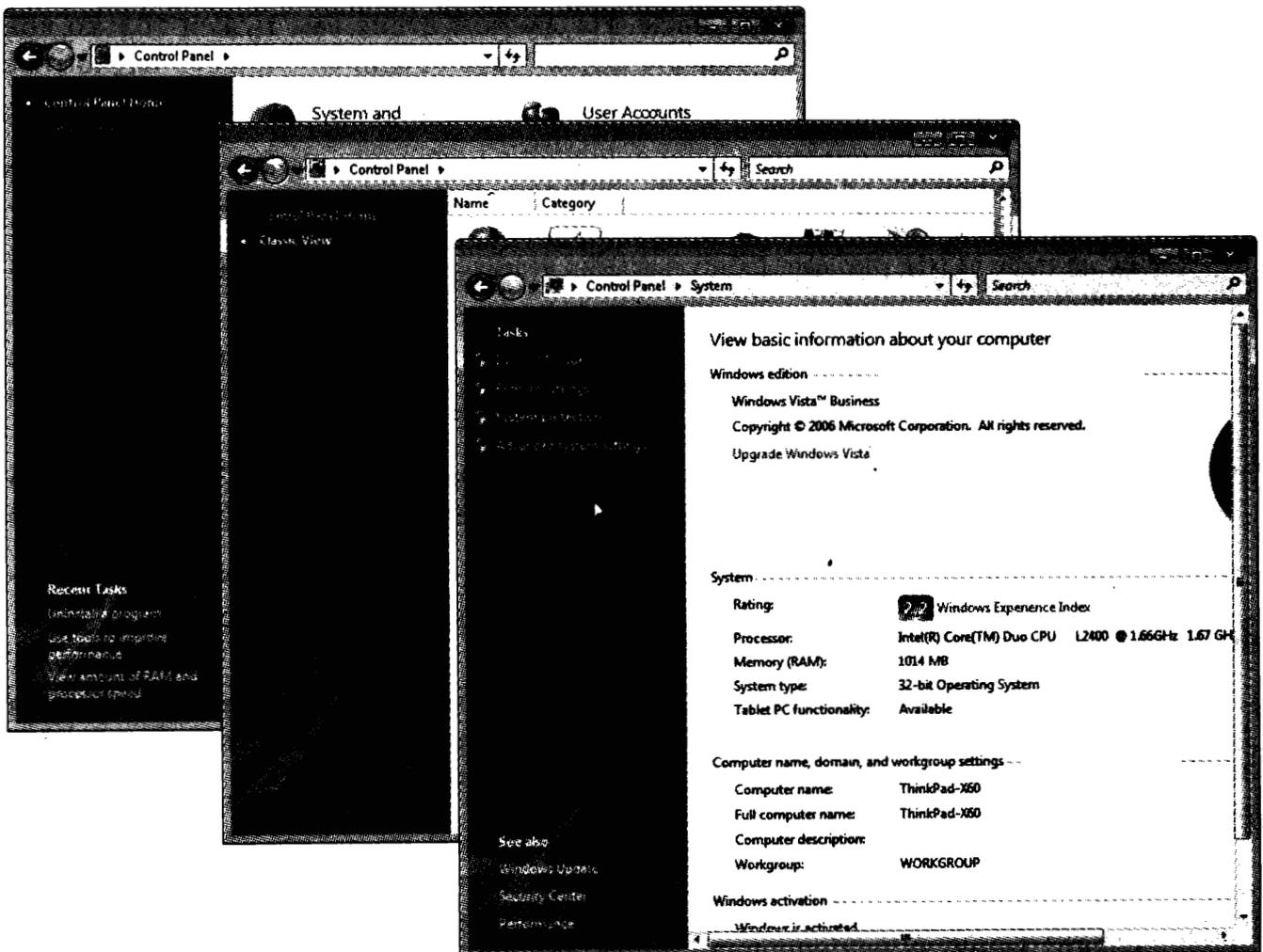


Рис. 2.1. Открытие диалогового окна свойств системы в Windows Vista

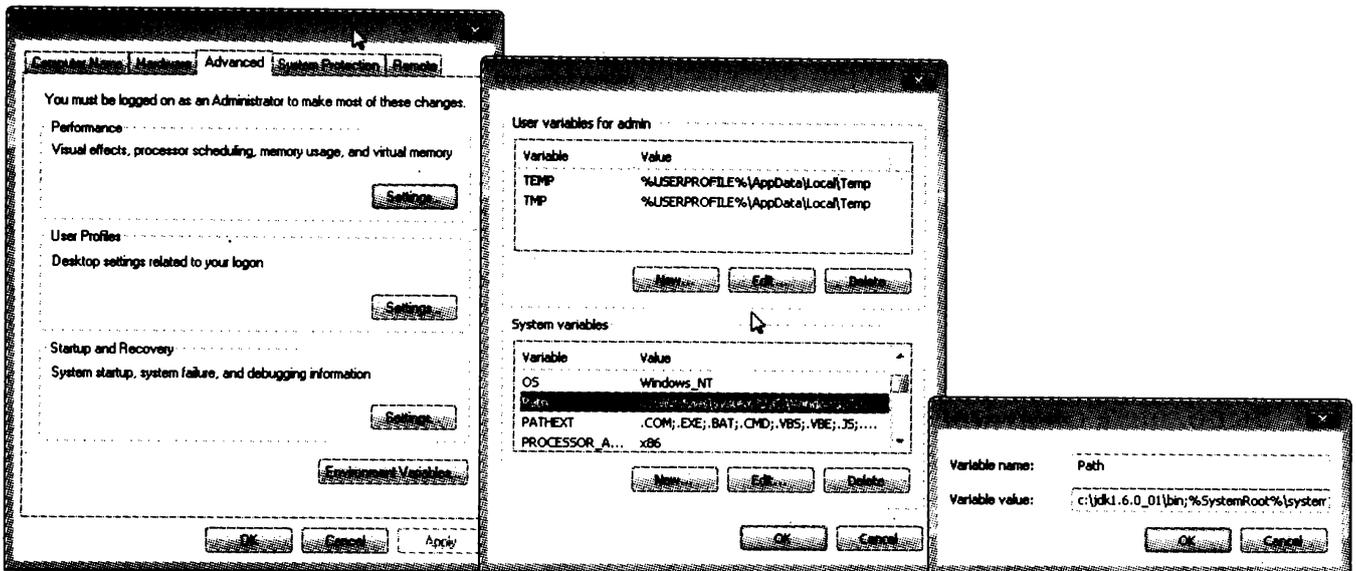


Рис. 2.2. Установка переменной окружения Path в Windows Vista

Правильность установок можно проверить следующим образом. Откройте окно командной оболочки. Как вы это сделаете, зависит от операционной системы. Введите следующую строку:

```
java -version
```

Нажмите клавишу <Enter>. На экране должно появиться следующее сообщение:

```
javac 1.7.0_02
```

Если вместо этого появится сообщение вроде "java:command not found" (java:command не найдено) или "The name specified is not recognized as an internal or external command, operable program or batch file" (Указанное имя не распознано ни как внутренняя или внешняя команда, ни как действующая программа или командный файл), следует еще раз проверить, правильно ли выполнена установка и задан для нее путь.



НА ЗАМЕТКУ! Для открытия окна командной оболочки в Windows выполните следующие действия. Если вы пользуетесь Windows XP, выберите пункт Run (Выполнить) в меню Start (Пуск) и наберите `cmd` в поле Open (Открыть) открывшегося диалогового окна запуска программ. А в Vista и Windows 7 просто введите `cmd` в поле Start Search (Найти программы и файлы) меню Start. Нажмите клавишу <Enter>, и окно командной оболочки немедленно появится.

Если вы никогда не имели дела с командной оболочкой, рекомендуем ознакомиться с руководством, которое научит вас основам работы в режиме командной строки, например, по адресу www.horstmann.com/bigj/help/windows/tutorial.html.

Установка библиотек и документации

Исходные файлы библиотек поставляются в комплекте JDK в виде архива, хранящегося в файле `src.zip`. Вам нужно распаковать этот файл, чтобы получить доступ к исходному коду. Настоятельно рекомендуется сделать это, выполнив следующие действия.

1. Убедитесь в том, что комплект JDK установлен, а имя каталога `jdk/bin` находится в списке путей к исполняемым файлам.
2. Откройте окно командной оболочки.
3. Перейдите к каталогу `jdk` (по команде `cd /usr/local/jdk1.7.0` или `cd C:/jdk1.7.0`).
4. Создайте подкаталог `src`, выполнив следующие команды:

```
mkdir src
cd src
```

5. Выполните следующую команду:

```
jar xvf ../src.jar
```

а если вы работаете в Windows, то команду

```
jar xvf ../src.zip
```



СОВЕТ. Архивный файл `src.zip` содержит исходный код всех общедоступных библиотек. Чтобы получить дополнительный исходный код (компилятора, виртуальной машины, собственных методов и закрытых вспомогательных классов), посетите веб-страницу по адресу <http://jdk7.java.net>.

Документация содержится в отдельном от JDK архиве. Вы можете загрузить ее по адресу www.oracle.com/technetwork/java/javase/downloads. Для этого выполните следующие действия.

1. Убедитесь в том, что пакет JDK установлен правильно, а имя каталога `jdk/bin` находится в списке путей к исполняемым файлам.
2. Загрузите архивный файл с расширением `.zip`, содержащий документацию, на свой компьютер и поместите его в каталог `jdk`. Этот файл называется `jdk-версия-doc.zip`, где вместо слова *версия* указан конкретный номер выпуска, например 7.
3. Откройте окно командной оболочки.
4. Перейдите к каталогу `jdk`.
5. Выполните следующую команду:

```
jar xvf jdk-версия-apidocs.zip
```

где *версия* — номер соответствующей версии.

Установка примеров программ

В процессе работы над материалом этой книги вам, скорее всего, понадобятся приведенные в ней примеры программ. Вы можете найти их на веб-странице по адресу <http://horstmann.com/corejava>. Программы хранятся в архивном файле `corejava.zip`, который нужно распаковать в отдельный каталог (рекомендуется назвать его `CoreJavaBook`). Для этого выполните следующие действия.

1. Убедитесь в том, что комплект JDK установлен правильно, а имя каталога `jdk/bin` находится в списке путей к исполняемым файлам.
2. Создайте каталог `CoreJavaBook`.
3. Загрузите архивный файл `corejava.zip` в этот каталог.
4. Откройте окно командной оболочки.
5. Перейдите к каталогу `CoreJavaBook`.
6. Выполните следующую команду:

```
jar xvf corejava.zip
```

Перемещение по каталогам

При изучении языка Java вам время от времени нужно будет просматривать содержимое исходных файлов с программами на Java. И разумеется, вам придется интенсивно работать с документацией. На рис. 2.3 представлено дерево каталогов пакета JDK.

Структура каталогов	Описание
<code>jdk</code>	(Имя может быть другим, например <code>jdk1.7.0_02</code>)
├── <code>bin</code>	Компилятор и инструментальные средства
├── <code>demo</code>	Демонстрационные программы
├── <code>docs</code>	Библиотечная документация в формате HTML (после распаковки архивного файла <code>j2sdk<версия>-doc.zip</code>)
├── <code>include</code>	Файлы для создания собственных методов (см. второй том)
├── <code>jre</code>	Файлы среды выполнения кода Java
├── <code>lib</code>	Файлы библиотек
└── <code>src</code>	Исходный код библиотек (после распаковки архивного файла <code>src.zip</code>)

Рис. 2.3. Дерево каталогов JDK

Для разработчиков, только приступающих к изучению языка Java, наиболее важными являются каталоги `docs` и `src`. Каталог `docs` содержит документацию на библиотеки Java в формате HTML. Их можно просматривать в окне веб-браузера, например Firefox.



СОВЕТ. Сделайте в окне браузера закладку на локальную страницу с файлом `docs/api/index.html`. При изучении платформы Java вам придется еще не раз обращаться к этой странице.

Каталог `src` содержит исходный код программ, находящихся в общедоступных библиотеках Java. Освоившись с языком, вы можете оказаться в ситуации, когда ни в сведениях, полученных из Интернета, ни в этой книге вы не найдете ответа на интересующий вас вопрос. В таком случае воспользуйтесь исходным кодом. По исходному коду вы всегда сможете понять, что именно делает та или иная библиотечная функция. Так, если вас интересует, как работает класс `System`, загляните в исходный файл `src/java/lang/System.java`.

Выбор среды для разработки программ

Если у вас имеется опыт программирования на Microsoft Visual Studio, значит, вы уже знакомы со средой разработки, состоящей из встроенного текстового редактора, меню для компиляции и запуска программ, а также отладчика. В комплект JDK не входят средства, даже отдаленно напоминающие интегрированную среду разработки (ИСР). Все команды выполняются из командной строки. И хотя такой подход к разработке программ на Java может показаться обременительным, тем не менее мастерское владение им является весьма существенным навыком. Если вы устанавливаете платформу Java впервые, вам придется найти и устранить выявленные неполадки, прежде чем устанавливать ИСР. Но выполняя даже самые элементарные действия самостоятельно, вы получаете лучшее представление о внутреннем механизме работы ИСР.

А после того как вы освоите самые элементарные действия для компиляции и выполнения программ на Java, вам, скорее всего, потребуется ИСР профессионального уровня. За последние десять лет такие ИСР стали настолько эффективны и удобны, что теперь просто нет особого смысла обходиться без них. К числу бесплатных и отличных ИСР относятся Eclipse и NetBeans. В этом разделе будет показано, с чего следует начинать работу с Eclipse, поскольку освоить эту ИСР немного легче, чем NetBeans, хотя и NetBeans постепенно находит все большее распространение. Разумеется, если вы предпочитаете другую ИСР, то вполне можете применять ее для работы с примерами программ из этой книги.

Раньше для написания простых программ мы рекомендовали пользоваться текстовым редактором вроде Emacs, JEdit или TextPad. Но теперь мы воздерживаемся от таких рекомендаций, поскольку ИСР стали намного более быстродействующими и удобными. Вообще говоря, вы должны сначала научиться свободно пользоваться основными инструментальными средствами JDK, и тогда вы почувствуете себя намного увереннее, имея дело с ИСР.

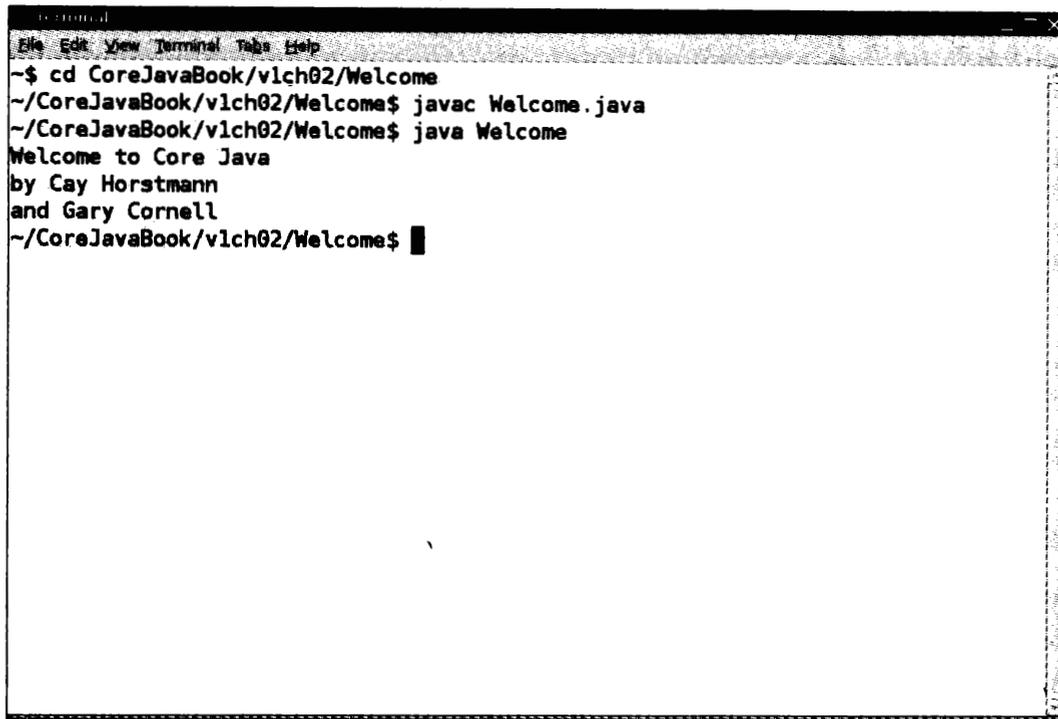
Использование инструментов командной строки

Выберем сначала самый трудный путь, вызывая компилятор и запуская программы на выполнение из командной строки. Для этого выполните следующие действия.

1. Откройте окно командной оболочки.
2. Перейдите к каталогу `CoreJavaBook/v1ch02/Welcome`. (Напомним, что каталог `CoreJavaBook` был специально создан для хранения исходного кода примеров программ из этой книги, как пояснялось ранее в разделе "Установка примеров программ".)
3. Введите следующие команды:

```
javac Welcome.java
java Welcome
```

На экране должен появиться результат, приведенный на рис. 2.4.



```
terminal
File Edit View Terminal Tabs Help
~$ cd CoreJavaBook/vlch02/Welcome
~/CoreJavaBook/vlch02/Welcome$ javac Welcome.java
~/CoreJavaBook/vlch02/Welcome$ java Welcome
Welcome to Core Java
by Cay Horstmann
and Gary Cornell
~/CoreJavaBook/vlch02/Welcome$
```

Рис. 2.4. Компиляция и выполнение программы `Welcome.java`

Примите наши поздравления! Вы только что в первый раз скомпилировали и выполнили программу на Java.

Что же произошло? Служебная программа (иначе — утилита) `javac` — это компилятор Java. Она скомпилировала исходный код из файла `Welcome.java` и преобразовала его в байт-код, сохранив последний в файле `Welcome.class`. А утилита `java` запускает виртуальную машину Java. Она выполняет байт-код, который компилятор поместил в указанный файл с расширением `.class`.



НА ЗАМЕТКУ! Если вы получите сообщение об ошибке в приведенной ниже строке кода, то, вероятнее всего, вы пользуетесь старой версией компилятора Java.

```
for (String g : greeting)
```

А если вы желаете продолжать работу со старой версией компилятора Java, вам придется переписать цикл следующим образом:

```
for (int i = 0; i < greeting.length; i++)
System.out.println(greeting[i]);
```

Программа `Welcome` очень проста и лишь выводит сообщение на экран. Исходный код этой программы приведен в листинге 2.1, а о том, как она работает, речь пойдет в следующей главе.

Листинг 2.1. Исходный код из файла `Welcome.java`

```
1 /**
2  * Эта программа отображает приветствие авторов книги
3  * @version 1.20 2004-02-28
4  * @author Cay Horstmann
5  */
```

```
6 public class Welcome
7 {
8     public static void main(String[] args)
9     {
10         String[] greeting = new String[3];
11         greeting[0] = "Welcome to Core Java";
12         greeting[1] = "by Cay Hortsman";
13         greeting[2] = "and Gary Cornell";
14
15         for (String g : greeting)
16             System.out.println(g);
17     }
18 }
```

Указания по выявлению ошибок

В эпоху визуальных сред разработки программ многие программисты просто не умеют работать в режиме командной строки. Такое неумение чревато неприятными ошибками. Поэтому, работая в режиме командной строки, необходимо принимать во внимание следующее.

- Если вы набираете код программы вручную, внимательно следите за употреблением прописных и строчных букв. Так, в рассмотренном выше примере имя класса должно быть набрано как `Welcome`, а не `welcome` или `WELCOME`.
- Компилятор требует указывать *имя файла* (в данном случае `Welcome.java`). При запуске программы следует указывать *имя класса* (в данном случае `Welcome`) без расширения `.java` или `.class`.
- Если вы получите сообщение "Bad command or file name" (Неверная команда или имя файла) или упоминавшееся ранее сообщение "javac:command not found", проверьте, правильно ли выполнена установка Java и верно ли указаны пути к исполняемым файлам.
- Если компилятор `javac` выдаст сообщение "cannot read: Welcome.java" (невозможно прочитать файл `Welcome.java`), следует проверить, имеется ли нужный файл в соответствующем каталоге.

Работая на платформе Unix, проверьте, правильно ли набраны прописные буквы в имени файла `Welcome.java`. А в Windows просматривайте содержимое каталогов по команде `dir`, а не средствами графического интерфейса Проводника по Windows. Некоторые текстовые редакторы (в частности, Notepad) сохраняют текст в файлах с расширением `.txt`. Если вы пользуетесь таким редактором для редактирования содержимого файла `Welcome.java`, он сохранит его в файле `Welcome.java.txt`. По умолчанию Проводник по Windows скрывает расширение `.txt`, поскольку оно предполагается по умолчанию. В этом случае следует переименовать файл, воспользовавшись командой `ren`, или повторно сохранить его, указав имя в кавычках, например "Welcome.java".

- Если при запуске программы вы получаете сообщение об ошибке типа `java.lang.NoClassDefFoundError`, проверьте, правильно ли вы указали имя файла. Если вы получите сообщение касательно имени `welcome`, начинающегося со строчной буквы `w`, еще раз выполните команду `java Welcome`, написав это имя

с прописной буквы `W`. Не забывайте, что в Java учитывается регистр символов. Если же вы получите сообщение по поводу ввода имени `Welcome/java`, значит, вы случайно ввели команду `java Welcome.java`. Повторите команду `java Welcome`.

- Если вы указали имя `Welcome`, а виртуальная машина не в состоянии найти класс с этим именем, то проверьте, не установлена ли каким-нибудь образом переменная окружения `CLASSPATH` в вашей системе. (Эту переменную, как правило, не стоит устанавливать глобально, но некоторые неудачно написанные установщики программного обеспечения в Windows это делают.) Вы можете временно отменить переменную окружения `CLASSPATH` в текущей командной оболочке, введя команду

```
set CLASSPATH=
```

Рассматриваемая здесь программа работает в Windows, а также в Unix/Linux с командной оболочкой `C shell`. Если же в Unix/Linux применяется командная оболочка `Bourne/bash`, то необходимо ввести следующую команду:

```
export CLASSPATH=
```



СОВЕТ. Отличное учебное пособие имеется по адресу <http://docs.oracle.com/javase/tutorial/getStarted/cupojava>. В нем подробно описываются скрытые препятствия, которые нередко встречаются на пути начинающих программировать на Java.

Применение ИСР

В этом разделе поясняется, как скомпилировать программу в ИСР Eclipse. Этот программный продукт распространяется свободно, а загрузить его можно по адресу <http://eclipse.org>. ИСР Eclipse написана на языке Java, но ее переносимость ограничена из-за применения в ней нестандартной библиотеки управления окнами. Тем не менее существуют версии Eclipse для операционных систем Linux, Mac OS X, Solaris и Windows.

Существуют и другие ИСР, но в настоящее время Eclipse распространена наиболее широко. Для того чтобы приступить к работе в этой ИСР, выполните следующие действия.

1. После запуска Eclipse выберите из меню команду `File⇒New Project` (Файл⇒Создать проект).
2. Выберите вариант `Java Project` (Проект Java) в диалоговом окне мастера проектов (рис. 2.5). Здесь и далее показаны моментальные снимки экрана из пользовательского интерфейса версии Eclipse 3.2. В вашей версии Eclipse элементы пользовательского интерфейса могут несколько отличаться.
3. Щелкните на кнопке `Next` (Далее), укажите в качестве имени проекта `Welcome` и введите полный путь к каталогу, содержащему файл `Welcome.java` (рис. 2.6).
4. Щелкните на кнопке-переключателе `Create project from existing source` (Создать проект из существующего источника).
5. Щелкните на кнопке `Finish` (Готово). В итоге будет создан новый проект.

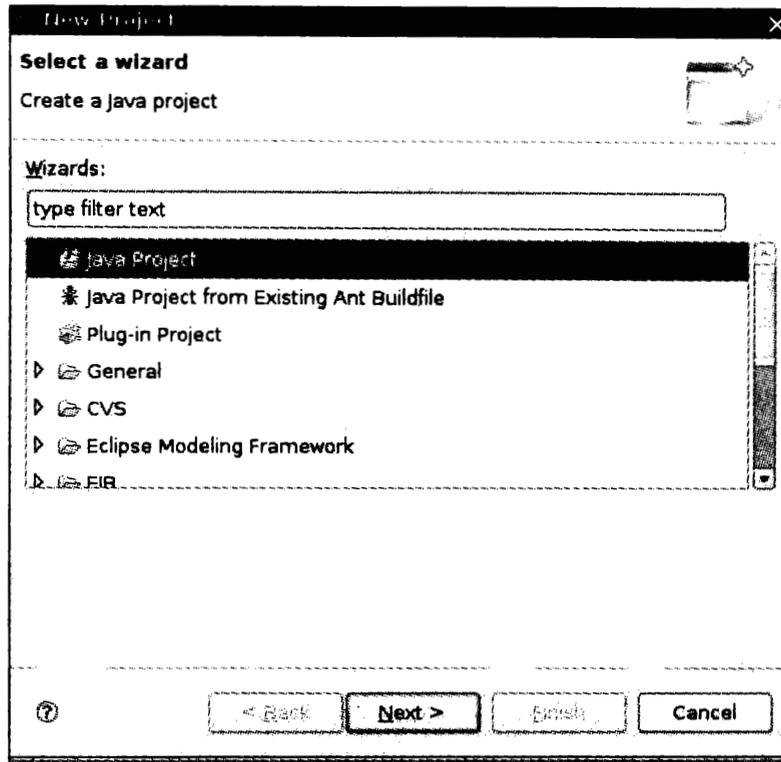


Рис. 2.5. Диалоговое окно Eclipse для создания нового проекта

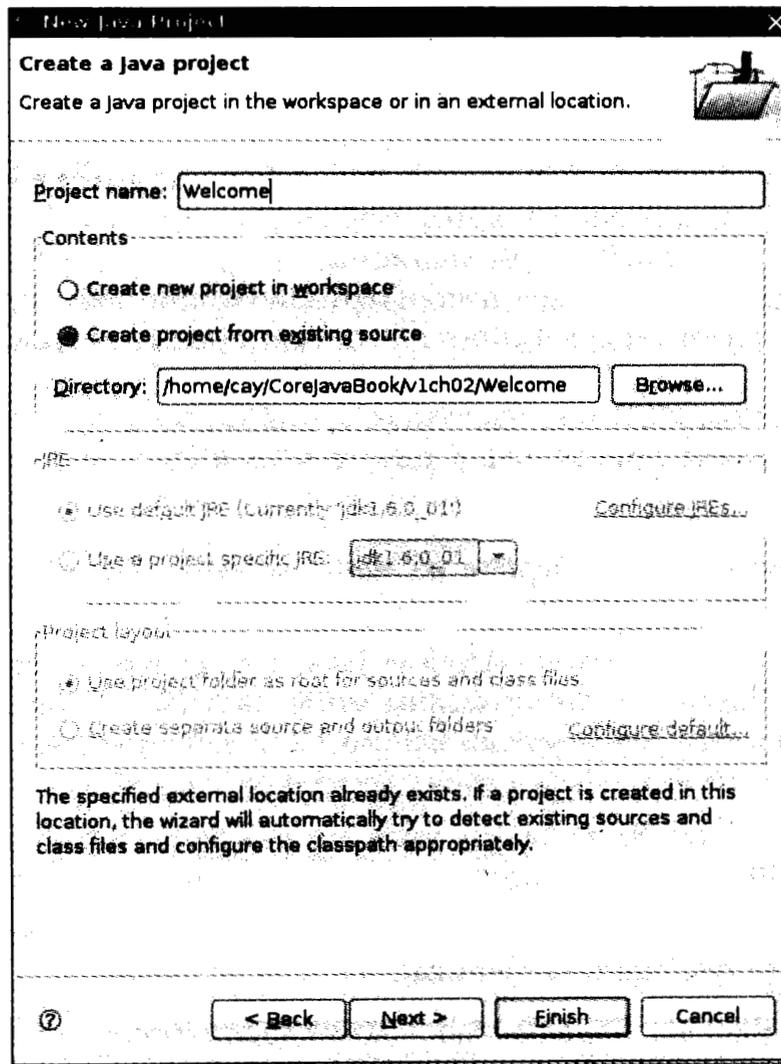


Рис. 2.6. Настройка проекта в Eclipse

- Щелкните на треугольной кнопке рядом с именем нового проекта на левой панели, а затем на треугольной кнопке слева от варианта (default package), т.е. пакет по умолчанию. Дважды щелкните на имени Welcome.java. Появится окно с исходным кодом программы, как показано на рис. 2.7.

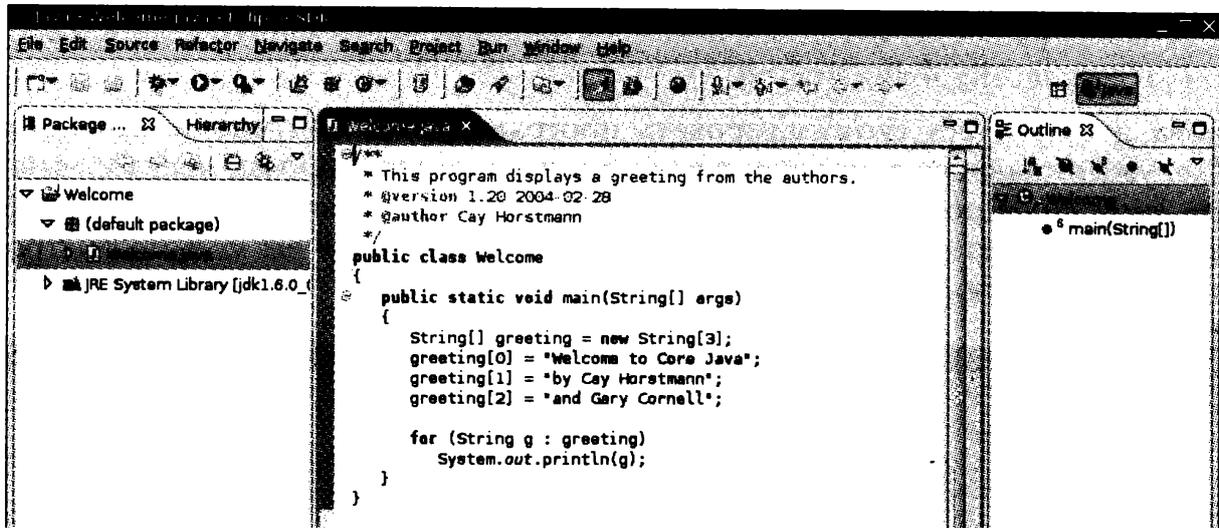


Рис. 2.7. Редактирование исходного кода в Eclipse

- Щелкните правой кнопкой мыши на имени проекта (Welcome) на левой панели. Выберите в открывшемся контекстном меню команду Run ⇒ Run As ⇒ Java Application (Выполнить ⇒ Выполнить как ⇒ Приложение Java). В нижней части окна появится окно для вывода результатов выполнения программы (рис. 2.8).

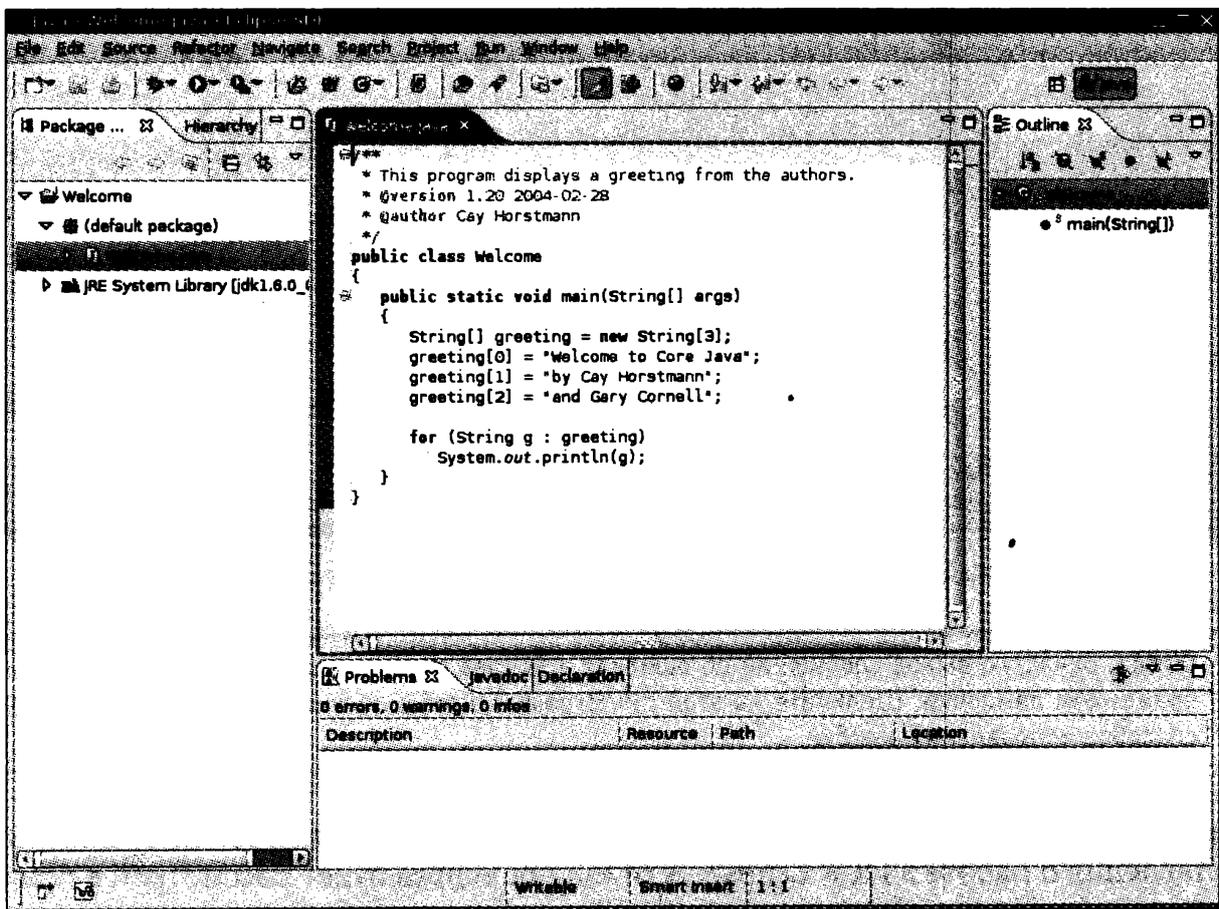


Рис. 2.8. Выполнение программы в Eclipse

Выявление ошибок компиляции

Рассматриваемая здесь программа состоит из нескольких строк кода, и поэтому в ней вряд ли имеются ошибки или даже опечатки. Но для того, чтобы продемонстрировать порядок обработки ошибок, допустим, что в имени String вместо прописной буквы набрана строчная:

```
String[] greeting = new string[3];
```

Снова попытайтесь скомпилировать программу. Вы получите сообщение об ошибке, уведомляющее о том, что в коде программы использован неизвестный тип string (рис. 2.9). Просто щелкните на сообщении. Курсор автоматически перейдет на соответствующую строку кода в окне редактирования, где вы можете быстро исправить допущенную ошибку.

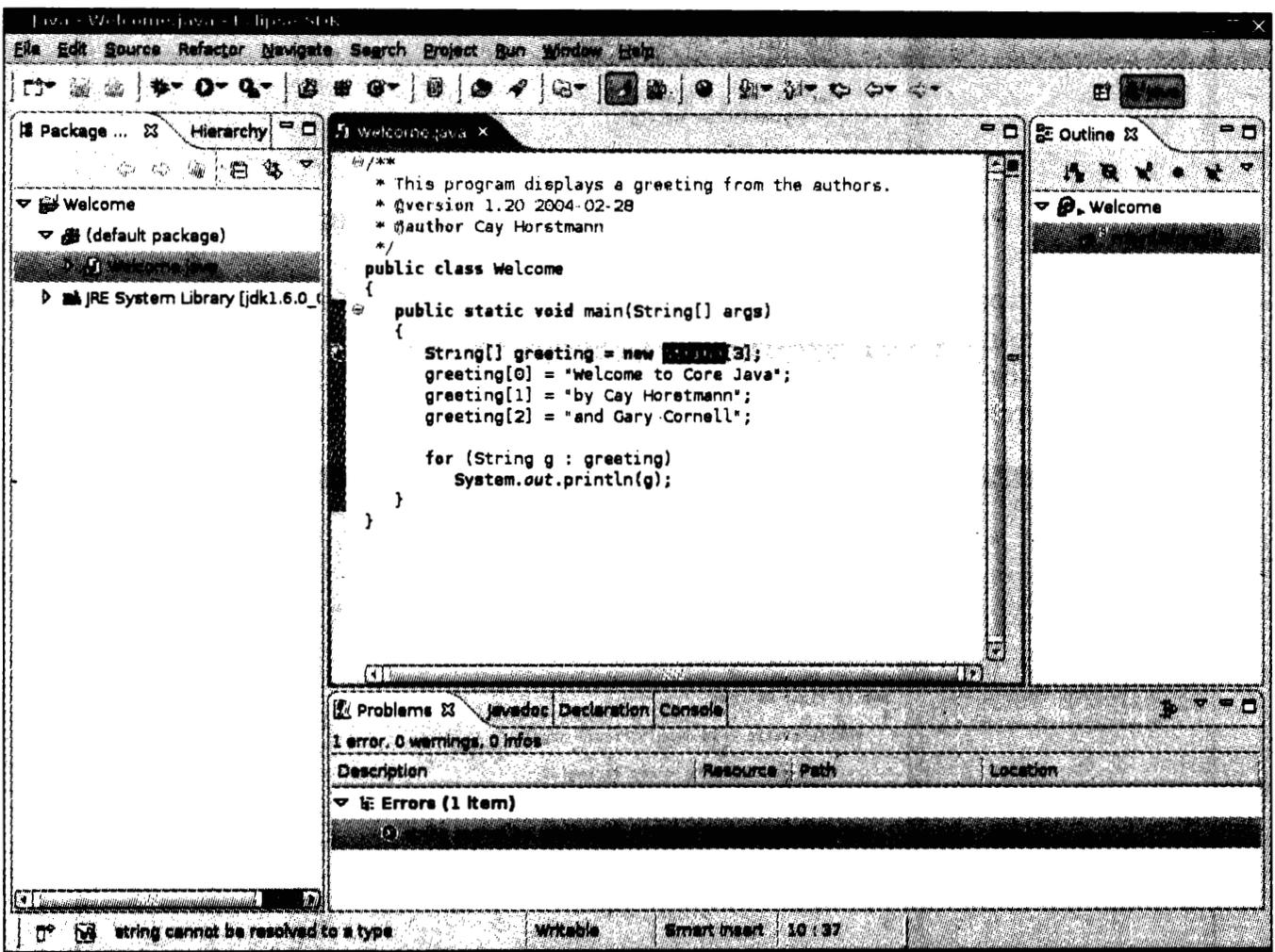


Рис. 2.9. Сообщение об ошибке, выводимое в Eclipse



СОВЕТ. Зачастую вывод сообщений об ошибках в Eclipse сопровождается пиктограммой с изображением лампочки. Щелкните на этой пиктограмме, чтобы получить список рекомендуемых способов исправления ошибки.

Таким образом, на рассмотренном выше простом примере вы получили ясное представление о работе в ИСП Eclipse. А отладчик Eclipse будет рассмотрен в главе 11.

Выполнение графического приложения

Программа `Welcome` не особенно впечатляет. Поэтому перейдем к рассмотрению примера графического приложения. Это приложение представляет собой очень простую программу, которая выводит на экран изображение из файла. Как и прежде, скомпилируем и выполним это приложение в режиме командной строки.

1. Откройте окно командной оболочки.
2. Перейдите к каталогу `CoreJavaBook/v1ch02/ImageViewer`.
3. Введите следующие команды:

```
javac ImageViewer.java  
java ImageViewer
```

На экране появится новое окно приложения `ImageViewer` (рис. 2.10).

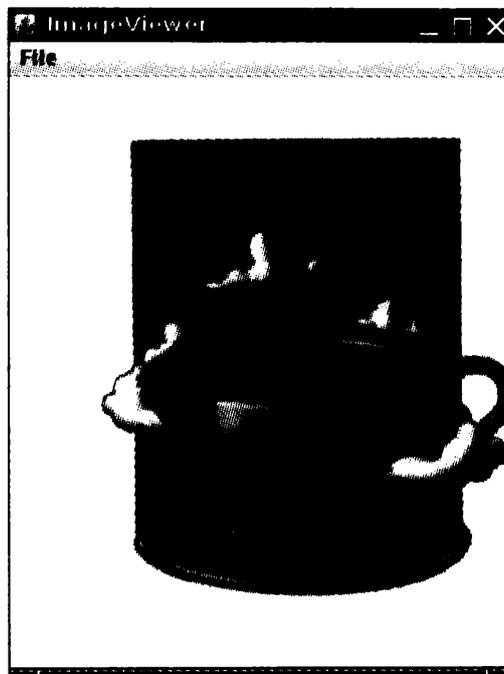


Рис. 2.10. Окно выполняющегося приложения `ImageViewer`

Выберите команду меню `File⇒Open` (Файл⇒Открыть) и найдите файл изображения, который хотите открыть. (В одном каталоге с данной программой находится несколько графических файлов.) Чтобы завершить выполнение программы, щелкните на кнопке `Close` (Закреть) в строке заголовка текущего окна или выберите команду меню `File⇒Exit` (Файл⇒Выйти).

Бегло просмотрите исходный код данной программы. Эта программа заметно длиннее, чем первая, но и она не слишком сложна, особенно если представить себе, сколько строк кода на языке `C` или `C++` нужно было бы написать, чтобы получить такой же результат. Разумеется, такую программу совсем не трудно написать на `Visual Basic`. Для этого достаточно перетащить мышью несколько компонентов и добавить несколько строк, чтобы сделать код работоспособным. В комплект `JDK` не входит визуальный построитель пользовательского интерфейса, поэтому для всех элементов последнего приходится писать соответствующий код, как показано в листинге 2.2. Написанию графических приложений посвящены главы 7–9.

Листинг 2.2. Исходный код из файла ImageViewer/ImageViewer.java

```
1 import java.awt.EventQueue;
2 import java.awt.event.*;
3 import java.io.*;
4 import javax.swing.*;
5
6 /**
7  * Программа для просмотра изображений
8  * @version 1.22 2007-05-21
9  * @author Cay Horstmann
10 */
11 public class ImageViewer
12 {
13     public static void main(String[] args)
14     {
15         EventQueue.invokeLater(new Runnable()
16         {
17             public void run()
18             {
19                 JFrame frame = new ImageViewerFrame();
20                 frame.setTitle("ImageViewer");
21                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22                 frame.setVisible(true);
23             }
24         });
25     }
26 }
27
28 /**
29  * Фрейм с текстовой меткой для вывода изображения.
30 */
31 class ImageViewerFrame extends JFrame
32 {
33     private JLabel label;
34     private JFileChooser chooser;
35     private static final int DEFAULT_WIDTH = 300;
36     private static final int DEFAULT_HEIGHT = 400;
37     public ImageViewerFrame()
38     {
39         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
40
41         // использовать метку для вывода изображений на экран
42         label = new JLabel();
43         add(label);
44
45         // установить селектор файлов
46         chooser = new JFileChooser();
47         chooser.setCurrentDirectory(new File("."));
48
49         // установить строку меню
50         JMenuBar menuBar = new JMenuBar();
51         setJMenuBar(menuBar);
52
53         JMenu menu = new JMenu("File");
54         menuBar.add(menu);
55     }
56 }
```

```
56 JMenuItem openItem = new JMenuItem("Open");
57 menu.add(openItem);
58 openItem.addActionListener(new ActionListener()
59     {
60         public void actionPerformed(ActionEvent event)
61         {
62             // отобразить диалоговое окно селектора файлов
63             int result = chooser.showOpenDialog(null);
64
65             // если файл выбран, задать его в качестве пиктограммы для метки
66             if (result == JFileChooser.APPROVE_OPTION)
67             {
68                 String name = chooser.getSelectedFile().getPath();
69                 label.setIcon(new ImageIcon(name));
70             }
71         }
72     });
73
74 JMenuItem exitItem = new JMenuItem("Exit");
75 menu.add(exitItem);
76 exitItem.addActionListener(new ActionListener()
77     {
78         public void actionPerformed(ActionEvent event)
79         {
80             System.exit(0);
81         }
82     });
83 }
84 }
```

Построение и запуск апплетов

Первые два примера кода, представленные в этой книге, являются *приложениями*, написанными на Java, т.е. независимыми прикладными программами, аналогичными любым другим собственным программам. Но как упоминалось в предыдущей главе, всеобщий интерес к языку Java был вызван в основном его возможностями выполнять апплеты в окне веб-браузера. В этом разделе мы рассмотрим построение и выполнение апплета в режиме командной строки. Затем мы загрузим этот апплет в специальную программу просмотра (утилиту `appletviewer`), входящую в состав JDK. И в завершение отобразим его в окне веб-браузера.

Итак, откройте сначала окно командной оболочки и перейдите к каталогу `CoreJavaBook/v1ch2/WelcomeApplet`, а затем введите следующие команды:

```
javac WelcomeApplet.java
appletviewer WelcomeApplet.html
```

Окно программы просмотра апплетов приведено на рис. 2.11.

Первая команда вам уже знакома — она вызывает компилятор языка Java. В процессе компиляции исходный код апплета из файла `WelcomeApplet.java` преобразуется в байт-код, который помещается в файл `WelcomeApplet.class`.

Но на этот раз вместо утилиты `java` запускается утилита `appletviewer`. Она специально предназначена для быстрой проверки апплетов и входит в состав JDK. Утилите `appletviewer` нужно указать имя файла формата HTML, а не файла класса. Содержимое файла `WelcomeApplet.html` приведено в листинге 2.3.

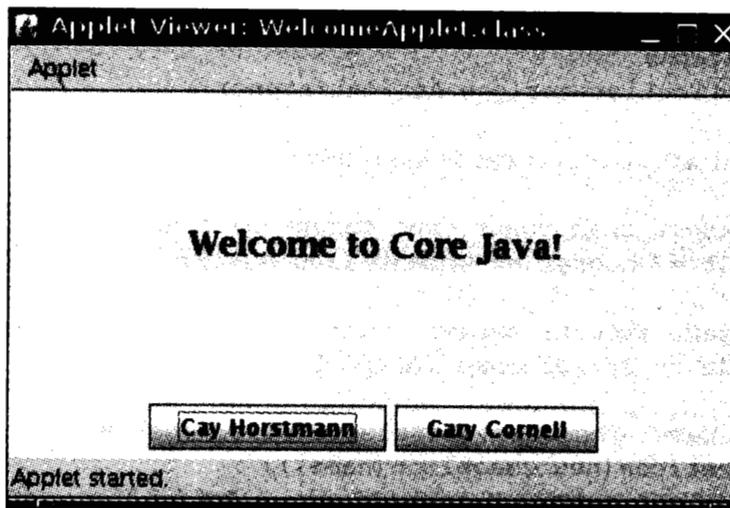


Рис. 2.11. Апплет WelcomeApplet, просматриваемый в окне Applet Viewer

Листинг 2.3. Содержимое файла WelcomeApplet.html

```
1 <html>
2   <head>
3     <title>WelcomeApplet</title>
4   </head>
5   <body>
6     <hr/>
7     <p>
8       This applet is from the book
9       <a href="http://www.horstmann.com/corejava.html">Core Java</a>
10      by <em>Cay Horstmann</em> and <em>Gary Cornell</em>.
11    </p>
12    <applet code="WelcomeApplet.class" width="400" height="200">
13      <param name="greeting" value="Welcome to Core Java!"/>
14    </applet>
15    <hr/>
16    <p><a href="WelcomeApplet.java">The source.</a></p>
17  </body>
18 </html>
```

Если вы знаете язык HTML, то заметите некоторые стандартные конструкции и дескриптор `<applet>`, который указывает утилите `appletviewer`, что необходимо загрузить апплет, код которого содержится в файле `WelcomeApplet.class`. Утилита `appletviewer` игнорирует все дескрипторы HTML, за исключением `<applet>`.

Разумеется, апплеты предназначены для просмотра в браузере. К сожалению, во многих браузерах отсутствует поддержка Java по умолчанию. На веб-странице по адресу http://java.com/ru/download/help/enable_browser.xml поясняется, как настроить наиболее распространенные браузеры на поддержку Java. Настроив соответствующим образом браузер, вы можете затем попытаться загрузить в него апплет. Для этого выполните следующие действия.

1. Запустите избранный браузер.
2. Выберите команду меню `File⇒Open File` (Файл⇒Открыть файл) или откройте файл другим способом.

3. Перейдите к каталогу CoreJavaBook/v1ch2/WelcomeApplet, найдите в нем файл WelcomeApplet.html и загрузите его.
4. Браузер загрузит апплет, включая окружающий текст документа. Веб-страница с апплетом будет выглядеть приблизительно так, как показано на рис. 2.12.

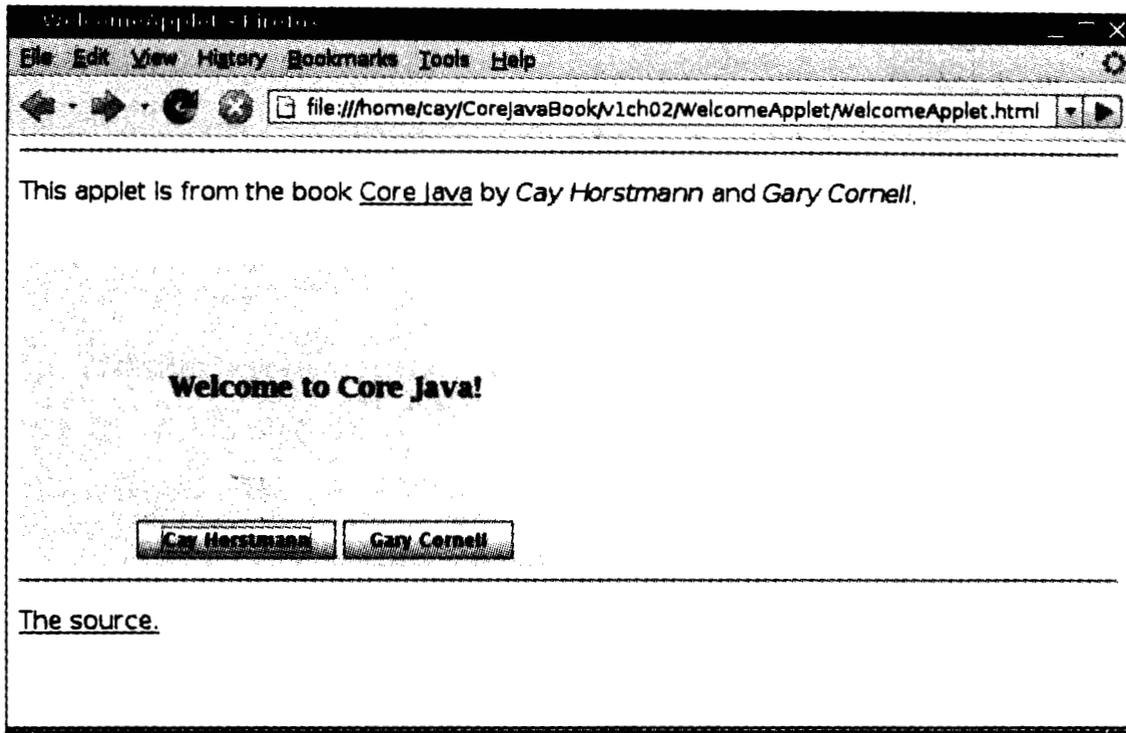


Рис. 2.12. Просмотр апплета WelcomeApplet в окне браузера

Нетрудно заметить, что это приложение действительно активно и готово к взаимодействию с пользователем через Интернет. Щелкните на кнопке Cay Horstmann, и на экране появится страница Кей Хорстманна. Щелкните на кнопке Gary Cornell. Апплет откроет окно для ввода электронной почты, в котором уже указан адрес Гари Корнелла.

Обратите внимание на то, что ни одна из этих кнопок не будет работать с программой просмотра апплетов appletviewer. Эта утилита не может посылать почту или отображать на экране веб-страницу, поэтому она попросту игнорирует соответствующие запросы. Тем не менее она удобна для проверки апплетов в автономном режиме. А для проверки правильности взаимодействия с браузером и пользователем через Интернет апплет необходимо загрузить в окно браузера.



СОВЕТ. Апплет можно также запустить из ИСР. Работая в Eclipse, выберите команду меню Run⇒Run as⇒Java Applet (Выполнить⇒Выполнить как⇒Апплет Java).

Код апплета Welcome приведен в листинге 2.4. В настоящий момент следует лишь бегло просмотреть его. К созданию апплетов мы еще вернемся в главе 10.

Листинг 2.4. Исходный код из файла WelcomeApplet/WelcomeApplet.java

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import java.net.*;
```

```
4 import javax.swing.*;
5 /**
6  * Этот апплет отображает приветствие авторов книги
7  * @version 1.22 2007-04-08
8  * @author Cay Horstmann
9  */
10 public class WelcomeApplet extends JApplet
11 {
12     public void init()
13     {
14         EventQueue.invokeLater(new Runnable()
15         {
16             public void run()
17             {
18                 setLayout(new BorderLayout());
19
20                 JLabel label =
21                     new JLabel(getParameter("greeting"), SwingConstants.CENTER);
22                 label.setFont(new Font("Serif", Font.BOLD, 18));
23                 add(label, BorderLayout.CENTER);
24
25                 JPanel panel = new JPanel();
26
27                 JButton cayButton = new JButton("Cay Horstmann");
28                 cayButton.addActionListener(makeAction("http://www.horstmann.com"));
29                 panel.add(cayButton);
30
31                 JButton garyButton = new JButton("Gary Cornell");
32                 garyButton.addActionListener(makeAction("mailto:gary_cornell@apress.com"));
33                 panel.add(garyButton);
34
35                 add(panel, BorderLayout.SOUTH);
36             }
37         });
38     }
39
40     private ActionListener makeAction(final String urlString)
41     {
42         return new ActionListener()
43         {
44             public void actionPerformed(ActionEvent event)
45             {
46                 try
47                 {
48                     getAppletContext().showDocument(new URL(urlString));
49                 }
50                 catch (MalformedURLException e)
51                 {
52                     e.printStackTrace();
53                 }
54             }
55         };
56     }
57 }
```

Итак, в этой главе были рассмотрены механизмы компиляции и запуска программ, написанных на Java. И теперь вы готовы перейти к главе 3, чтобы приступить непосредственно к изучению языка Java.

Основные языковые конструкции Java

В этой главе...

- ▶ Простая программа на Java
- ▶ Комментарии
- ▶ Типы данных
- ▶ Переменные
- ▶ Операции
- ▶ Символьные строки
- ▶ Ввод и вывод
- ▶ Управляющая логика
- ▶ Большие числа
- ▶ Массивы

Будем считать, что вы успешно установили JDK и смогли запустить простые программы, примеры которых были рассмотрены в главе 2. И теперь самое время приступить непосредственно к программированию. Из этой главы вы узнаете, каким образом в Java реализуются основные понятия программирования, в том числе типы данных, ветви и циклы.

К сожалению, написать программу с ГПИ на Java нелегко, ведь для этого нужно изучить немало вопросов, связанных с окнами, полями ввода, кнопками и прочими элементами. А описание способов построения ГПИ увело бы нас далеко в сторону от главной цели — анализа основных языковых конструкций, поэтому в этой главе

мы рассмотрим лишь самые простые примеры программ, иллюстрирующие то или иное понятие. Во всех этих программах ввод и вывод данных производится в окне командной оболочки.

И наконец, если у вас имеется опыт программирования на C/C++, вы можете бегло просмотреть эту главу, сосредоточив основное внимание на комментариях к C/C++, разбросанных по всему ее тексту. Тем, у кого имеется опыт программирования на других языках, например Visual Basic, многие понятия также будут знакомы, хотя синтаксис рассматриваемых здесь языковых конструкций будет существенно отличаться. Таким читателям рекомендуется тщательно изучить материал этой главы.

Простая программа на Java

Рассмотрим самую простую программу, какую только можно написать на Java. В процессе выполнения она лишь выводит сообщение на консоль.

```
public class FirstSample
{
    public static void main(String[] args)
    {
        System.out.println("We will not use 'Hello, World!'");
    }
}
```

Этому примеру стоит посвятить столько времени, сколько потребуется, чтобы привыкнуть к особенностям языка и рассмотреть характерные черты программ на Java, которые будут еще не раз встречаться во всех приложениях. Прежде всего следует обратить внимание на то, что в языке Java *учитывается регистр символов*. Если вы перепутаете их (например, наберете Main вместо main), программа выполняться не будет.

А теперь проанализируем исходный код данной программы построчно. Ключевое слово `public` называется *модификатором доступа*. Такие модификаторы управляют обращением к коду из других частей программы. Более подробно этот вопрос будет рассматриваться в главе 5. Ключевое слово `class` напоминает нам, что все элементы программ на Java находятся в составе классов. Классы будут подробно обсуждаться в следующей главе, а до тех пор будем считать их некими контейнерами, в которых реализована программная логика, определяющая порядок работы приложения. Классы являются стандартными блоками, из которых состоят все приложения и апплеты, написанные на Java. Все, что имеется в программе на Java, должно находиться внутри класса.

За ключевым словом `class` следует имя класса. Правила составления имен классов не слишком строги. Имя должно начинаться с буквы, а остальная его часть может представлять собой произвольное сочетание букв и цифр. Длина имени не ограничена. В качестве имени класса нельзя использовать зарезервированные слова языка Java (например, `public` или `class`). (Список зарезервированных слов приведен в приложении к этому тому данной книги.)

Согласно принятым условным обозначениям, имя класса должно начинаться с прописной буквы (именно так сформировано имя `FirstSample`). Если имя состоит из нескольких слов, каждое из них должно начинаться с прописной буквы. (Это так называемое *смешанное написание* в "верблюжьем стиле" — `CamelCase`.)

Файл, содержащий исходный текст, должен называться так же, как и открытый (`public`) класс, и иметь расширение `.java`. Таким образом, код рассматриваемого

здесь класса следует поместить в файл `FirstSample.java`. (Как упоминалось выше, регистр символов непременно учитывается, поэтому имя `firstsample.java` для обозначения исходного файла программы не годится.)

Если вы правильно назвали файл и не допустили ошибок в исходном тексте программы, то в результате компиляции получите файл, содержащий байт-код данного класса. Компилятор Java автоматически назовет этот файл `FirstSample.class` и сохранит его в том же каталоге, где находится исходный файл. Осталось лишь запустить программу на выполнение с помощью загрузчика Java, набрав следующую команду:

```
java FirstSample
```

(Расширение `.class` не указывается!) При выполнении этой программы на экран выводится символьная строка "We will not use 'Hello, World!'" (Мы не будем пользоваться избитой фразой «Здравствуй, мир!»).

Когда для запуска скомпилированной программы на выполнение используется следующая команда:

```
java ИмяКласса
```

виртуальная машина Java всегда начинает свою работу с выполнения метода `main()` указанного класса (Термином *метод* в Java принято обозначать функцию.) Следовательно, для нормального выполнения программы в классе должен присутствовать метод `main()`. Разумеется, в класс можно добавить и другие методы и вызывать их из метода `main()`. (Мы покажем, как создавать такие методы, в следующей главе.)



НА ЗАМЕТКУ! В соответствии со спецификацией языка Java (Java Language Specification) метод `main()` должен быть объявлен как `public`. (Спецификация языка Java является официальным документом. Ее можно загрузить по адресу <http://docs.oracle.com/javase/specs>. Некоторые версии загрузчика Java допускали выполнение программ, даже когда метод `main()` не имел модификатора доступа `public`. Эта ошибка была внесена в список замеченных ошибок, доступный по адресу сайта <http://bugs.sun.com/bugdatabase/index.jsp>, и получила номер **4252539**. Но она была помечена меткой "исправлению не подлежит". Разработчики из компании Sun Microsystems разъяснили, что спецификация виртуальной машины Java не требует, чтобы метод `main()` был непременно открытым (см. веб-страницу по адресу <http://docs.oracle.com/javase/specs/jvms/se7/html>), а попытка исправить эту ошибку "может вызвать потенциальные осложнения". К счастью, здравый смысл в конечном итоге возобладал. Загрузчик Java в версии Java SE 1.4 требует, чтобы метод `main()` был открытым (`public`).

Эта история имеет пару интересных аспектов. С одной стороны, становится как-то неуютно от того, что инженеры, занимающиеся контролем качества программ, перегружены работой и не всегда оказываются компетентными в тонких моментах Java, принимая сомнительные решения относительно замеченных ошибок. С другой стороны, стоит отметить тот факт, что в свое время компания Sun Microsystems разместила списки ошибок и способы их исправления на своем веб-сайте, открыв их для всеобщего обозрения. Эта информация весьма полезна для программистов. Вы даже можете проголосовать за вашу "любимую" ошибку. Ошибки, набравшие наибольшее число голосов, будут исправлены в следующих выпусках JDK.

Обратите внимание на фигурные скобки в исходном коде рассматриваемой здесь программы. В Java, так же, как и в C/C++, фигурные скобки используются для выделения *блоков* программы. В Java код любого метода должен начинаться с открывающей фигурной скобки (`{`) и завершаться закрывающей фигурной скобкой (`}`).

Стиль расстановки фигурных скобок всегда вызывал споры. Обычно мы стараемся располагать скобки одну под другой, выравнивая их с помощью пробелов. А поскольку пробелы не имеют значения для компилятора Java, вы можете употреблять какой угодно вам стиль расположения фигурных скобок. В дальнейшем, рассматривая различные операторы цикла, мы обсудим применение фигурных скобок более подробно.

Можете пока что не обращать внимания на ключевые слова `static void`, считая их просто необходимой частью программы на Java. К концу главы 4 вам полностью раскроется подлинный смысл этих ключевых слов. А до тех пор запомните, что каждое приложение на Java должно *непрерывно* содержать метод `main()`, объявление которого приведено ниже.

```
public class ИмяКласса
{
    public static void main(String[] args)
    {
        операторы программы
    }
}
```



НА ЗАМЕТКУ C++! Если вы программируете на C++, то, безусловно, знаете, что такое класс. Классы в Java похожи на классы в C++, но у них имеются существенные отличия. Так, в языке Java все функции являются методами того или иного класса. (Термин *метод* в Java соответствует термину *функция-член* в C++.) Следовательно, в Java должен существовать класс, которому принадлежит метод `main()`. Вероятно, вы знакомы с понятием *статических функций-членов* в C++. Это функции-члены, определенные в классе и не принадлежащие ни одному из объектов этого класса. Метод `main()` в Java всегда является статическим. И наконец, как и в C/C++, ключевое слово `void` означает, что метод не возвращает никакого значения. В отличие от C/C++, метод `main()` не передает операционной системе код завершения. Если этот метод корректно завершает свою работу, код завершения равен 0. А для того чтобы выйти из программы с другим кодом завершения, следует вызвать метод `System.exit()`.

Обратите внимание на следующий фрагмент кода:

```
{
    System.out.println("We will not use 'Hello, World!'");
}
```

Фигурные скобки обозначают начало и конец *тела* метода, которое в данном случае состоит из единственной строки кода. Как и в большинстве языков программирования, операторы Java можно считать равнозначными предложениям в обычном языке. В Java каждый оператор должен *непрерывно* оканчиваться точкой с запятой. В частности, символ конца строки не означает конец оператора, поэтому оператор может занимать столько строк, сколько потребуется.

В рассматриваемом здесь примере кода при выполнении метода `main()` на консоль выводится одна текстовая строка. Для этой цели используется объект `System.out` и вызывается его метод `println()`. Обратите внимание на то, что метод отделяется от объекта точкой. В общем случае вызов метода в Java принимает приведенную ниже синтаксическую форму, что равнозначно вызову функции:

объект.метод(параметры)

В данном примере при вызове метода `println()` в качестве параметра ему передается символьная строка. Этот метод выводит символьную строку на консоль, дополняя ее символом перевода строки. В Java, как и в C/C++, строковый литерал

заключается в двойные кавычки. (Порядок обращения с символьными строками будет рассмотрен далее в этой главе.)

Методам в Java, как и функциям в любом другом языке программирования, можно вообще не передавать параметры или же передавать один или несколько *параметров*, которые иногда еще называют *аргументами*. Даже если у метода отсутствуют параметры, после его имени обязательно ставят пустые скобки. Например, при вызове метода `println()` без параметров на экран выводится пустая строка. Такой вызов выглядит следующим образом:

```
System.out.println();
```



НА ЗАМЕТКУ! У объекта `System.out` имеется метод `print()`, который выводит символьную строку, не добавляя к ней символ перехода на новую строку. Например, при вызове метода `System.out.print("Hello")` выводится строка "Hello" и в конце нее ставится курсор. А следующие выводимые на экран данные появятся сразу после буквы **o**.

Комментарии

Комментарии в Java, как и в большинстве языков программирования, игнорируются при выполнении программы. Таким образом, в программу можно добавить столько комментариев, сколько потребуется, не опасаясь увеличить ее объем. В Java предоставляются три способа выделения комментариев в тексте. Чаще всего для этой цели используются две косые черты (`//`), а комментарий начинается сразу за символами `//` и продолжается до конца строки. Если же требуются комментарии, состоящие из нескольких строк, каждую их строку следует начинать символами `//`, как показано ниже.

```
System.out.println("We will not use 'Hello, World!'");  
// Мы не будем пользоваться избитой фразой "Здравствуй, мир!".  
// Остроумно, не правда ли?
```

Кроме того, для создания больших блоков комментариев можно использовать разделители `/*` и `*/`. И наконец, третьей разновидностью комментариев можно пользоваться для автоматического формирования документации. Эти комментарии начинаются символами `/**` и заканчиваются символами `*/`, как показано в листинге 3.1. Подробнее об этой разновидности комментариев и автоматическом формировании документации речь пойдет в главе 4.

Листинг 3.1. Исходный код из файла `FirstSample/FirstSample.java`

```
1 /**  
2  * Это первый пример программы в главе 3  
3  * @version 1.01 1997-03-22  
4  * @author Gary Cornell  
5  */  
6 public class FirstSample  
7 {  
8     public static void main(String[] args)  
9     {  
10         System.out.println("We will not use 'Hello, World!'");  
11     }  
12 }
```



ВНИМАНИЕ! В Java комментарии, выделяемые символами `/*` и `*/`, не могут быть вложенными. Это означает, что фрагмент кода нельзя исключить из программы, просто закомментировав его парой символов `/*` и `*/`, поскольку в этом коде могут, в свою очередь, присутствовать разделители `/*` и `*/`.

Типы данных

Язык Java является *строго типизированным*. Это означает, что тип каждой переменной должен быть объявлен. В Java имеются восемь основных *простых* или *примитивных* типов данных. Четыре из них представляют целые числа, два — действительные числа с плавающей точкой, один — символы в уникоде (как поясняется далее в разделе “Тип char”), а последний — логические значения.



НА ЗАМЕТКУ! В Java предусмотрен пакет для выполнения арифметических операций с произвольной точностью. Но так называемые “большие числа” в Java являются объектами и не считаются новым типом данных. Далее в этой главе будет показано, как обращаться с ними.

Целочисленные типы данных

Целочисленные типы данных служат для представления как положительных, так и отрицательных чисел без дробной части. В Java имеются четыре целочисленных типа. Все они представлены в табл. 3.1.

Таблица 3.1. Целочисленные типы данных в Java

Тип	Требуемый объем памяти (байты)	Диапазон допустимых значений (включительно)
int	4	От -2147483648 до 2147483647 (т.е. больше 2 млрд.)
short	2	От -32768 до 32767
long	8	От -9223372036854775808 до -9223372036854775807
byte	1	От -128 до 127

Как правило, наиболее удобным оказывается тип `int`. Так, если требуется представить в числовом виде количество обитателей планеты, то нет никакой нужды прибегать к типу `long`. Типы `byte` и `short` используются, главным образом, в специальных приложениях, например, при низкоуровневой обработке файлов или ради экономии памяти при формировании больших массивов, когда во главу угла ставится размер информационного хранилища.

В Java диапазоны допустимых значений целочисленных типов не зависят от машины, на которой выполняется программа. Это существенно упрощает перенос программного обеспечения с одной платформы на другую. Сравните данный подход с принятым в C и C++, где используется наиболее эффективный тип для каждого конкретного процессора. В итоге программа на C, которая отлично работает на 32-разрядном процессоре, может привести к целочисленному переполнению в 16-разрядной системе. Но программы на Java должны одинаково работать на всех машинах, и поэтому диапазоны допустимых значений для различных типов данных фиксированы.

Длинные целые числа указываются с суффиксом **L** (например, **4000000000L**), шестнадцатеричные числа — с префиксом **0x** (например, **0xCAFÉ**), восьмеричные числа — с префиксом **0**. Так, **010** — это десятичное число **8** в восьмеричной форме. Такая запись иногда приводит к недоразумениям, поэтому мы не рекомендуем пользоваться восьмеричными числами.

Начиная с версии Java 7 числа можно указывать и в двоичной форме с префиксом **0b**. Например, **0b1001** — это десятичное число **9** в двоичной форме. Кроме того, числовые литералы можно теперь указывать со знаками подчеркивания, как, например, **1_000_000** (или **0b1111_0100_0010_0100_0000**) для обозначения одного миллиона. Знаки подчеркивания добавляются только ради повышения удобочитаемости больших чисел, а компилятор Java просто удаляет их.

С **НА ЗАМЕТКУ C++!** В C и C++ разрядность таких целочисленных типов, как `int` и `long`, зависит от конкретной платформы. Так, на платформе 8086 с 16-разрядным процессором разрядность целочисленного типа `int` составляет 2 байта, а на таких платформах с 32-разрядным процессором, как Pentium или SPARC, — 4 байта. Аналогично разрядность целочисленного типа `long` на платформах с 32-разрядным процессором составляет 4 байта, а на платформах с 64-разрядным процессором — 8 байт. Именно эти отличия затрудняют написание межплатформенных программ. А в Java разрядность всех числовых типов данных не зависит от конкретной платформы. Следует также заметить, что в Java отсутствуют беззнаковые типы `unsigned`.

Числовые типы данных с плавающей точкой

Типы данных с плавающей точкой представляют числа с дробной частью. В Java имеются два числовых типа данных с плавающей точкой. Они приведены в табл. 3.2.

Таблица 3.2. Числовые типы данных с плавающей точкой в Java

Тип	Требуемый объем памяти (байт)	Диапазон допустимых значений (включительно)
<code>float</code>	4	Приблизительно $\pm 3,40282347E+38F$ (6-7 значащих десятичных цифр)
<code>double</code>	8	Приблизительно $\pm 1,7976931348623157E+308F$ (15 значащих десятичных цифр)

Название `double` означает, что точность таких чисел вдвое превышает точность чисел типа `float`. (Некоторые называют их числами с двойной точностью.) Для большинства приложений тип `double` считается более удобным, а ограниченной точности чисел типа `float` во многих случаях оказывается совершенно недостаточно. Семи значащих (десятичных) цифр, возможно, хватит для того, чтобы точно представить ваш годовой доход в местной валюте, но вряд ли — доход президента вашей компании. Причинами, по которым тип `float` все еще применяется, служат скорость обработки числовых данных (для чисел типа `float` она выше), а также экономия памяти для хранения данных (это важно для больших массивов вещественных чисел).

Числовые значения типа `float` указываются с суффиксом **F**, например **3.14F**. А числовые значения с плавающей точкой, указываемые без суффикса **F** (например, **3.14**), всегда рассматриваются как относящиеся к типу `double`. Для их представления можно (но не обязательно) использовать суффикс **D**, например **3.14D**.



НА ЗАМЕТКУ! Числовые литералы с плавающей точкой могут быть представлены в шестнадцатеричной форме. Например, числовое значение $0,125 = 2^{-3}$ можно записать как `0x1.0p-3`. В шестнадцатеричной форме для выделения показателя степени числа служит обозначение **p**, а не **e**, поскольку **e** — шестнадцатеричная цифра. Обратите внимание на то, что дробная часть числа записывается в шестнадцатеричной форме, а показатель степени — в десятичной, тогда как основание показателя степени — **2**, но не **10**.

Все операции над числами с плавающей точкой производятся по стандарту IEEE 754. В частности, в Java имеются три специальных значения с плавающей точкой:

- `POSITIVE_INFINITY` (положительная бесконечность);
- `NEGATIVE_INFINITY` (отрицательная бесконечность);
- `NaN` (не число).

Например, результат деления положительного числа на **0** равен положительной бесконечности. А вычисление выражения `0/0` или извлечение квадратного корня из отрицательного числа дает нечисловой результат `NaN`.



НА ЗАМЕТКУ! В Java существуют константы `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY` и `Double.NaN` (а также соответствующие константы типа `float`). Но на практике они редко применяются. В частности, для того, чтобы убедиться, что некоторый результат равен константе `Double.NaN`, нельзя выполнить приведенную ниже проверку.

```
if (x == Double.NaN) // никогда не будет истинно
```

Все величины, "не являющиеся числами", считаются разными. Но в то же время можно вызвать метод `Double.isNaN()`, как показано ниже.

```
if (Double.isNaN(x)) // проверить, не является ли числом значение переменной x
```



ВНИМАНИЕ! Числа с плавающей точкой нельзя использовать в финансовых расчетах, где ошибки округления недопустимы. Например, в результате выполнения оператора `System.out.println(2.0 - 1.1)` будет выведено значение `0.8999999999999999`, а не `0.9`, как было бы логично предположить. Подобные ошибки связаны с внутренним двоичным представлением чисел. Как в десятичной системе счисления нельзя точно представить результат деления `1/3`, так и в двоичной системе невозможно точно представить результат деления `1/10`. Если же требуется исключить ошибки округления, то следует воспользоваться классом `BigDecimal`, рассматриваемым далее в главе.

Тип данных `char`

Тип данных `char` служит для представления отдельных символов. Чаще всего это символьные константы. Например, символьной константой является `'A'`, которой соответствует числовое значение `65`. Не следует путать символ `'A'` со строкой `"A"`, состоящей из одного символа. Кодовые единицы уникода (Unicode) могут быть представлены шестнадцатеричными числами в пределах от `\u0000` до `\uFFFF`. Например, значение `\u2122` соответствует символу торговой марки (™), а значение `\u03C0` — греческой букве π .

Кроме префикса `\u`, который предваряет кодовую единицу в уникоде, существует также несколько специальных управляющих последовательностей символов, приведенных табл. 3.3. Эти управляющие последовательности можно вводить в символьные константы или строки, например `'\u2122'` или `"Hello\n"`. Управляющие последовательности, начинающиеся с префикса `\u` (и никакие другие), можно даже указывать за пределами символьных констант или строк, заключаемых в кавычки. Приведенная ниже строка кода вполне допустима, потому что последовательности `\u005B` и `\u005D` соответствуют кодировке символов [и].

```
public static void main(String\u005B\u005D args)
```

Таблица 3.3. Управляющие последовательности специальных символов

Управляющая последовательность	Назначение	Значение в уникоде
<code>\b</code>	Возврат на одну позицию	<code>\u0008</code>
<code>\t</code>	Табуляция	<code>\u0009</code>
<code>\n</code>	Переход на новую строку	<code>\u000a</code>
<code>\r</code>	Возврат каретки	<code>\u000d</code>
<code>\"</code>	Двойная кавычка	<code>\u0022</code>
<code>\'</code>	Одинарная кавычка	<code>\u0027</code>
<code>\\</code>	Обратная косая черта	<code>\u005c</code>

Для того чтобы полностью уяснить тип `char`, нужно иметь ясное представление о принципах кодировки уникода. Кодировка уникода была изобретена для преодоления ограничений традиционных кодировок символов. До появления уникода существовало несколько различных стандартных кодировок: ASCII, ISO 8859-1, KOI-8, GB18030, BIG-5 и т.д. При этом возникали два затруднения. Во-первых, один и тот же код в различных кодировках соответствовал разным символам. Во-вторых, в языках с большим набором символов использовался код различной длины: часто употребляющиеся символы представлялись одним байтом, а остальные символы — двумя, тремя и большим количеством байтов.

Для разрешения этих затруднений была разработана кодировка уникода. В результате исследований, начавшихся в 1980-х годах, выяснилось, что двухбайтового кода более чем достаточно для представления всех символов, использующихся во всех языках мира. И еще оставался достаточный резерв для любых мыслимых расширений. В 1991 году была выпущена спецификация Unicode 1.0, в которой использовалось меньше половины из возможных **65536** кодов. В Java изначально были приняты 16-разрядные символы уникода, что дало ему еще одно преимущество над другими языками программирования, где используются 8-разрядные символы.

Но впоследствии случилось непредвиденное: количество символов превысило допустимый для кодировки предел **65536**. Причиной тому стали чрезвычайно большие наборы иероглифов китайского, японского и корейского языков. Поэтому в настоящее время 16-разрядного типа `char` недостаточно для описания всех символов уникода.

Для того чтобы стало понятнее, каким образом данное затруднение разрешается в Java, начиная с версии Java SE 5.0, необходимо ввести ряд терминов. В частности, *кодовой точкой* называется значение, связанное с символом в кодировке. Согласно

стандарту на уникод, кодовые точки записываются в шестнадцатеричной форме и предваряются символами U+. Например, кодовая точка латинской буквы **A** равна **U+0041**. В уникоде кодовые точки объединяются в 17 *кодowych плоскостей*. Первая кодовая плоскость, называемая *основной многоязыковой плоскостью*, состоит из “классических” символов уникода с кодовыми точками от U+0000 до U+FFFF. Шестнадцать дополнительных плоскостей с кодовыми точками от U+10000 до U+10FFFF содержат *дополнительные символы*.

Кодировка UTF-16 — это способ представления в уникоде всех кодовых точек кодом переменной длины. Символы из основной многоязыковой плоскости представляются 16-битовыми значениями, называемыми *кодowymi единицами*. Дополнительные символы обозначаются последовательными парами кодовых единиц. Каждое из значений кодируемой подобным образом пары попадает в область 2048 неиспользуемых значений из основной многоязыковой плоскости. Эта так называемая *область подстановки* простирается в пределах от U+D800 до U+DBFF для первой кодовой единицы и от U+DC00 до U+DFFF для второй кодовой единицы. Такой подход позволяет сразу определить, соответствует ли значение коду конкретного символа или является частью кода дополнительного символа. Например, математическому коду символа **O**, обозначающего множество октонионов, соответствует кодовая точка U+1D546 и две кодовые единицы — U+D835 и U+DD46 (с описанием алгоритма кодировки UTF-16 можно ознакомиться, обратившись по адресу <http://ru.wikipedia.org/wiki/UTF-16>).

В Java тип `char` описывает *кодovую единицу* в кодировке UTF-16. Начиная программировать на Java рекомендуется пользоваться кодировкой UTF-16 лишь в случае крайней необходимости. Старайтесь чаще пользоваться символьными строками как абстрактными типами данных (подробнее о них речь пойдет ниже, в разделе “Символьные строки”).

Тип данных `boolean`

Тип данных `boolean` имеет два логических значения: `false` и `true`. Они служат для вычисления логических выражений. Преобразование значений типа `boolean` в целочисленные и наоборот невозможно.



НА ЗАМЕТКУ C++! В C++ вместо логических значений можно использовать числа и даже указатели. Так, нулевое значение эквивалентно логическому значению `false`, а ненулевые значения — логическому значению `true`. А в Java представить логические значения посредством других типов нельзя. Следовательно, программирующий на Java защищен от недоразумений, подобных следующему:

```
if (x = 0) // Вместо проверки x == 0 выполнили присваивание x = 0!
```

В C++ эта строка кода компилируется и выполняется проверка по условию, которая всегда дает логическое значение `false`. А в Java наличие такой строки приведет к ошибке на этапе компиляции, поскольку целочисленное выражение `x = 0` нельзя преобразовать в логическое.

Переменные

В языке Java каждая переменная имеет свой *тип*. При объявлении переменной сначала указывается ее тип, а затем имя. Ниже приведен ряд примеров объявления переменных.

```
double salary;
int vacationDays;
long earthPopulation;
char yesChar;
boolean done;
```

Обратите внимание на точку с запятой в конце каждого объявления. Она необходима, поскольку объявление в языке Java считается полным оператором.

Имя переменной должно начинаться с буквы и представлять собой сочетание букв и цифр. Термины *буквы* и *цифры* в Java имеют более широкое значение, чем в большинстве других языков программирования. Буквами считаются символы 'A'-'Z', 'a'-'z', '_' и любой другой символ в кодировке уникада, соответствующий букве. Например, немецкие пользователи в именах переменных могут использовать букву 'ä', а греческие пользователи — букву 'π'. Аналогично цифрами считаются как обычные десятичные цифры, '0'-'9', так и любые символы в кодировке уникада, используемые для обозначения цифры в конкретном языке. Символы вроде '+' или '©', а также пробел нельзя использовать в именах переменных. Все символы в имени переменной важны, причем *регистр* также *учитывается*. Длина имени переменной не ограничивается.



СОВЕТ! Если вам действительно интересно знать, какие именно символы уникада считаются "буквами" в Java, воспользуйтесь для этого методами `isJavaIdentifierStart()` и `isJavaIdentifierPart()` из класса `Character`.



СОВЕТ! Несмотря на то что знак \$ считается действительной буквой в Java, пользоваться им для именования элементов прикладного кода не рекомендуется. Ведь он служит для обозначения имен, формируемых компилятором Java и другими инструментальными средствами.

В качестве имен переменных нельзя использовать зарезервированные слова Java. (Список зарезервированных слов приведен в приложении к этому тому данной книги.) В одной строке программы можно разместить несколько объявлений переменных:

```
int i,j; // обе переменные – целочисленные
```

Но придерживаться такого стиля программирования все же не рекомендуется. Поскольку, если объявить каждую переменную в отдельной строке, читать исходный код программы будет много легче.



НА ЗАМЕТКУ! Как упоминалось ранее, в Java различаются прописные и строчные буквы. Так, переменные `hireday` и `hireDay` считаются разными. Вообще говоря, употреблять в коде две переменные, имена которых отличаются только регистром символов, не рекомендуется. Но иногда для переменной трудно подобрать подходящее имя. Одни программисты в подобных случаях дают переменной имя, совпадающее с именем типа, но отличающееся регистром символов. Например:

```
Box box; // Box – это тип, а box – имя переменной
```

А другие программисты предпочитают использовать в имени переменной префикс `a`:

```
Box aBox;
```

Инициализация переменных

После объявления переменной ее нужно инициализировать с помощью оператора присваивания, поскольку использовать переменную, которой не присвоено никакого значения, нельзя. Например, приведенный ниже фрагмент кода будет признан ошибочным уже на этапе компиляции.

```
int vacationDays;  
System.out.println(vacationDays); // ОШИБКА! Переменная не инициализирована
```

Для того чтобы присвоить ранее объявленной переменной какое-нибудь значение, следует указать слева ее имя, поставить знак равенства (=), а справа записать любое допустимое на языке Java выражение, задающее требуемое значение:

```
int vacationDays;  
vacationDays = 12;
```

При желании переменную можно объявить и инициализировать одновременно. Например:

```
int vacationDays = 12;
```

В Java объявление переменной можно размещать в любом месте кода; Так, приведенный ниже фрагмент вполне допустим.

```
double salary = 65000.0;  
System.out.println(salary);  
int vacationDays = 12; // здесь можно объявить переменную
```

Тем не менее при написании программ на Java переменную рекомендуется объявлять как можно ближе к тому месту кода, где предполагается ее использовать.



НА ЗАМЕТКУ C++! В C и C++ различаются объявление и определение переменной. Ниже приведен пример определения переменной.

```
int i = 10;
```

Объявление переменной выглядит следующим образом:

```
extern int i;
```

А в Java объявления и определения переменных не различаются.

Константы

В Java для обозначения констант служит ключевое слово `final`, как показано в приведенном ниже фрагменте кода.

```
public class Constants  
{  
    public static void main(String[] args)  
    {  
        final double CM_PER_INCH = 2.54;  
        double paperWidth = 8.5;  
        double PaperHeight = 11;  
        System.out.println("Paper size in centimeters: "  
            + paperWidth * CM_PER_INCH + "by" + paperHeight * CM_PER_INCH);  
    }  
}
```

Ключевое слово `final` означает, что присвоить данной переменной какое-нибудь значение можно лишь один раз, после чего изменить его уже нельзя. Использовать в именах констант только прописные буквы необязательно, но такой стиль способствует удобочитаемости кода.

В программах на Java часто возникает потребность в константах, доступных нескольким методам в одном классе. Обычно они называются *константами класса*. Константы класса объявляются с помощью ключевых слов `static final`. Ниже приведен пример применения константы класса в коде.

```
public class Constants2
{
    public static final double CM_PER_INCH = 2.54;
    public static void main(String[] args)
    {
        double paperWidth = 8.5;
        double paperHeight = 11;
        System.out.println("Paper size in centimeters: "
            + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
    }
}
```

Обратите внимание на то, что константа класса задается *за пределами* метода `main()`, поэтому ее можно использовать в других методах того же класса. Более того, если константа объявлена как `public`, методы из других классов также могут получить к ней доступ. В данном примере это можно сделать с помощью выражения `Constants2.CM_PER_INCH`.



НА ЗАМЕТКУ C++! В Java слово `const` является зарезервированным, но в настоящее время оно уже не употребляется. Для объявления констант следует использовать ключевое слово `final`.

Операции

Для обозначения арифметических операций сложения, вычитания, умножения и деления в Java используются обычные знаки подобных операций: `+` `-` `*` и `/` соответственно. Операция `/` обозначает целочисленное деление, если оба ее аргумента являются целыми числами. В противном случае эта операция обозначает деление чисел с плавающей точкой. Остаток от деления целых чисел обозначается символом `%`. Например, `15/2` равно `7`, `15%2` равно `1`, а `15.0/2` — `7.5`.

Заметим, что в результате целочисленного деления на нуль генерируется исключение, в то время как результатом деления на нуль чисел с плавающей точкой является бесконечность или NaN.

В Java предусмотрена сокращенная запись бинарных арифметических операций (т.е. операций, предполагающих два операнда). Например, оператор

```
x += 4;
```

равнозначен оператору

```
x = x + 4;
```

(В сокращенной записи символ арифметической операции, например `*` или `%`, размещается перед знаком равенства, например `*=` или `%=`.)



НА ЗАМЕТКУ! Одной из заявленных целей языка Java является переносимость. Вычисления должны приводить к одинаковому результату, независимо от того, какая виртуальная машина их выполняет. Для арифметических операций над числами с плавающей точкой соблюдение этого требования неожиданно оказалось непростой задачей. Для хранения числовых значений типа `double` используются 64 бита, но в некоторых процессорах применяются 80-разрядные регистры с плавающей точкой. Эти регистры обеспечивают дополнительную точность на промежуточных этапах вычисления. Рассмотрим в качестве примера следующее выражение:

```
double w = x * y / z;
```

Многие процессоры компании Intel вычисляют выражение $x * y$ и сохраняют этот промежуточный результат в 80-разрядном регистре, затем делят его на значение переменной z и, наконец, округляют результат до 64 бит. Подобным образом можно повысить точность вычислений, избежав переполнения. Но этот результат может оказаться иным, если в процессе всех вычислений используется 64-разрядный процессор. По этой причине в первоначальном описании виртуальной машины Java указывалось, что все промежуточные вычисления должны округляться. Это вызвало протест многих специалистов. Округление не только может привести к переполнению. Вычисления при этом происходят медленнее, поскольку операции округления отнимают некоторое время. По этой причине язык Java был усовершенствован таким образом, чтобы распознавать случаи конфликтующих требований для достижения оптимальной производительности и точной воспроизводимости результатов. По умолчанию при промежуточных вычислениях в виртуальной машине может использоваться повышенная точность. Но в методах, помеченных ключевым словом `strictfp`, должны применяться точные операции над числами с плавающей точкой, гарантирующие воспроизводимость результатов. Например, метод `main()` можно записать так:

```
public static strictfp void main(String[] args)
```

В этом случае все команды в теле метода `main()` будут выполнять точные операции над числами с плавающей точкой. А если пометить ключевым словом `strictfp` класс, то во всех его методах должны выполняться точные операции с плавающей точкой.

Многое при подобных вычислениях зависит от особенностей работы процессоров Intel. По умолчанию в промежуточных результатах может использоваться расширенный показатель степени, но не расширенная мантисса. (Процессоры компании Intel поддерживают округление мантиссы без потери производительности.) Следовательно, вычисления по умолчанию отличаются от точных вычислений лишь тем, что в последнем случае возможно переполнение.

Если сказанное выше кажется вам слишком сложным, не отчаивайтесь. Переполнение при вычислениях с плавающей точкой, как правило, не возникает. А в примерах, рассматриваемых в этой книге, ключевое слово `strictfp` использоваться не будет.

Операции инкрементирования и декрементирования

Программисты, конечно, знают, что одной из самых распространенных операций с числовыми переменными является добавление или вычитание единицы. В Java, как и в C и C++, для этой цели предусмотрены операции инкрементирования и декрементирования. Так, в результате вычисления оператора `n++` к текущему значению переменной `n` прибавляется единица, а оператор `n--` уменьшает ее значение на единицу. После выполнения приведенного ниже фрагмента кода значение переменной `n` становится равным 13.

```
int n = 12;  
n++;
```

Операции `++` и `--` изменяют значение переменной, и поэтому их нельзя применять к самим числам. Например, выражение `4++` считается недопустимым.

Существуют два вида операций инкрементирования и декрементирования. Выше продемонстрирована постфиксная форма, в которой символы операции размещаются после операнда. Но есть и префиксная форма: `++n`. Обе эти операции изменяют значение переменной на единицу. Их отличие проявляется только тогда, когда эти операции присутствуют в выражениях. В префиксной форме сначала изменяется значение переменной, и для дальнейших вычислений уже используется новое значение, а в постфиксной форме используется старое значение этой переменной, и лишь после данной операции оно изменяется, как показано в приведенных ниже примерах.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // теперь значение a равно 16, а m равно 8
int b = 2 * n++; // теперь значение b равно 14, а n равно 8
```

Пользоваться операциями инкрементирования и декрементирования в выражениях не рекомендуется, поскольку это зачастую запутывает код и приводит к досадным ошибкам.

(Именно операция `++` дала название языку C++, что и послужило поводом к первой шутке о нем. Недоброжелатели отмечают, что даже имя этого языка содержит в себе ошибку: “Язык следовало бы назвать ++C, потому что мы хотели бы пользоваться им только после его улучшения”.)

Операции отношения и логические операции

В состав Java входит полный набор операций отношения. Для проверки на равенство служат знаки `==`. Например, выражение `3 == 7` дает в результате логическое значение `false`. Для проверки на неравенство служат знаки `!=`. Так, выражение `3 != 7` дает в результате логическое значение `true`. Кроме того, в Java поддерживаются обычные операции сравнения: `<` (меньше), `>` (больше), `<=` (меньше или равно) и `=>` (больше или равно).

В Java, как и в C++, знаки `&&` служат для обозначения логической операции И, а знаки `||` — для обозначения логической операции ИЛИ. Как обычно, знак восклицания (!) означает логическую операцию отрицания. Операции `&&` и `||` задают порядок вычисления по сокращенной схеме: если первый операнд определяет значение всего выражения, то остальные операнды не вычисляются. Рассмотрим для примера два выражения, объединенных логической операцией `&&`.

```
выражение_1 && выражение_2
```

Если первое выражение ложно, то вся конструкция не может быть истинной. Поэтому второе выражение вычислять не имеет смысла. Например, в приведенном ниже выражении вторая часть не вычисляется, если значение переменной `x` равно нулю.

```
x != 0 && 1/x > x+y // не делить на нуль
```

Таким образом, деление на нуль не происходит. Аналогично значение `выражение_1 || выражение_2` оказывается истинным, если истинным является значение первого выражения. В этом случае вычислять второе выражение нет нужды.

В Java имеется также тернарная операция `?:`, которая иногда оказывается полезной. Ниже приведена ее общая форма.

```
условие ? выражение_1 : выражение_2
```

Если *условие* истинно, то вычисляется первое выражение, а если ложно — второе выражение. Например, вычисление выражения $x < y ? x : y$ дает в результате меньшее из значений переменных x и y .

Поразрядные операции

Работая с любыми целочисленными типами данных, можно применять операции, непосредственно обрабатывающие двоичные разряды, или биты, из которых состоят целые числа. Это означает, что для определения состояния отдельных битов числа можно использовать маски. В Java имеются следующие поразрядные операции: $\&$ (И), $|$ (ИЛИ), \wedge (исключающее ИЛИ), \sim (НЕ). Так, если n — целое число, то приведенное ниже выражение будет равно единице только в том случае, если четвертый бит в двоичном представлении числа равен единице.

```
int fourthBitFromRight = (n & 8) / 8;
```

Используя поразрядную операцию $\&$ в сочетании с соответствующей степенью 2, можно замаскировать все биты, кроме одного.



НА ЗАМЕТКУ! При выполнении поразрядной операции $\&$ и $|$ над логическими переменными типа `boolean` получаются логические значения. Эти операции аналогичны логическим операциям $\&\&$ и $||$, за исключением того, что вычисление производится по полной схеме, т.е. обрабатываются все элементы выражения.

В Java поддерживаются также операции \gg и \ll , сдвигающие двоичное представление числа вправо или влево. Эти операции удобны в тех случаях, если требуется сформировать двоичное представление для поразрядного маскирования:

```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```

Имеется даже операция \ggg , заполняющая старшие разряды нулями, в то время как операция \gg восстанавливает в старших разрядах знаковый бит. А такой операции, как \lll , в Java нет.



ВНИМАНИЕ! Значение в правой части операций поразрядного сдвига сокращается по модулю 32 (если левая часть является целочисленным значением типа `long`, то правая часть сокращается по модулю 64). Например, выражение $1 \ll 35$ равнозначно выражению $1 \ll 3$ и дает в итоге значение 8.



НА ЗАМЕТКУ! В C/C++ не определено, какой именно сдвиг выполняет операция \gg : арифметический (при котором знаковый бит восстанавливается) или логический (при котором старшие разряды заполняются нулями). Разработчики вольны выбрать тот вариант, который покажется им наиболее эффективным. Это означает, что результат выполнения операции сдвига вправо в C/C++ определен лишь для неотрицательных чисел. А в Java подобная неоднозначность устранена.

Математические функции и константы

В состав класса `Math` входит целый набор математических функций, которые нередко требуются для решения практических задач. В частности, чтобы извлечь квадратный корень из числа, применяется метод `sqrt()`:

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // выводит числовое значение 2.0
```



НА ЗАМЕТКУ! У методов `println()` и `sqrt()` имеется едва заметное, но существенное отличие. Метод `println()` принадлежит объекту `System.out`, определенному в классе `System`, тогда как метод `sqrt()` — классу `Math`, а не объекту. Такие методы называются статическими. Они будут рассматриваться в главе 4.

В Java не поддерживается операция возведения в степень. Для этого нужно вызывать метод `pow()` из класса `Math`. В результате выполнения приведенной ниже строки кода переменной `y` присваивается значение переменной `x`, возведенное в степень `a`. Оба параметра метода `pow()`, а также возвращаемое им значение относятся к типу `double`.

```
double y = Math.pow(x, a);
```

В состав класса `Math` входят также перечисленные ниже методы для вычисления обычных тригонометрических функций.

```
Math.sin()
Math.cos()
Math.tan()
Math.atan()
Math.atan2()
```

Кроме того, в него включены методы для вычисления экспоненциальной и обратной к ней логарифмической функции (натурального и десятичного логарифмов):

```
Math.exp()
Math.log()
Math.log10()
```

В данном классе определены также две константы — приближенное представление чисел π и e .

```
Math.PI()
Math.E()
```



СОВЕТ. При вызове методов для математических вычислений класс `Math` можно не указывать явно, включив вместо этого в начало файла с исходным кодом следующее выражение:

```
import static java.lang.Math.*;
```

Например, при компиляции приведенной ниже строки кода ошибка не возникает.

```
System.out.println("The square root of \u03C0 is " + sqrt(PI));
```

Более подробно вопросы статического импортирования обсуждаются в главе 4.



НА ЗАМЕТКУ! Для повышения производительности методов из класса `Math` применяются процедуры из аппаратного модуля, выполняющего операции с плавающей точкой. Если для вас важнее не быстрдействие, а предсказуемые результаты, пользуйтесь классом `StrictMath`. В нем реализуются алгоритмы из свободно распространяемой библиотеки `fdlibm` математических функций, гарантирующей идентичность результатов на всех платформах. Исходные коды, реализующие эти алгоритмы, можно найти по адресу <http://www.netlib.org/fdlibm/index.html>. (В библиотеке `fdlibm` каждая функция определена неоднократно, поэтому согласно стандарту IEEE 754 имена функций в классе `StrictMath` начинаются с буквы "e".)

Преобразование числовых типов

Нередко возникает потребность преобразовать один числовой тип в другой. На рис. 3.1 представлены допустимые преобразования.

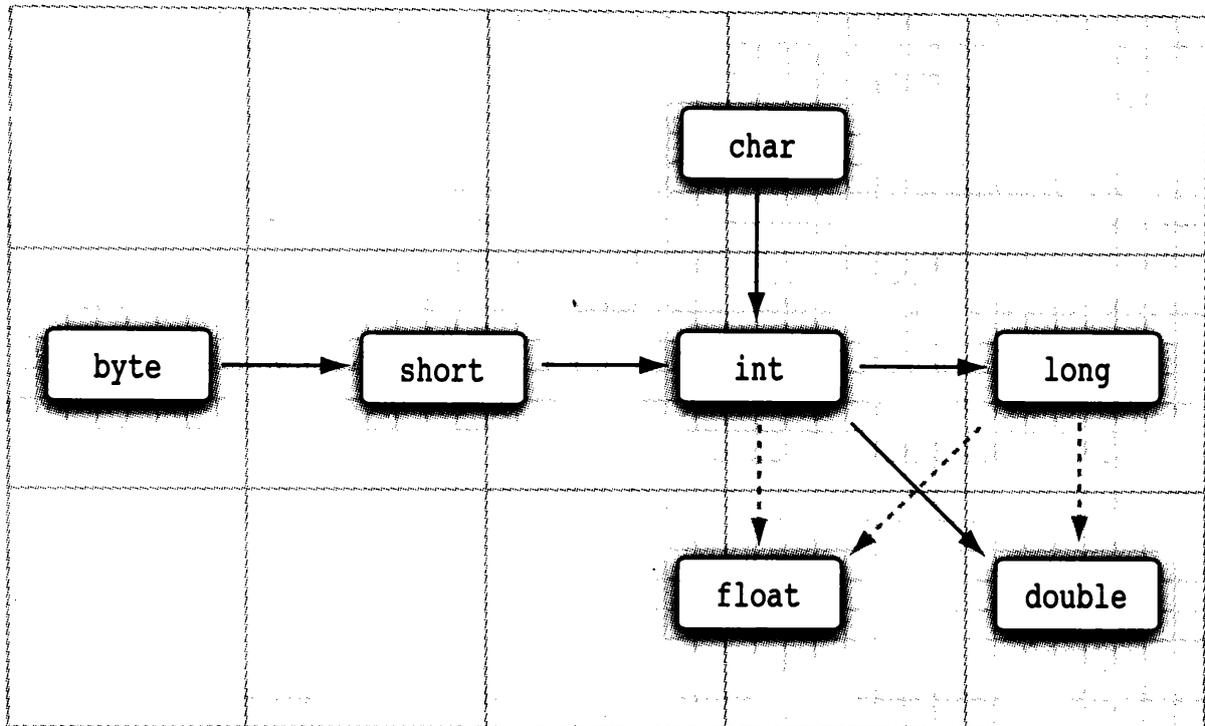


Рис. 3.1. Допустимые преобразования числовых типов

Пять сплошных линий со стрелками на рис. 3.1 обозначают преобразования, которые выполняются без потери данных, а три штриховые линии со стрелками — преобразования, при которых может произойти потеря точности. Например, количество цифр в длинном целом числе **123456789** превышает количество цифр, которое может быть представлено типом **float**. Число, преобразованное в тип **float**, имеет тот же порядок, но несколько меньшую точность:

```
int n = 123456789;
float f = n; // значение переменной f равно 1.234567892E8
```

Если два числовых значения объединяются бинарной операцией (например, $n+f$, где n — целочисленное значение, а f — значение с плавающей точкой), то перед выполнением операции оба операнда преобразуются в числовые значения одинакового типа по следующим правилам.

- Если хотя бы один из операндов относится к типу **double**, то и второй операнд преобразуется в тип **double**.
- Иначе, если хотя бы один из операндов относится к типу **float**, то и второй операнд преобразуется в тип **float**.
- Иначе, если хотя бы один из операндов относится к типу **long**, то и второй операнд преобразуется в тип **long**.
- Иначе оба операнда преобразуются в тип **int**.

Приведение типов

Как пояснялось выше, значения типа `int` автоматически преобразуются, если требуется, в значения типа `double`. С другой стороны, в ряде ситуаций числовое значение типа `double` должно рассматриваться как целое. Преобразования числовых типов в Java возможны, но они могут, конечно, сопровождаться потерей данных. Такие преобразования называются *приведением типов*. Синтаксически приведение типов задается парой скобок, в которых указывается желательный тип, а затем имя переменной:

```
double x = 9.997;
int nx = (int)x;
```

В результате приведения к целому типу числового значения с плавающей точкой в переменной `x` значение переменной `nx` становится равным **9**, поскольку дробная часть числа при этом отбрасывается. Если же требуется округлить число с плавающей точкой до ближайшего целого числа (что во многих случаях намного полезнее), то для этой цели служит метод `Math.round()`, как показано ниже.

```
double x = 9.997;
int nx = (int)Math.round(x);
```

Теперь значение переменной `nx` становится равным **10**. При вызове метода `round()` по-прежнему требуется выполнять приведение типов `(int)`. Дело в том, что значение, возвращаемое методом `round()`, относится к типу `long`, и поэтому оно может быть присвоено переменной типа `int` только с явным приведением. Иначе существует вероятность потери данных.



ВНИМАНИЕ! При попытке приведения типов результат может выйти за пределы диапазона допустимых значений. И в этом случае произойдет усечение. Например, при вычислении выражения `(byte) 300` получается значение `44`.



НА ЗАМЕТКУ C++! Приведение логических значений к целым и наоборот невозможно. Такое ограничение предотвращает появление ошибок. В тех редких случаях, когда действительно требуется представить логическое значение в виде целого, можно составить условное выражение вроде `b ? 1 : 0`.

Скобки и иерархия операций

В табл. 3.4 приведена информация о приоритетности операций. Если скобки не используются, то сначала выполняются более приоритетные операции. Операции, находящиеся на одном уровне иерархии, выполняются слева направо, за исключением операций, имеющих правую ассоциативность, как показано в табл. 3.4. Например, операция `&&` приоритетнее операции `||`, и поэтому выражение `a && b || c` равнозначно выражению `(a && b) || c`. Операция `+=` ассоциируется справа налево, а следовательно, выражение `a += b += c` означает `a += (b += c)`. В данном случае значение выражения `b += c` (т.е. значение переменной `b` после прибавления к нему значения переменной `c`) прибавляется к значению переменной `a`.

Таблица 3.4. Приоритетность операций

Операции	Ассоциативность
[] . () (вызов метода)	Слева направо
! ~ ++ -- + (унарная) - (унарная) () (приведение) new	Справа налево
* / %	Слева направо
+ -	Слева направо
<< >> >>>	Слева направо
< <= > >= instanceof	Слева направо
= !=	Слева направо
&	Слева направо
^	Слева направо
	Слева направо
&&	Слева направо
	Слева направо
?:	Справа налево
= += -= *= /= %= = ^= <<= >>= >>>=	Справа налево



В отличие от C и C++, в Java отсутствует операция-запятая. Но в первой и третьей части оператора цикла `for` можно использовать список выражений, разделенных запятыми.

Перечислимые типы

В некоторых случаях переменной должны присваиваться лишь значения из ограниченного набора. Допустим, вы продаете пиццу четырех размеров: малого, среднего, большого и очень большого. Конечно, вы можете представить размеры целыми числами (1, 2, 3 и 4) или буквами (S, M, L и X). Но такой подход чреват ошибками. В процессе написания программы можно присвоить переменной недопустимое значение, например 0 или m.

В подобных случаях можно воспользоваться *перечислимым типом*. Перечислимый тип имеет конечный набор именованных значений. Например:

```
enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

После этого можно определить переменные данного типа, как показано ниже.

```
Size s = Size.MEDIUM;
```

Переменная типа `Size` может содержать только predefined значения. Допускается также пустое значение `null`, указывающее на то, что в данной переменной не установлено никакого значения. Более подробно перечислимые типы рассматриваются в главе 5.

Символьные строки

По существу, символьная строка Java представляет собой последовательность символов в уникоде. Например, строка "Java\u2122" состоит из пяти символов: J, a, v, a и ™. В Java отсутствует встроенный тип для символьных строк. Вместо этого в стандартной библиотеке Java содержится класс String. Каждая символьная строка, заключенная в кавычки, представляет собой экземпляр класса String:

```
String e = ""; // пустая строка
String greeting = "Hello";
```

Подстроки

С помощью метода `substring()` из класса String можно выделить подстроку из отдельной символьной строки. Например, в результате выполнения приведенного ниже фрагмента кода формируется подстрока "Hel":

```
String greeting = "Hello";
String s = greeting.substring(0, 3);
```

Второй параметр метода `substring()` обозначает позицию символа, который *не* следует включать в состав подстроки. В данном примере требуется скопировать символы на трех позициях 0, 1 и 2 (т.е. от позиции 0 до позиции 2 включительно), поэтому при вызове метода **substring()** указываются значения 0 и 3, обозначающие копируемые символы от позиции 0 и до позиции 2 *включительно*, но *исключая* позицию 3.

Описанный способ вызова метода `substring()` имеет следующую положительную особенность: вычисление длины подстроки осуществляется исключительно просто. Строка `s.substring(a, b)` всегда имеет длину $b - a$ символов. Так, сформированная выше подстрока "Hel" имеет длину $3 - 0 = 3$.

Сцепление

В Java, как и в большинстве языков программирования, предоставляется возможность объединить две символьные строки, используя знак + операции сцепления.

```
String expletive = "Expletive";
String PG13 = "deleted";
String message = expletive + PG13;
```

В приведенном выше фрагменте кода переменной `message` присваивается символьная строка "Expletivedeleted", сцепленная из двух исходных строк. (Обратите внимание на отсутствие пробела между словами в этой строке. Знак + операции сцепления соединяет две строки *точно* в том порядке, в каком они были заданы в качестве операндов.)

При сцеплении символьной строки со значением, не являющимся строковым, это значение преобразуется в строковое. (Как станет ясно из главы 5, каждый объект в Java может быть преобразован в символьную строку.) В приведенном ниже примере переменной `rating` присваивается символьная строка "PG13", полученная путем сцепления символьной строки с числовым значением, автоматически преобразуемым в строковое.

```
int age = 13;
String rating = "PG" + age;
```

Такая возможность широко используется в операторах вывода. Например, приведенная ниже строка кода вполне допустима для вывода результата в нужном формате, т.е. с пробелом между сцепляемыми строками:

```
System.out.println("The answer is " + answer);
```

Принцип постоянства символьных строк

В классе `String` отсутствуют методы, которые позволяли бы *изменять* символы в существующей строке. Так, если требуется заменить символьную строку в переменной `greeting` с "Hello" на "Help!", этого нельзя добиться одной лишь заменой двух последних символов. Программирующим на С это покажется, по меньшей мере, странным. "Как же видоизменить строку?" — спросят они. В Java внести необходимые изменения можно, выполнив сцепление подстроки, которую требуется сохранить, с заменяющими символами, как показано ниже.

```
greeting = greeting.substring(0, 3) + "p!";
```

В итоге переменной `greeting` присваивается символьная строка "Help!".

Программируя на Java, нельзя изменять отдельные символы в строке, поэтому в документации объекты типа `String` называются *неизменяемыми*, т.е. постоянными. Как число 3 всегда равно 3, так и строка "Hello" всегда состоит из последовательности кодовых единиц символов 'H', 'e', 'l', 'l' и 'o'. Изменить ее состав нельзя. Но, как мы только что убедились, можно изменить содержимое строковой переменной `greeting` и заставить ее ссылаться на другую символьную строку подобно тому, как числовой переменной, в которой хранится число 3, можно присвоить число 4.

Не приводит ли это к снижению эффективности кода? Казалось бы, намного проще изменять символы, чем создавать новую строку заново. Возможно, это и так. В самом деле, неэффективно создавать новую строку путем сцепления символьных строк "Hel" и "p!". Но у неизменяемых строк имеется одно существенное преимущество: компилятор может сделать строки *совместно используемыми*.

Чтобы стал понятнее принцип постоянства символьных строк, представьте, что в общем пуле находятся разные символьные строки. Строковые переменные указывают на объекты в этом пуле. При копировании строковой переменной оригинал и копия содержат одну и ту же общую последовательность символов. Логично не прибегать к дублированию строк, а поместить в переменные ссылки на одну и ту же область памяти. Одним словом, создатели языка Java решили, что эффективность совместного использования памяти перевешивает неэффективность редактирования строк путем выделения подстрок и сцепления.

Посмотрите на свои программы. Мы подозреваем, что чаще всего вы не изменяете символьные строки, а сравниваете их. Разумеется, бывают случаи, когда непосредственные манипуляции символьными строками оказываются более эффективными. Одна из таких ситуаций возникает, когда нужно составить строку из отдельных символов, поступающих из файла или с клавиатуры. Для подобных ситуаций в языке Java предусмотрен отдельный класс `StringBuffer`, рассматриваемый далее в разделе "Построение строк", а в остальном достаточно средств, предоставляемых в классе `String`.



НА ЗАМЕТКУ C++! Когда программирующие на С обнаруживают символьные строки в программе на Java, они обычно попадают в тупик, поскольку привыкли рассматривать строки как массивы символов:

```
char greeting[] = "Hello";
```

Это не совсем подходящая аналогия: символьная строка в Java больше напоминает указатель `char*`:

```
char* greeting = "Hello";
```

При замене содержимого переменной `greeting` другой символьной строкой в коде Java выполняется примерно следующее:

```
char* temp = malloc(6);
strncpy(temp, greeting, 4);
strncpy(temp + 4, "!", 2);
greeting = temp;
```

Разумеется, теперь переменная `greeting` указывает на строку "Help!". И даже самые убежденные поклонники языка C должны признать, что синтаксис Java более изящный, чем последовательность вызовов функции `strncpy()`. А что, если присвоить переменной `greeting` еще одно строковое значение, как показано ниже?

```
greeting = "Howdy";
```

Не возникнут ли при этом утечки памяти? К счастью, в Java имеется механизм автоматической "сборки мусора". Если память больше не нужна, она в конечном итоге освобождается.

Если вы программируете на C++ и применяете класс `string`, определенный в стандарте ANSI C++, вам будет намного легче работать с объектами типа `String` в Java. Объекты класса `string` в C++ также обеспечивают автоматическое выделение и освобождение памяти. Управление памятью осуществляется явным образом с помощью конструкторов, деструкторов и операций присваивания. Но символьные строки в C++ являются изменяемыми, а это означает, что отдельные символы в строке можно видоизменять.

Проверка символьных строк на равенство

Чтобы проверить две символьные строки на равенство, достаточно вызвать метод `equals()`. Так, выражение `s.equals(t)` возвращает логическое значение `true`, если символьные строки `s` и `t` равны, а иначе — логическое значение `false`. Следует иметь в виду, что в качестве `s` и `t` могут быть использованы строковые переменные или константы. Например, следующее выражение вполне допустимо:

```
"Hello!".equals(greeting);
```

А для того чтобы проверить идентичность строк, игнорируя отличия в прописных и строчных буквах, следует вызвать метод `equalsIgnoreCase()`, как показано ниже.

```
"Hello".equalsIgnoreCase("hello");
```

Для проверки символьных строк на равенство *нельзя* применять операцию `==`. Она лишь определяет, хранятся ли обе строки в одной и той же области памяти. Разумеется, если обе строки хранятся в одном и том же месте, они должны совпадать. Но вполне возможна ситуация, когда одинаковые символьные строки хранятся в разных местах. Ниже приведен соответствующий пример.

```
String greeting = "Hello"; // инициализировать переменную greeting строкой
if (greeting = "Hello") ...
    // возможно, это условие истинно
if (greeting.substring(0, 3) == "Hel") ...
    // возможно, это условие ложно
```

Если виртуальная машина всегда обеспечивает совместное использование одинаковых символьных строк, то для проверки их равенства можно применять операцию `==`. Но совместно использовать можно лишь константы, а не символьные строки, получающиеся в результате таких операций, как сцепление или извлечение подстроки методом `substring()`. Следовательно, лучше вообще отказаться от проверки символьных строк на равенство с помощью операции `==`, чтобы исключить в программе наихудшую из возможных ошибок, проявляющуюся лишь время от времени и практически не предсказуемую.



НА ЗАМЕТКУ C++! Если вы привыкли пользоваться классом `string` в C++, будьте особенно внимательны при проверке символьных строк на равенство. В классе `string` операция `==` перегружается и позволяет проверять идентичность содержимого символьных строк. Возможно, создатели Java напрасно отказались от обработки символьных строк подобно числовым значениям, но в то же время это позволило сделать символьные строки похожими на указатели. Создатели Java могли бы, конечно, переопределить операцию `==` для символьных строк таким же образом, как они это сделали с операцией `+`. Что ж, у каждого языка программирования свои недостатки.

Программирующие на C вообще не пользуются операцией `==` для проверки строк на равенство, а вместо этого они вызывают функцию `strcmp()`. Метод `compareTo()` является в Java точным аналогом функции `strcmp()`. Конечно, можно воспользоваться выражением следующего вида:

```
if (greeting.compareTo("Help") == 0) ...
```

Но, на наш взгляд, применение метода `equals()` делает программу более удобочитаемой.

Пустые и нулевые строки

Пустой считается символьная строка `""` нулевой длины. Для того чтобы проверить, является ли символьная строка пустой, достаточно составить выражение вида `if (str.length() == 0)` или `if (str.equals(""))`.

В Java пустая строка является объектом, в котором хранится нулевая (т.е. 0) длина символьной строки и пустое содержимое. Но в переменной типа `String` может также храниться специальное пустое значение `null`, указывающее на то, что в настоящий момент ни один из объектов не связан с данной переменной. (Подробнее пустое значение `null` обсуждается в главе 4.) Для того чтобы проверить, является ли символьная строка нулевой, т.е. содержит значение `null`, служит условие `if (str == null)`.

Иногда требуется проверить, не является ли символьная строка ни пустой, ни нулевой. Для этой цели служит условие `if (str != null && str.length() != 0)`. Но сначала нужно проверить, не является ли символьная строка `str` нулевой. Как будет показано в главе 4, вызов метода с пустым значением `null` считается ошибкой.

Кодовые точки и кодовые единицы

В Java символьные строки реализованы в виде последовательности значений типа `char`. Как пояснялось ранее в разделе “Тип данных `char`”, с помощью типа данных `char` можно задавать кодовые единицы, представляющие кодовые точки уникода в кодировке UTF-16. Наиболее часто употребляемые символы уникода представлены одной кодовой единицей, а дополнительные символы — парами кодовых единиц.

Метод `length()` возвращает количество кодовых единиц для данной строки в кодировке UTF-16. Ниже приведен пример применения данного метода.

```
String greeting = "Hello";
int n = greeting.length(); // значение n равно 5
```

Чтобы определить истинную длину, представляющую собой число кодовых точек, нужно сделать следующий вызов:

```
int cpCount = greeting.codePointCount(0, greeting.length());
```

При вызове метода `s.charAt(n)` возвращается кодовая единица на позиции `n`, где `n` находится в пределах от 0 до `s.length() - 1`. Ниже приведены примеры вызова данного метода.

```
char first = greeting.charAt(0); // первый символ - 'H'
char last = greeting.charAt(4); // последний символ - 'o'
```

Для получения *i*-й кодовой точки служат приведенные ниже выражения.

```
int index = greeting.offsetByCodePoints(0, i);
int cp = greeting.codePointAt(index);
```



НА ЗАМЕТКУ! В Java, как и в C/C++, кодовые единицы и кодовые точки в символьных строках подсчитываются с нулевой позиции.

А зачем вообще обсуждать кодовые единицы? Рассмотрим следующую символьную строку:

```
0 is the set of octonions
```

Для представления символа `0` требуются две кодовые единицы в кодировке UTF-16. В результате приведенного ниже вызова будет получен не код пробела, а вторая кодовая единица символа `0`:

```
char ch = sentence.charAt(1)
```

Чтобы избежать подобных осложнений, не следует применять тип `char`, поскольку он представляет символы на слишком низком уровне.

Если же требуется просмотреть строку посимвольно, т.е. получить по очереди каждую кодовую точку, то для этой цели можно воспользоваться фрагментом кода, аналогичным приведенному ниже.

```
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i += 2;
else i++;
```

А организовать просмотр строки и в обратном направлении можно следующим образом:

```
i--;
if (Character.isSurrogate(sentence.charAt(i))) i--;
int cp = sentence.codePointAt(i);
```

Прикладной интерфейс API класса `String`

В Java класс `String` содержит свыше 50 методов. Многие из них оказались очень полезными и применяются очень часто. Приведенный ниже фрагмент описания прикладного интерфейса API данного класса содержит наиболее полезные из них.



НА ЗАМЕТКУ! Время от времени в качестве вспомогательного материала на страницах этой книги будут появляться фрагменты описания прикладного программного интерфейса API для Java. Каждый такой фрагмент начинается с имени класса, например `java.lang.String`, где `java.lang` — пакет (подробнее о пакетах речь пойдет в главе 4). После имени класса следуют имена конкретных методов и их описание.

Обычно в подобных описаниях перечисляются не все, а только наиболее употребительные методы конкретного класса. Полный перечень методов описываемого класса можно найти в оперативно доступной документации, как поясняется в соответствующем разделе далее в этой главе. Кроме того, приводится номер версии, в которой был реализован класс. Если же тот или иной метод был добавлен позже, то у него имеется отдельный номер версии.

`java.lang.String` 1.0

- `char charAt(int index)`
Возвращает символ, расположенный на указанной позиции. Вызывать этот метод следует только в том случае, если интересуют низкоуровневые кодовые единицы.
- `int codePointAt(int index)` 5.0
Возвращает кодовую точку, начало или конец которой находится на указанной позиции.
- `int offsetByCodePoints(int startIndex, int cpCount)` 5.0
Возвращает индекс кодовой точки, которая отстоит на количество `cpCount` кодовых точек от исходной кодовой точки на позиции `startIndex`.
- `int compareTo(String other)`
Возвращает отрицательное значение, если данная строка лексикографически предшествует строке `other`, положительное значение — если строка `other` предшествует данной строке, и нулевое значение — если строки одинаковы.
- `boolean endsWith(String suffix)`
Возвращает логическое значение `true`, если строка оканчивается подстрокой `suffix`.
- `boolean equals(Object other)`
Возвращает логическое значение `true`, если данная строка совпадает со строкой `other`.
- `boolean equalsIgnoreCase(String other)`
Возвращает логическое значение `true`, если данная строка совпадает со строкой `other` без учета регистра символов.
- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`
- `int indexOf(int cp)`
- `int indexOf(int cp, int fromIndex)`
Возвращают индекс начала первой подстроки, совпадающей со строкой `str`, или же индекс указанной кодовой точки `cp`. Отсчет начинается с позиции 0 или `fromIndex`. Если указанная подстрока отсутствует в данной строке, возвращается значение, равное `-1`.
- `int lastIndexOf(String str)`

- `int lastIndexOf(String str, int fromIndex)`
- `int lastindexOf(int cp)`
- `int lastindexOf(int cp, int fromIndex)`

Возвращают начало последней подстроки, равной строке `str`, или же индекс указанной кодовой точки `cp`. Отсчет начинается с конца строки или с позиции `fromIndex`. Если указанная подстрока отсутствует в данной строке, возвращается значение, равное `-1`.

- `int length()`

Возвращает длину строки.

- `int codePointCount(int startIndex, int endIndex)` 5.0

Возвращает количество кодовых точек между позициями `startIndex` и `endIndex` - 1. Неспаренные суррогаты считаются кодовыми точками.

- `String replace(CharSequence oldString, CharSequence newString)`

Возвращает новую строку, которая получается путем замены всех подстрок, совпадающих с `oldString`, строкой `newString`. В качестве параметров типа `CharSequence` могут быть указаны объекты типа `String` или `StringBuilder`.

- `boolean startsWith(String prefix)`

Возвращает логическое значение `true`, если строка начинается с подстроки `prefix`.

- `String substring(int beginIndex)`

- `String substring(int beginIndex, int endIndex)`

Возвращают новую строку, состоящую из всех кодовых единиц, начиная с позиции `beginIndex` и до конца строки или позиции `endIndex` - 1.

- `String toLowerCase()`

Возвращает новую строку, состоящую из всех символов исходной строки. Отличие между исходной и результирующей строками состоит в том, что все буквы преобразуются в нижний регистр.

- `String toUpperCase()`

Возвращает новую строку, состоящую из всех символов исходной строки. Отличие между исходной и результирующей строкой состоит в том, что все буквы преобразуются в верхний регистр.

- `String trim()`

Возвращает новую строку, из которой исключены все начальные и конечные пробелы.

Оперативно доступная документация на API

Как упоминалось выше, класс `String` содержит немало методов. Более того, в стандартной библиотеке существуют тысячи классов, содержащих огромное количество методов. И запомнить все полезные классы и методы просто невозможно. Поэтому очень важно уметь пользоваться интерактивной оперативно доступной документацией на прикладной интерфейс API, чтобы быстро находить в ней сведения о классах и методах из стандартной библиотеки. Такая документация является

составной частью комплекта JDK. Она представлена в формате HTML. Откройте в окне избранного браузера документ `docs/api/index.html`, находящийся в том каталоге, где установлен комплект JDK. В итоге появится страница, приведенная на рис. 3.2.

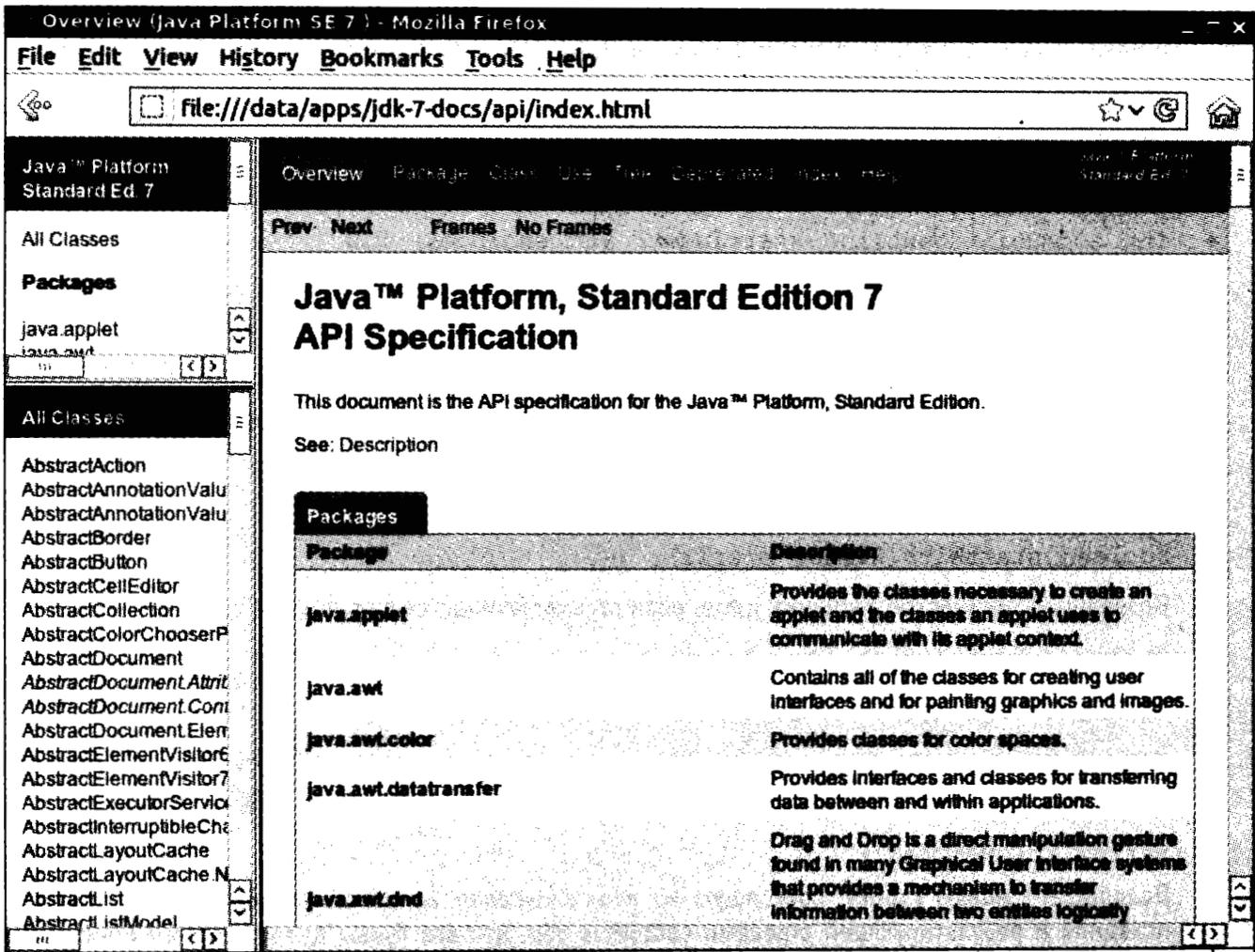


Рис. 3.2. Три панели на странице с оперативно доступной документацией на прикладной интерфейс API

Страница документа разделена на три панели. На небольшой панели в верхнем левом углу отображаются имена всех доступных пакетов. Ниже нее на более крупной панели перечислены все классы. Щелкните мышью на любом из имен классов, и сведения об этом классе появятся на крупной панели справа (рис. 3.3). Так, если требуется получить дополнительные сведения о методах класса `String`, прокрутите содержимое левой нижней панели до тех пор, пока не увидите ссылку `String`, а затем щелкните на ней.

Далее прокручивайте содержимое правой панели до тех пор, пока не увидите краткое описание всех методов. Методы расположены в алфавитном порядке (рис. 3.4). Щелкните на имени интересующего вас метода, чтобы получить его подробное описание (рис. 3.5). Так, если вы щелкнете на ссылке `compareToIgnoreCase`, то получите описание одноименного метода.

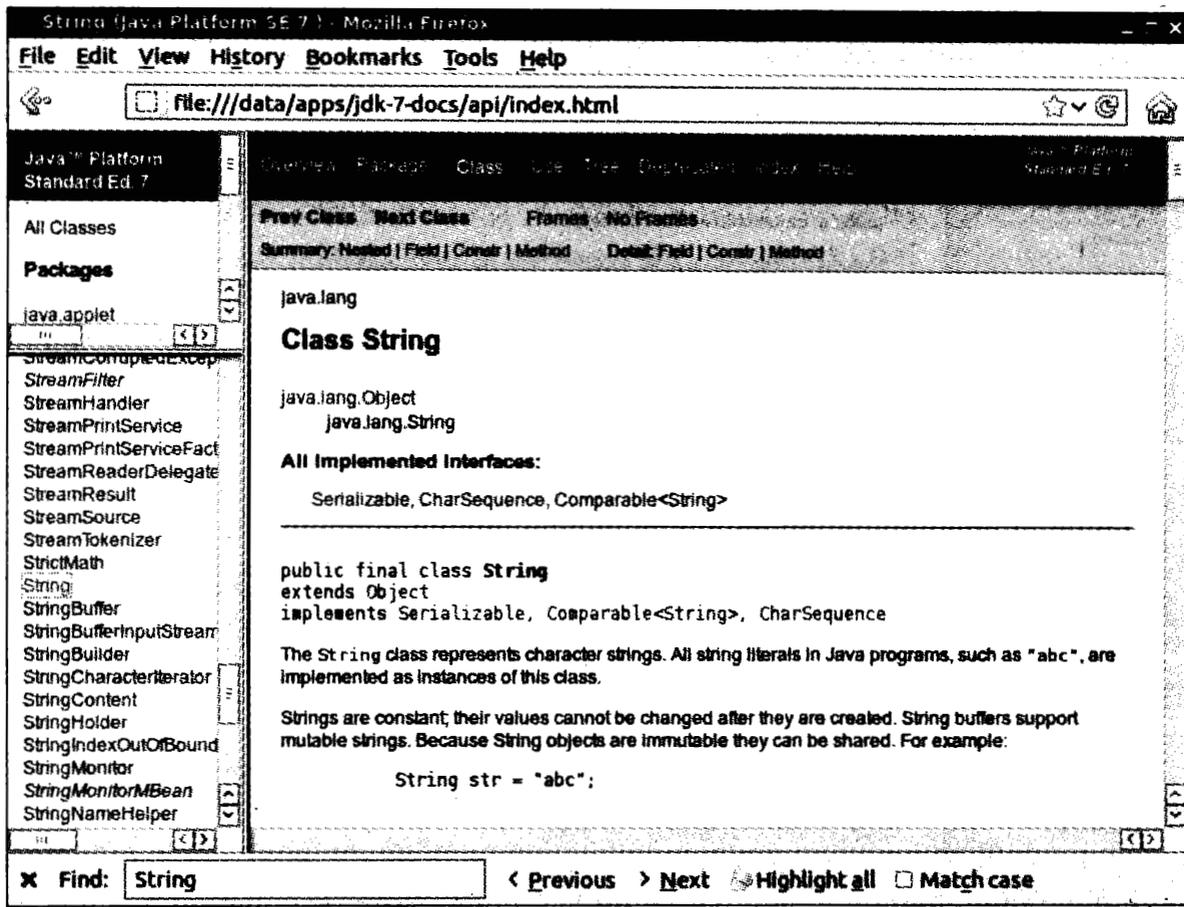


Рис. 3.3. Описание класса String

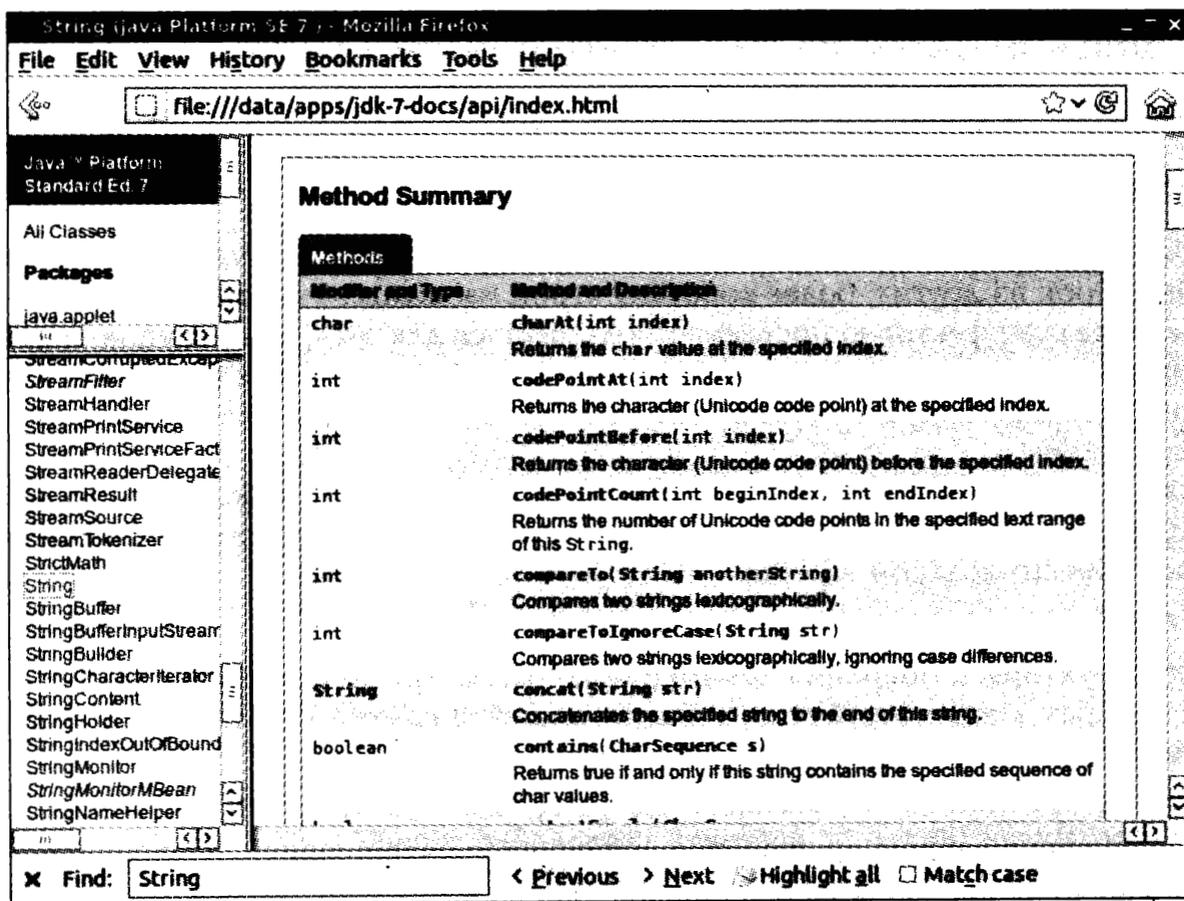


Рис. 3.4. Перечень методов из класса String

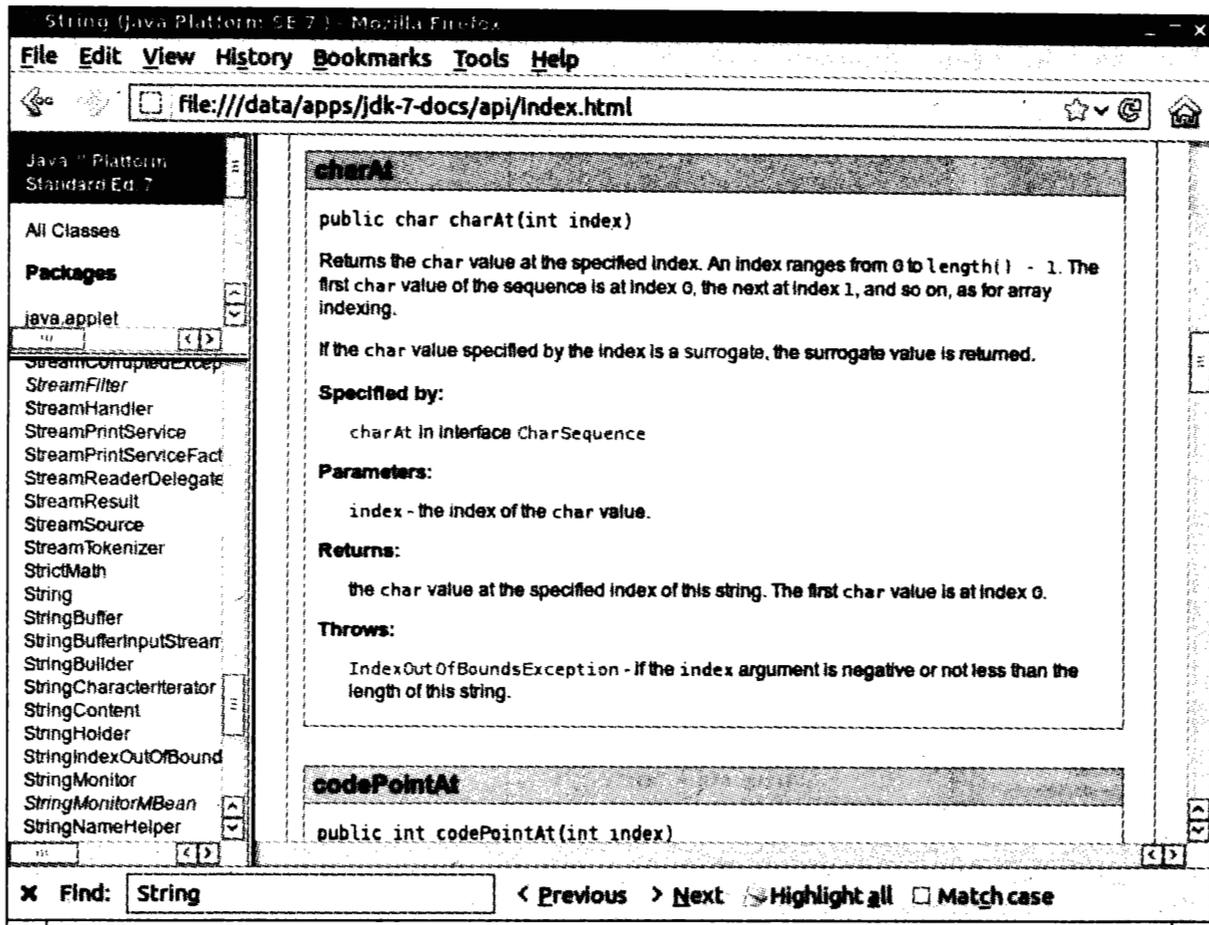


Рис. 3.5. Подробное описание метода `charAt()` из класса `String`



СОВЕТ. Непременно сделайте в браузере закладку на документ `docs/api/index.html`.

Построение символьных строк

Время от времени у вас будет возникать потребность в составлении одних символьных строк из других, более коротких строк, вводимых с клавиатуры или из файла. Было бы неэффективно постоянно пользоваться для этой цели сцеплением строк. Ведь при каждом сцеплении символьных строк конструируется новый объект типа `String`, на что расходуется время и память. Этого можно избежать, применяя класс `StringBuilder`.

Если требуется создать символьную строку из нескольких небольших фрагментов, выполните следующие действия. Во-первых, сконструируйте пустой объект в качестве строителя символьной строки:

```
StringBuilder builder = new StringBuilder();
```

(Конструкторы и операция `new` подробно рассматриваются в главе 4.)

Всякий раз, когда потребуется добавить новый фрагмент в символьную строку, вызовите метод `append()`, как показано ниже.

```
builder.append(ch); // добавить единственный символ
builder.append(str); // добавить символьную строку
```

Завершив составление символьной строки, вызовите метод `toString()`. Таким образом, вы получите объект типа `String`, состоящий из последовательности символов, содержащихся в объекте построителя символьных строк:

```
String completedString = builder.toString();
```



НА ЗАМЕТКУ! Класс `StringBuilder` появился в версии JDK 5.0. Его предшественник, класс `StringBuffer`, менее эффективен, но позволяет добавлять и удалять символы во многих потоках. Если же редактирование символьной строки происходит полностью в одном потоке (как это обычно и бывает), то следует, напротив, использовать класс `StringBuilder`. А прикладные интерфейсы API обоих классов идентичны.

Следующее описание прикладного интерфейса API содержит наиболее употребительные методы из класса `StringBuilder`.

`java.lang.StringBuilder` 5.0

- `StringBuilder()`
Конструирует пустой объект построителя символьных строк.
- `int length()`
Возвращает количество кодовых единиц из объекта построителя символьных строк или буфера.
- `StringBuilder append(String str)`
Добавляет строку и возвращает ссылку `this` на текущий объект построителя символьных строк.
- `StringBuilder append(char c)`
Добавляет кодовую единицу и возвращает ссылку `this` на текущий объект построителя символьных строк.
- `StringBuilder appendCodePoint(int cp)`
Добавляет кодовую точку, преобразуя ее в одну или две кодовые единицы, возвращает ссылку `this` на текущий объект построителя символьных строк.
- `void setCharAt(int i, int c)`
Устанавливает символ `c` на позиции `i`-й кодовой единицы.
- `StringBuilder insert(int offset, String str)`
Вставляет строку на позиции `offset` и возвращает ссылку `this` на текущий объект построителя символьных строк.
- `StringBuilder insert(int offset, char c)`
Вставляет кодовую единицу на позиции `offset` и возвращает ссылку `this` на текущий объект построителя символьных строк.
- `StringBuilder delete(int startIndex, int endIndex)`
Удаляет кодовые единицы со смещениями от `startIndex` до `endIndex - 1` и возвращает ссылку `this` на текущий объект построителя символьных строк.
- `String toString()`
Вставляет строку, содержащую те же данные, что и объект построителя символьных строк или буфер.

Ввод и вывод

Для того чтобы немного “оживить” программы, рассматриваемые здесь в качестве примеров, организуем ввод информации и форматирование выводимых данных. Конечно, в современных приложениях для ввода данных используются элементы ГПИ, но для программирования такого интерфейса требуются приемы и инструментальные средства, которые мы пока еще не рассматривали. Наша текущая цель — ознакомить вас с основными языковыми средствами Java, и поэтому мы ограничимся лишь консольным вводом-выводом. А вопросы программирования ГПИ будут подробно рассмотрены в главах 7–9.

Чтение вводимых данных

Как вам должно быть уже известно, информацию можно легко вывести в стандартный поток вывода (т.е. в консольное окно), вызвав метод `System.out.println()`. А вот организовать чтение из стандартного потока ввода `System.in` (т.е. с консоли) не так-то просто. Для этого придется создать объект типа `Scanner` и связать его со стандартным потоком ввода `System.in`, как показано ниже.

```
Scanner in = new Scanner(System.in);
```

(Конструкторы и операция `new` подробно рассматриваются в главе 4.)

Сделав это, вы получите в свое распоряжение многочисленные методы из класса `Scanner`, предназначенные для чтения вводимых данных. Например, метод `nextLine()` осуществляет чтение вводимой строки, как показано ниже.

```
System.out.print("What is your name? ");  
String name = in.nextLine();
```

В данном случае метод `nextLine()` был применен, потому что вводимая строка может содержать пробелы. А для того чтобы прочитать одно слово, отделяемое пробелами, можно сделать следующий вызов:

```
String firstName = in.next();
```

Для чтения целочисленного значения служит метод `nextInt()`:

```
System.out.print("How old are you? ");  
int age = in.nextInt();
```

Как нетрудно догадаться, метод `nextDouble()` осуществляет чтение очередного числового значения в формате с плавающей точкой.

Программа, код которой представлен в листинге 3.2, запрашивает имя пользователя и его возраст, а затем выводит сообщение наподобие следующего:

```
Hello, Cay. Next year, you'll be 52  
(Здравствуйтесь, Кей. В следующем году вам будет 52)
```

В первой строке кода этой программы содержится следующее выражение:

```
import java.util.*;
```

Класс `Scanner` принадлежит пакету `java.util`. Если вы собираетесь использовать в программе класс, не содержащийся в базовом пакете `java.lang`, вам придется включить в код своей программы директиву `import`. Более подробно пакеты и директива `import` будут рассмотрены в главе 4.

Листинг 3.2. Исходный код из файла InputTest/InputTest.java

```
1 import java.util.*;
2
3 /**
4  * Эта программа демонстрирует консольный ввод
5  * @version 1.10 2004-02-10
6  * @author Cay Horstmann
7  */
8 public class InputTest
9 {
10     public static void main(String[] args)
11     {
12         Scanner in = new Scanner(System.in);
13
14         // получить первую вводимую строку
15         System.out.print("What is your name? ");
16         String name = in.nextLine();
17
18         // получить вторую вводимую строку
19         System.out.print("How old are you? ");
20         int age = in.nextInt();
21
22         // вывести результат на консоль
23         System.out.println(
24             "Hello, " + name + ". Next year, you'll be " + (age + 1));
25     }
26 }
```



НА ЗАМЕТКУ! Класс `Scanner` не подходит для ввода паролей с консоли, поскольку такой ввод будет явно виден всякому. В версии Java SE 6 появился класс `Console`, специально предназначенный для этой цели. Чтобы организовать ввод пароля с консоли, можно воспользоваться следующим фрагментом кода:

```
Console cons = System.console();
String username = cons.readLine("User name: ");
char[] passwd = cons.readPassword("Password: ");
```

Из соображений безопасности пароль возвращается в виде массива символов, а не в виде символьной строки. После обработки пароля следует немедленно перезаписать элементы массива значением заполнителя. (Обработку массивов мы обсудим далее в этой главе.)

Обработка вводимых данных с помощью объекта типа `Console` не так удобна, как с помощью объекта типа `Scanner`. Ведь вводимые данные в этом случае можно читать только построчно. В классе `Console` отсутствуют методы для чтения отдельных слов или чисел.

java.util.Scanner 5.0

- `Scanner(InputStream in)`
Конструирует объект типа `Scanner` на основе заданного потока ввода.
- `String nextLine()`
Читает очередную строку.

- `String next()`
Читает очередное вводимое слово, отделяемое пробелами.
- `int nextInt()`
- `double nextDouble()`
Читает очередную последовательность символов, представляющую целое число или число с плавающей точкой, выполняя соответствующее преобразование.
- `boolean hasNext()`
Проверяет, присутствует ли еще одно слово в потоке ввода.
- `boolean hasNextInt()`
- `boolean hasNextDouble()`
Проверяют, присутствует ли в потоке ввода последовательность символов, представляющая целое число или число с плавающей точкой.

`java.lang.System` 1.0

- `static Console console()`
Возвращает объект типа `Console` для взаимодействия с пользователем через консольное окно, а если такое взаимодействие невозможно, — пустое значение `null`. Объект типа `Console` доступен в любой программе, запущенной в консольном окне. В противном случае его доступность зависит от конкретной системы.

`java.io.Console` 6

- `static char[] readPassword(String prompt, Object... args)`
- `static String readLine(String prompt, Object... args)`
Отображают приглашение и читают вводимые пользователем данные до тех пор, пока не получают конец вводимой строки. Параметры *args* могут быть использованы для предоставления аргументов форматирования, как поясняется в следующем разделе.

Форматирование выводимых данных

Числовое значение *x* можно вывести на консоль с помощью выражения `System.out.println(x)`. В результате на экране отобразится число с максимальным количеством значащих цифр, допустимых для данного типа. Так, в результате выполнения приведенного ниже фрагмента кода на экран будет выведено число **3333.333333333335**.

```
double x = 10000.0 / 3.0;  
System.out.print(x);
```

В ряде случаев это вызывает осложнения. Так, если требуется вывести на экран сумму в долларах и центах, большое количество цифр ее затруднит восприятие.

В ранних версиях Java процесс форматирования чисел был сопряжен с определенными трудностями. К счастью, в версии Java SE 5.0 была реализована в виде метода `printf()` почтенная функция, хорошо известная из библиотеки C.

Например, с помощью приведенного ниже оператора можно вывести значение **x** в виде числа, ширина поля которого составляет 8 цифр, а дробная часть равна двум цифрам. (Число цифр дробной части иначе называется *точностью*.)

```
System.out.printf("%8.2f", x);
```

В результате на экран будет выведено семь символов, не считая начальных пробелов.

```
3333.33
```

Метод `printf()` позволяет задавать произвольное количество параметров. Ниже приведен пример вызова этого метода с несколькими параметрами.

```
System.out.printf("Hello, %s. Next year, you'll be %d", name, age);
```

Каждый *спецификатор формата*, начинающийся с символа **%**, заменяется соответствующим параметром. А *символ преобразования*, которым завершается спецификатор формата, задает тип форматируемого значения: **f** — число с плавающей точкой; **s** — символьная строка; **d** — десятичное число. Все символы преобразования приведены в табл. 3.5.

Таблица 3.5. Символы преобразования для метода `printf()`

Символ преобразования	Тип	Пример
d	Десятичное целое	159
x	Шестнадцатеричное целое	9f
f	Число с фиксированной или плавающей точкой	15.9
e	Число с плавающей точкой в экспоненциальной форме	1.59e+01
g	Число с плавающей точкой в общем формате (чаще всего используется формат e или f , в зависимости от того, какой из них дает более короткую запись)	—
a	Шестнадцатеричное представление числа с плавающей точкой	0x1.fcddp3
s	Символьная строка	Hello
c	Символ	H
b	Логическое значение	true
h	Хеш-код	42628b2
tx	Дата и время	См. табл. 3.7
%	Знак процента	%
n	Разделитель строк, зависящий от платформы	—

В спецификаторе формата могут также присутствовать *флаги*, управляющие форматом выходных данных. Назначение всех флагов вкратце описано в табл. 3.6. Например, запятая, указываемая в качестве флага, добавляет разделители групп. Так, в результате выполнения приведенного ниже оператора на экран будет выведена строка **3,333.33**.

```
System.out.printf("%,.2f", 10000.0 / 3.0);
```

В одном спецификаторе формата можно использовать несколько флагов, например, последовательность символов "%, (.2f" указывает на то, что при выводе будут использованы разделители групп, а отрицательные числа будут заключены в скобки.



НА ЗАМЕТКУ! Преобразование типа `s` можно использовать для форматирования любого объекта. Если этот объект реализует интерфейс `Formattable`, то вызывается метод `formatTo()`. В противном случае для преобразования объекта в символьную строку применяется метод `toString()`. Метод `toString()` будет обсуждаться в главе 5, а интерфейсы — в главе 6.

Таблица 3.6. Флаги для метода `printf()`

Флаг	Назначение	Пример
+	Выводит знак не только для отрицательных, но и для положительных чисел	+3333.33
Пробел	Добавляет пробел перед положительными числами	3333.33
0	Выводит начальные нули	003333.33
-	Выравнивает поле по левому краю	3333.33
(Заключает отрицательные числа в скобки	(3333.33)
,	Задаёт использование разделителя групп	3,333.33
# (для формата <code>f</code>)	Всегда отображает десятичную точку	3,333
# (для формата <code>x</code> или <code>o</code>)	Добавляет префикс <code>0x</code> или <code>0</code>	0x3333
\$	Определяет индекс параметра, предназначенного для форматирования. Например, выражение <code>%1\$d %1\$x</code> указывает на то, что первый параметр должен быть сначала выведен в десятичной, а затем в шестнадцатеричной форме	159 9F
<	Задаёт форматирование того же самого значения, которое отформатировано предыдущим спецификатором. Например, выражение <code>%d %<x</code> указывает на то, что одно и то же значение должно быть представлено как в десятичной, так и в шестнадцатеричной форме	159 9F

Для составления отформатированной символьной строки без последующего ее вывода можно вызвать статический метод `String.format()`, как показано ниже.

```
String message =
    String.format("Hello, %s. Next year, you'll be %d", name, age);
```

Несмотря на то что тип `Date` будет подробно рассмотрен лишь в главе 4, для того чтобы завершить обсуждение метода `printf()`, следует хотя бы бегло коснуться средств форматирования даты и времени. Для этой цели используется последовательность из двух символов, начинающаяся с буквы `t`, после которой следует одна из букв, приведенных в табл. 3.7. Ниже приведен пример такого форматирования.

```
System.out.printf("%tc", new Date());
```

В результате выполнения данного оператора выводятся текущая дата и время:

```
Mon Feb 09 18:05:19 PST 2012
```

Как следует из табл. 3.7, некоторые форматы предполагают отображение лишь отдельных составляющих даты — дня или месяца. Было бы неразумно многократно задавать дату лишь для того, чтобы отформатировать различные ее составляющие. По этой причине в строке, определяющей формат даты, может задаваться индекс формируемого параметра. Индекс должен следовать сразу после знака `%` и завершаться знаком `$`, как показано ниже.

```
System.out.printf("%1$s %2$tB %2$te, %2$tY", "Due date:", new Date());
```

В результате выполнения этой строки кода будет выведена следующая строка:

```
Due date: February 9, 2012
```

Можно также воспользоваться флагом `<`, который означает, что форматированию подлежит тот же самый параметр, который был отформатирован последним. Так, приведенная ниже строка кода дает точно такой же результат, как и упомянутая выше.

Таблица 3.7. Символы для форматирования даты и времени

Символ преобразования	Выходные данные	Пример
c	Полные сведения о дате и времени	Mon Feb 04 18:05:19 PST 2012
F	Дата в формате ISO 8601	2012-02-04
D	Дата в формате, принятом в США (месяц/день/год)	02/04/2012
T	Время в 24-часовом формате	18:05:19
t	Время в 12-часовом формате	06:05:19 pm
R	Время в 24-часовом формате (без секунд)	18:05
Y	Год в виде четырех цифр (с начальными нулями, если требуется)	2012
y	Год в виде двух последних цифр (с начальными нулями, если требуется)	12
C	Год в виде двух первых цифр (с начальными нулями, если требуется)	20
B	Полное название месяца	February
ь или h	Сокращенное название месяца	Feb
m	Месяц в виде двух цифр (с начальными нулями, если требуется)	02
d	День в виде двух цифр (с начальными нулями, если требуется)	09
e	День в виде одной или двух цифр (без начальных нулей)	4
A	Полное название дня недели	Monday
a	Сокращенное название дня недели	Mon
j	День в году в виде трех цифр в пределах от 001 до 366 (с начальными нулями, если требуется)	035
H	Час в виде двух цифр в пределах от 00 до 23 (с начальными нулями, если требуется)	18
k	Час в виде одной или двух цифр в пределах от 0 до 23 (без начальных нулей)	9

Окончание табл. 3.7

Символ преобразования	Выходные данные	Пример
I	Час в виде двух цифр в пределах от 01 до 12 (с начальными нулями, если требуется)	06
l	Час в виде одной или двух цифр в пределах от 01 до 12 (без начальных нулей)	6
M	Минуты в виде двух цифр (с начальными нулями, если требуется)	05
S	Секунды в виде двух цифр (с начальными нулями, если требуется)	19
L	Миллисекунды в виде трех цифр (с начальными нулями, если требуется)	047
N	Наносекунды в виде девяти цифр (с начальными нулями, если требуется)	047000000
P	Метка времени до полудня или после полудня прописными буквами	AM или PM
p	Метка времени до полудня или после полудня строчными буквами	am или pm
z	Смещение относительно времени по Гринвичу (GMT) по стандарту RFC 822	-0800
Z	Часовой пояс	PST (Стандартное тихоокеанское время)
s	Количество секунд от начала отсчета времени 1970-01-01 00:00:00 GMT	1078884319
Q	Количество миллисекунд от начала отсчета времени 1970-01-01 00:00:00 GMT	1078884319047



ВНИМАНИЕ! Индексы аргументов начинаются с единицы, а не с нуля. Так, выражение %1\$ задает форматирование первого параметра. Это сделано для того, чтобы избежать конфликтов с флагом 0.

Итак, мы рассмотрели все особенности применения метода `printf()`. На рис. 3.6 приведена блок-схема, наглядно показывающая синтаксический порядок указания спецификаторов формата.

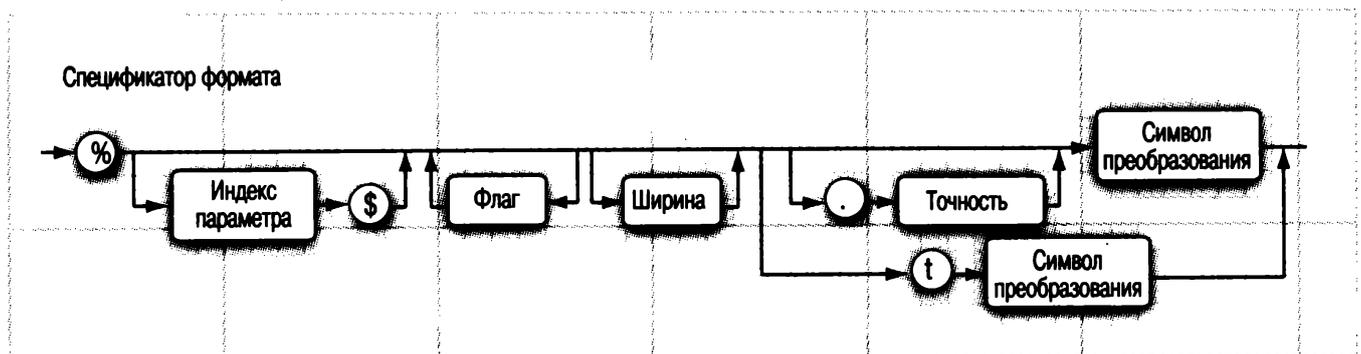


Рис. 3.6. Синтаксический порядок указания спецификаторов формата

Файловый ввод и вывод

Чтобы прочитать данные из файла, достаточно сконструировать объект типа **Scanner**:

```
Scanner in = new Scanner(Paths.get("myfile.txt"));
```

Если имя файла содержит знаки обратной косой черты, их следует экранировать дополнительными знаками обратной косой черты, как, например: "c:\\mydirectory\\myfile.txt". После этого можно произвести чтение из файла, используя любые упомянутые выше методы из класса **Scanner**.

А для того чтобы записать данные в файл, достаточно сконструировать объект типа **PrintWriter**, указав в его конструкторе имя файла, как показано ниже.

```
PrintWriter out = new PrintWriter("myfile.txt");
```

Если файл еще не существует, он создается. Для вывода в файл можно воспользоваться методами `print()`, `println()` и `printf()` точно так же, как это делается для вывода на консоль (или в стандартный поток вывода `System.out`).



ВНИМАНИЕ! Объект типа **Scanner** можно сконструировать со строковым параметром, но в этом случае символьная строка будет интерпретирована как данные, а не как имя файла. Так, если вызвать конструктор следующим образом:

```
Scanner in = new Scanner("myfile.txt"); // Ошибка?
```

объект типа **Scanner** интерпретирует заданное имя файла как отдельные символы 'm', 'y', 'f' и т.д. Но ведь это совсем не то, что требуется.



НА ЗАМЕТКУ! Когда указывается относительное имя файла, например "myfile.txt", "mydirectory/myfile.txt" или "../myfile.txt", то поиск файла осуществляется относительно того каталога, в котором была запущена виртуальная машина Java. Если запустить программу на выполнение из командной строки следующим образом:

```
java MyProg
```

то начальным окажется текущий каталог командной оболочки. Но если программа запускается на выполнение в ИСР, то начальный каталог определяется этой средой. Задать начальный каталог можно, сделав следующий вызов:

```
String dir = System.getProperty("user.dir");
```

Если вы испытываете трудности при обнаружении файлов, попробуйте применить абсолютные путевые имена вроде "c:\\mydirectory\\myfile.txt" или "/home/me/mydirectory/myfile.txt".

Как видите, обращаться к файлам так же легко, как и при консольном вводе и выводе в стандартные потоки `System.in` и `System.out` соответственно. Правда, здесь имеется одна уловка: если вы конструируете объект типа **Scanner** с файлом, который еще не существует, или объект типа **PrintWriter** с именем файла, который не может быть создан, возникает исключение. Компилятор Java рассматривает подобные исключения как более серьезные, чем, например, исключение при делении на нуль. В главе 11 будут рассмотрены различные способы обработки исключений. А до тех пор достаточно уведомить компилятор о том, что при файловом вводе и выводе

может возникнуть исключение типа “файл не найден”. Для этого в метод `main()` вводится оператор `throws`:

```
public static void main(String[] args) throws FileNotFoundException
{
    Scanner in = new Scanner(new File("myfile.txt"));
    . . .
}
```

Теперь вы знаете, как читать и записывать текстовые данные в файлы. Более сложные вопросы файлового ввода-вывода, в том числе применение различных кодировок символов, обработка двоичных данных, чтение каталогов и запись архивных файлов, рассматриваются в главе 1 второго тома данной книги.



НА ЗАМЕТКУ! Запуская программу из командной строки, вы можете воспользоваться синтаксисом перенаправления ввода-вывода из командной оболочки, чтобы направить любой файл в стандартные потоки `System.in` и `System.out`, как показано ниже.

```
java MyProg < myfile.txt > output.txt
```

В этом случае вам не придется организовывать обработку исключения типа `FileNotFoundException`.

java.util.Scanner 5.0

- `Scanner(File f)`
Конструирует объект типа `Scanner`, который читает данные из указанного файла.
- `Scanner(String data)`
Конструирует объект типа `Scanner`, который читает данные из указанной символьной строки.

java.io.PrintWriter 1.1

- `PrintWriter(String fileName)`
Конструирует объект типа `PrintWriter`, который записывает данные в файл с указанным именем.

java.nio.file.Paths 7

- `static Path get(String pathname)`
Конструирует объект типа `Path` для ввода данных из файла с указанным путевым именем.

Управляющая логика

В Java, как и в любом другом языке программирования, в качестве управляющей логики программы служат условные операторы и циклы. Рассмотрим сначала

условные операторы, а затем перейдем к циклам. И завершим обсуждение управляющей логики довольно громоздким оператором `switch`, который можно применять для проверки многих значений одного выражения.



НА ЗАМЕТКУ С++! Языковые конструкции управляющей логики в Java такие же, как и в С и С++, за исключением двух особенностей. Среди них нет оператора безусловного перехода `goto`, но имеется версия оператора `break` с метками, который можно использовать для выхода из вложенного цикла (в С для этого пришлось бы применять оператор `goto`). Кроме того, в Java реализован вариант цикла `for`, не имеющий аналогов в С или С++. Его можно сравнить с циклом `foreach` в С#.

Область действия блоков

Прежде чем перейти к обсуждению конкретных языковых конструкций управляющей логики, необходимо рассмотреть блоки. *Блок*, или составной оператор, представляет собой произвольное количество простых операторов Java, заключенных в фигурные скобки. Блоки определяют область действия переменных. Блоки могут быть *вложенными* друг в друга. Ниже приведен пример блока, вложенного в другой блок в методе `main()`.

```
public static void main(String[] args)
{
    int n;
    ...
    {
        int k;
        ...
    } // переменная k определена только в этом блоке
}
```

В Java нельзя объявить переменные с одинаковым именем в двух вложенных блоках. Например, приведенный ниже фрагмент кода содержит ошибку и не будет скомпилирован.

```
public static void main(String[] args)
{
    int n;
    ...
    {
        int k;
        int n; // ОШИБКА: переопределить переменную n во внутреннем блоке нельзя
        ...
    }
}
```



НА ЗАМЕТКУ С++! В С++ переменные во вложенных блоках можно переопределять. Внутреннее определение маскирует внешнее. Это может привести к ошибкам, поэтому в Java подобный подход не реализован.

Условные операторы

Условный оператор `if` в Java имеет приведенную ниже форму. Условие должно быть заключено в скобки.

```
if (условие) оператор
```

В программах на Java, как и на большинстве других языков программирования, часто приходится выполнять много операторов в зависимости от истинности одного условия. В этом случае составляется блок операторов, как показано ниже.

```
{  
  Оператор1;  
  Оператор2;  
  ...  
}
```

Рассмотрим в качестве примера следующий фрагмент кода:

```
if (yourSales >= target)  
{  
  performance = "Satisfactory";  
  bonus = 100;  
}
```

В этом фрагменте кода все операторы, заключенные в фигурные скобки, будут выполнены при условии, что значение переменной `yourSales` больше значения переменной `target` или равно ему (рис. 3.7).

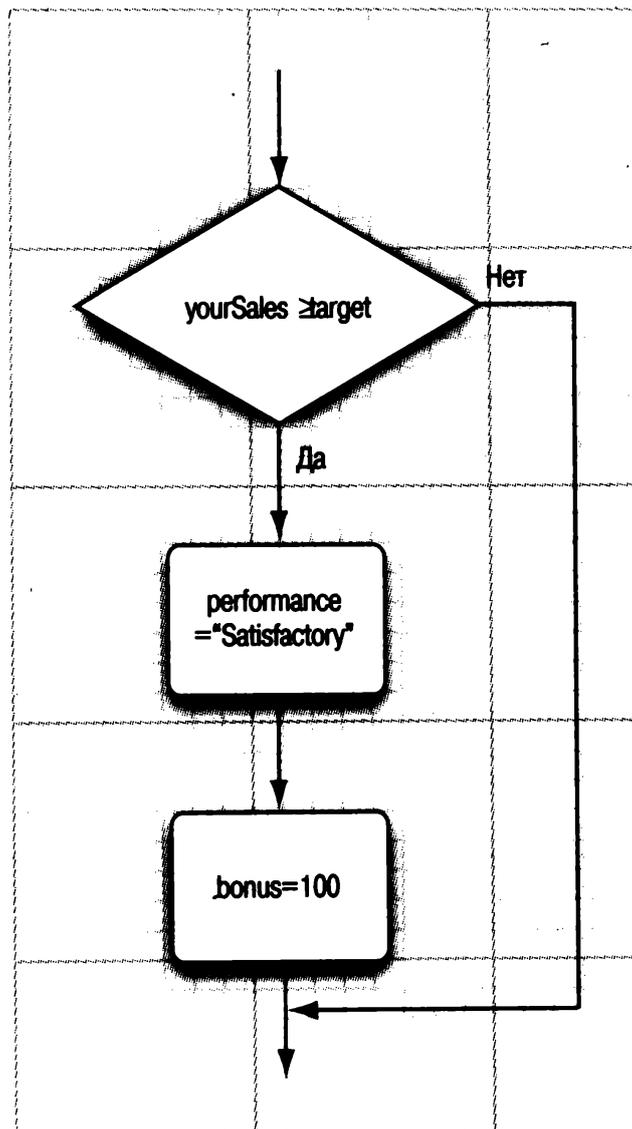


Рис. 3.7. Блок-схема, иллюстрирующая принцип действия условного оператора `if`



НА ЗАМЕТКУ! Блок (иногда называемый *составным оператором*) позволяет включать несколько (простых) операторов в любую языковую конструкцию Java, которая в противном случае допускает лишь один (простой) оператор.

Ниже приведена более общая форма условного оператора `if` в Java. А принцип его действия в данной форме наглядно показан на рис. 3.8.

`if (условие) оператор1 else оператор2`

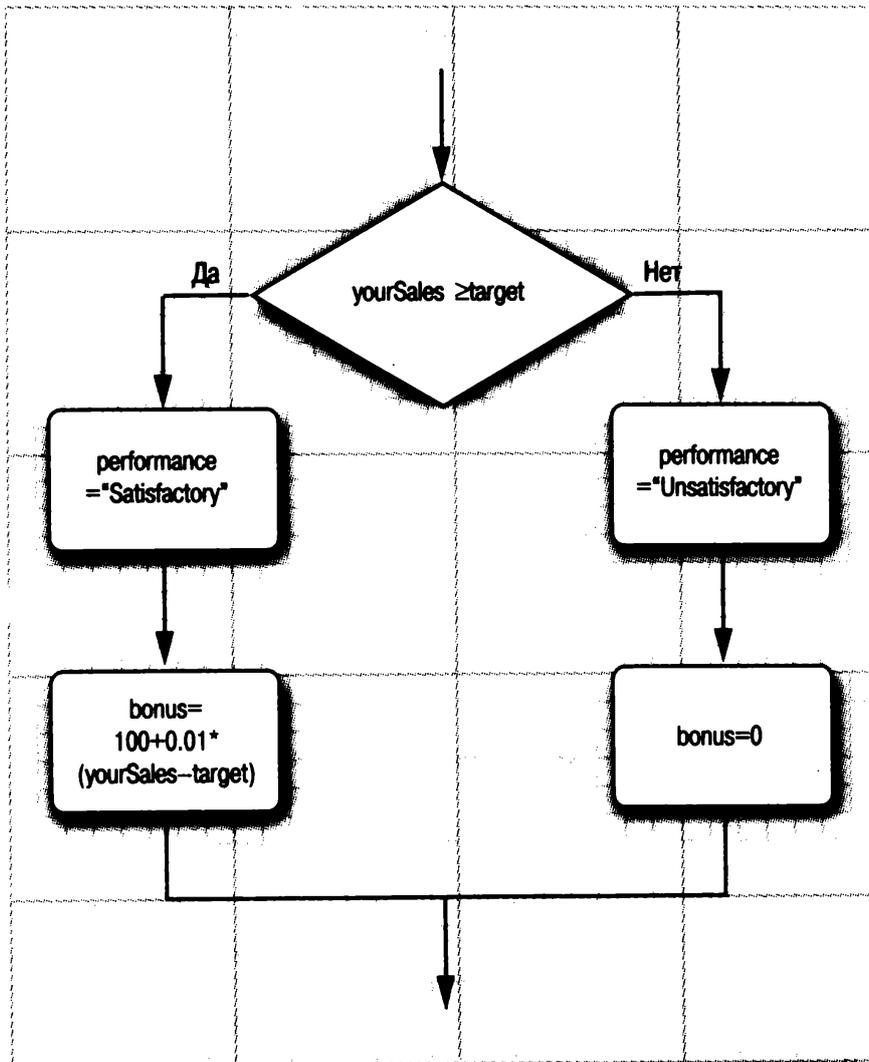


Рис. 3.8. Блок-схема, иллюстрирующая принцип действия условного оператора `if/else`

Ниже приведен пример употребления условного оператора `if/else` в коде.

```
if (yourSales >= target)
{
    performance = "Satisfactory";
    bonus = 100 + 0.01 * (yourSales - target);
}
else
{
    performance = "Unsatisfactory";
    bonus = 0;
}
```

Часть `else` данного оператора не является обязательной. Она объединяется с ближайшим условным оператором `if`. Таким образом, в следующей строке кода оператор `else` относится ко второму оператору `if`:

```
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

Разумеется, для повышения удобочитаемости такого кода следует воспользоваться фигурными скобками, как показано ниже.

```
if (x <= 0) { if (x == 0) sign = 0; else sign = -1; }
```

В программах на Java часто встречаются также повторяющиеся условные операторы `if...else if...` (рис. 3.9). Ниже приведен пример применения такой языковой конструкции в коде.

```
if (yourSales >= 2 * target)
{
    performance = "Excellent";
    bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
    performance = "Fine";
    bonus = 500;
}
else if (yourSales >= target)
{
    performance = "Satisfactory";
    bonus = 100;
}
```

Неопределенные циклы

Цикл `while` обеспечивает выполнение выражения (или группы операторов, составляющих блок) до тех пор, пока условие истинно, т.е. имеет логическое значение `true`. Ниже приведена общая форма записи этого цикла.

```
while (условие) оператор
```

Цикл `while` не будет выполнен ни разу, если его условие изначально ложно, т.е. имеет логическое значение `false` (рис. 3.10).

В листинге 3.3 приведен пример программы, подсчитывающей, сколько лет нужно вносить деньги на счет, чтобы накопить определенную сумму на заслуженный отдых. Считается, что каждый год вносится одна и та же сумма и процентная ставка не меняется. В данном примере в теле цикла увеличивается счетчик и обновляется сумма, накопленная на текущий момент. И так происходит до тех пор, пока итоговая сумма не превысит заданную величину:

```
while (balance < goal)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    years++;
}
System.out.println(years + " years.");
```

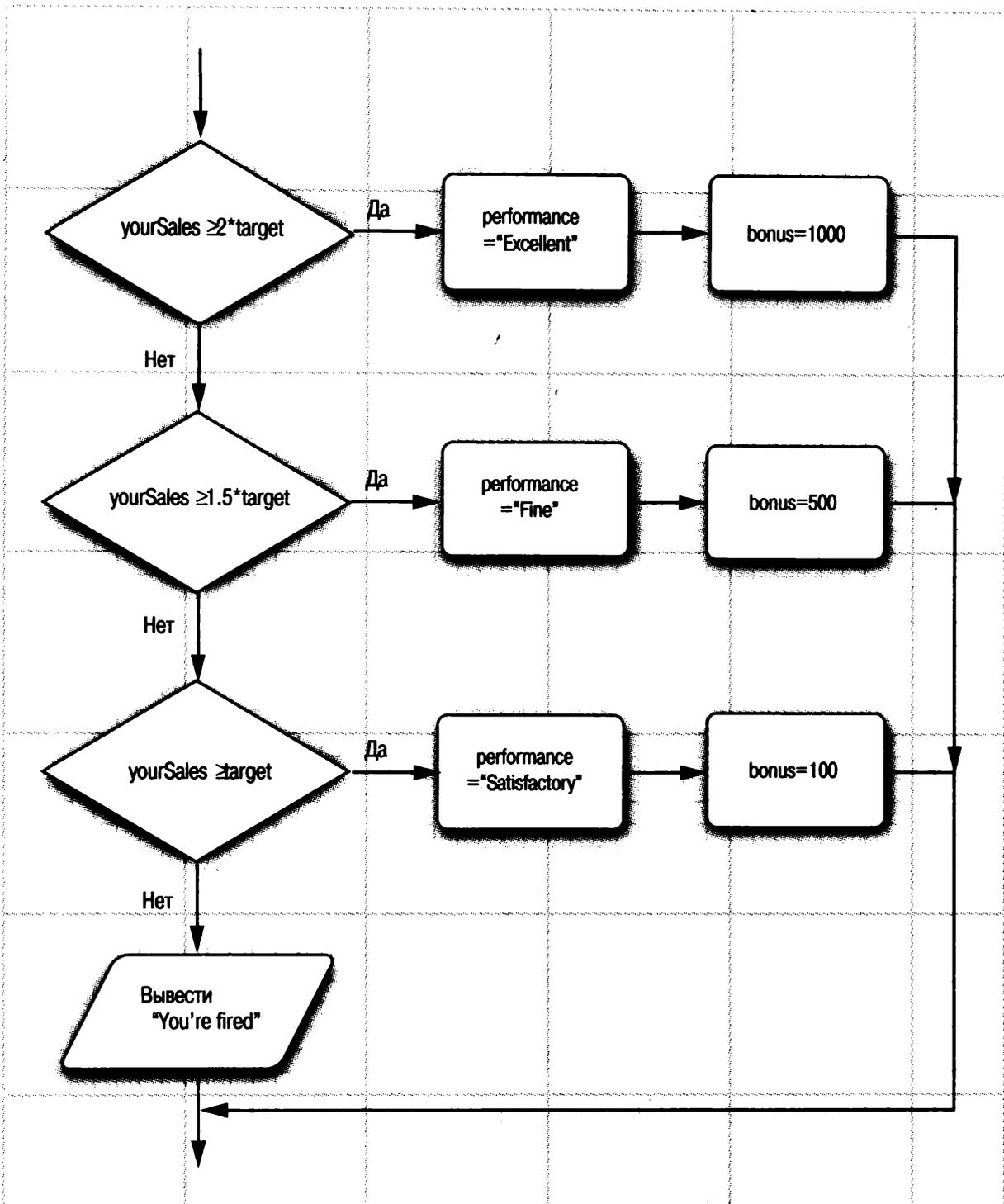


Рис. 3.9. Блок-схема, иллюстрирующая принцип действия условного оператора if/else if со множественным ветвлением

(Не особенно полагайтесь на эту программу при планировании своей пенсии. В ней не учтены такие немаловажные детали, как инфляция и ожидаемая продолжительность вашей жизни.)

Условие цикла while проверяется в самом начале. Следовательно, возможна ситуация, когда код, содержащийся в блоке, образующем тело цикла, вообще не будет выполнен. Если же требуется, чтобы блок выполнялся хотя бы один раз, проверку условия следует перенести в конец. Это можно сделать с помощью цикла do-while, общая форма которого записывается следующим образом:

```
do оператор while (условие);
```

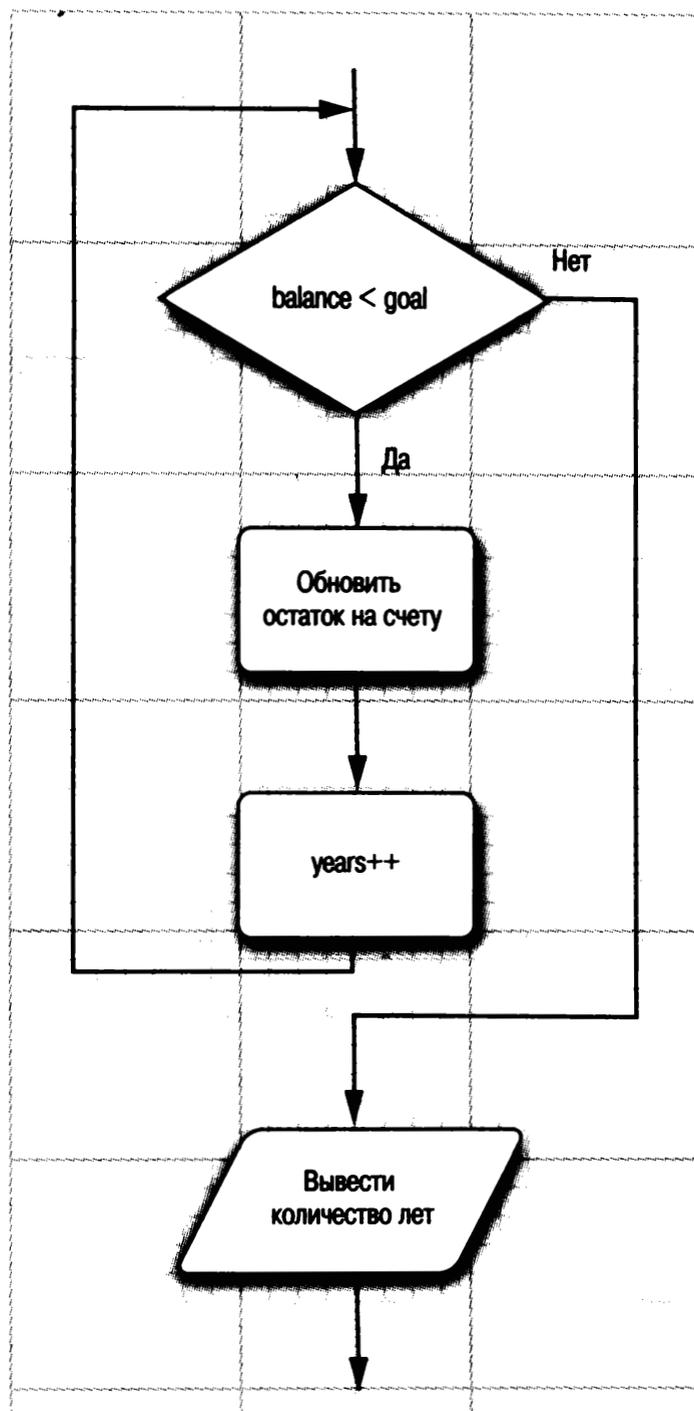


Рис. 3.10. Блок-схема, иллюстрирующая принцип действия оператора цикла while

Условие проверяется лишь после выполнения оператора, а зачастую блока операторов, в теле данного цикла. Затем цикл повторяется, снова проверяет условие и т.д. Например, программа, исходный код которой приведен в листинге 3.4, вычисляет новый остаток на пенсионном счету работника, а затем запрашивает, не собирается ли он на заслуженный отдых:

```
do
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    year++;
}
```

```
// вывести текущий остаток на счету  
.  
.  
// запросить готовность работника выйти на пенсию и получить ответ  
.  
.  
}  
while (input.equals("N"));
```

Цикл повторяется до тех пор, пока не будет получен положительный ответ "Y" (рис. 3.11). Эта программа служит характерным примером применения циклов, которые нужно выполнить хотя бы один раз, поскольку ее пользователь должен увидеть остаток на своем пенсионном счету, прежде чем решать, хватит ли ему средств на жизнь после выхода на пенсию.

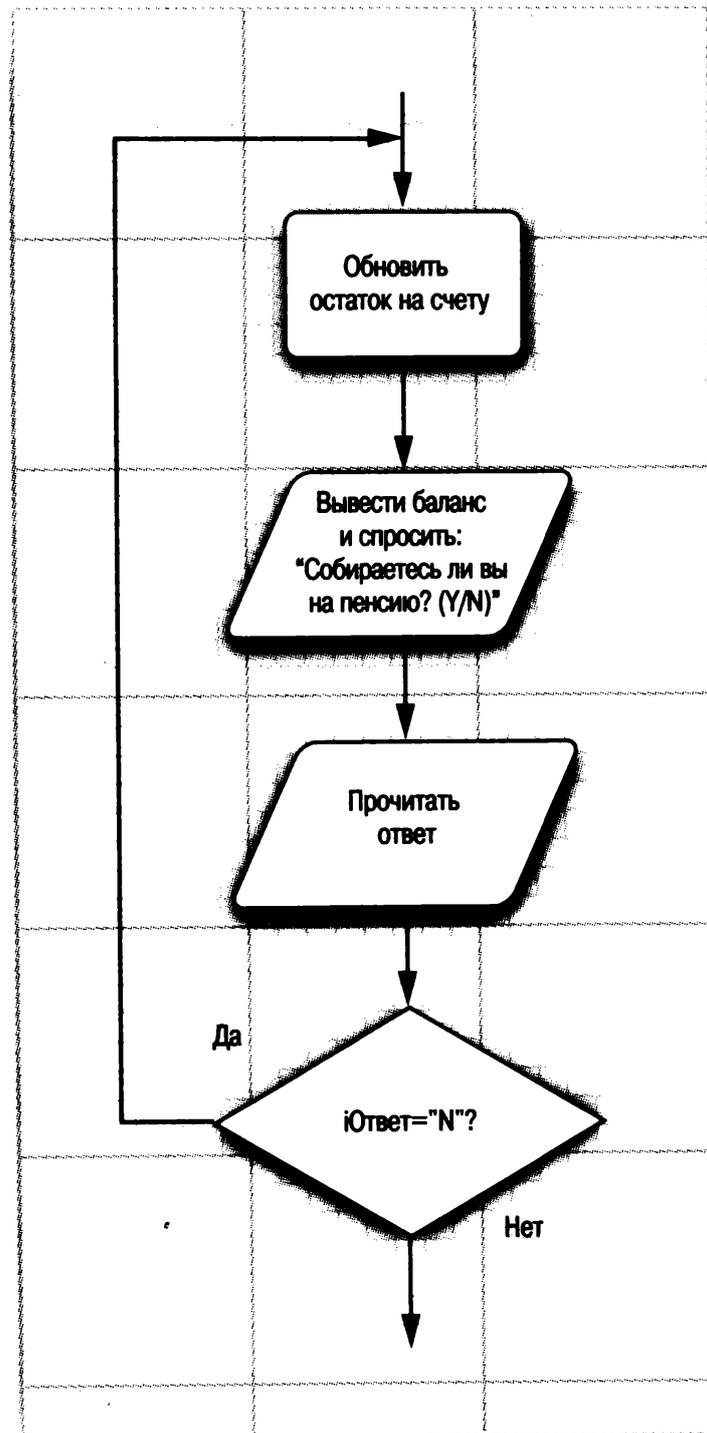


Рис. 3.11. Блок-схема, поясняющая принцип действия оператора do-while

Листинг 3.3. Исходный код из файла Retirement/Retirement.java

```
1 import java.util.*;
2
3 /**
4  * В этой программе демонстрируется применение цикла while
5  * @version 1.20 2004-02-10
6  * @author Cay Horstmann
7  */
8 public class Retirement
9 {
10  public static void main(String[] args)
11  {
12      // прочитать вводимые данные
13      Scanner in = new Scanner(System.in);
14
15      System.out.print("How much money do you need to retire? ");
16      double goal = in.nextDouble();
17
18      System.out.print("How much money will you contribute every year? ");
19      double payment = in.nextDouble();
20
21      System.out.print("Interest rate in %: ");
22      double interestRate = in.nextDouble();
23
24      double balance = 0;
25      int years = 0;
26
27      // обновить остаток на счету, пока не достигнута заданная сумма
28      while (balance < goal)
29      {
30          // добавить ежегодный взнос и проценты
31          balance += payment;
32          double interest = balance * interestRate / 100;
33          balance += interest;
34          years++;
35      }
36
37      System.out.println("You can retire in " + years + " years.");
38  }
39 }
```

Листинг 3.4. Исходный код из файла Retirement2/Retirement2.java

```
1 import java.util.*;
2
3 /**
4  * В этой программе демонстрируется применение цикла do-while
5  * @version 1.20 2004-02-10
6  * @author Cay Horstmann
7  */
8 public class Retirement2
9 {
10  public static void main(String[] args)
11  {
12      Scanner in = new Scanner(System.in);
```

```
13
14 System.out.print("How much money will you contribute every year? ");
15 double payment = in.nextDouble();
16
17 System.out.print("Interest rate in %: ");
18 double interestRate = in.nextDouble();
19
20 double balance = 0;
21 int year = 0;
22
23 String input;
24
25 // обновить остаток на счету, пока работник не готов выйти на пенсию
26 do
27 {
28     // добавить ежегодный взнос и проценты
29     balance += payment;
30     double interest = balance * interestRate / 100;
31     balance += interest;
32
33     year++;
34
35     // вывести текущий остаток на счету
36     System.out.printf(
37         "After year %d, your balance is %, .2f%n", year, balance);
38
39     // запросить готовность работника выйти на пенсию и получить ответ
40     System.out.print("Ready to retire? (Y/N) ");
41     input = in.next();
42 }
43 while (input.equals("N"));
44 }
45 }
```

Определенные циклы

Цикл `for` является весьма распространенной языковой конструкцией. В нем количество повторений находится под управлением переменной, выполняющей роль счетчика и обновляемой на каждом шаге цикла. В приведенном ниже примере цикла `for` на экран выводятся числа от 1 до 10. А порядок выполнения этого цикла наглядно показан на рис. 3.12.

```
for (int i = 1; i <= 10; i++)
    System.out.println(i);
```

В первой части оператора `for` обычно выполняется инициализация счетчика, во второй его части задается условие выполнения тела цикла, а в третьей — порядок обновления счетчика.

Несмотря на то что в Java, как и в C++, отдельными частями оператора цикла `for` могут быть практически любые выражения, существуют неписанные правила, согласно которым все три части оператора цикла `for` должны только инициализировать, проверять и обновлять один и тот же счетчик. Если не придерживаться этих правил, полученный код станет неудобным, а то и вообще непригодным для чтения.

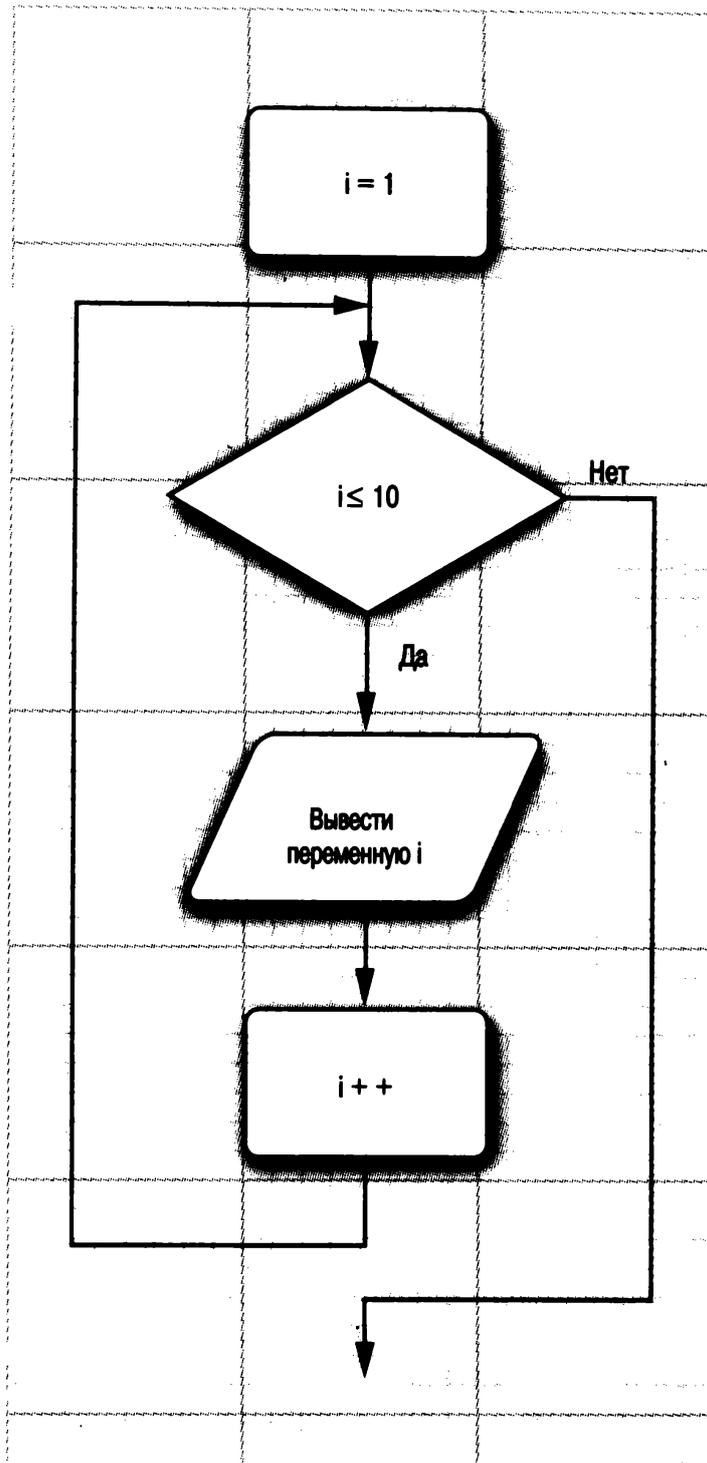


Рис. 3.12. Блок-схема, поясняющая порядок выполнения оператора цикла `for`

Подобные правила не слишком сковывают инициативу программирующего. Даже если придерживаться их, с помощью оператора `for` можно сделать немало, например, реализовать цикл с обратным отсчетом шагов:

```

for (int i = 10; i > 0; i--)
    System.out.println("Counting down . . . " + i);
    System.out.println("Blastoff!");
  
```



ВНИМАНИЕ! Будьте внимательны, проверяя в цикле равенство двух чисел с плавающей точкой. Так, приведенный ниже цикл может вообще не завершиться.

```

for (double x = 0; x != 10; x += 0.1) ...
  
```

Из-за ошибок округления окончательный результат никогда не будет достигнут. В частности, переменная x изменит свое значение с 9.999999999999998 сразу на 10.099999999999998 , потому что для числа 0.1 не существует точного двоичного представления.

При объявлении переменной в первой части оператора `for` ее область действия простирается до конца тела цикла, как показано ниже.

```
for (int i = 1; i <= 10; i++)
{
    ...
}
// здесь переменная i уже не определена
```

В частности, если переменная определена в операторе цикла `for`, ее нельзя использовать за пределами этого цикла. Следовательно, если требуется использовать конечное значение счетчика за пределами цикла `for`, соответствующую переменную следует объявить до начала цикла! Ниже показано, как это делается.

```
int i;
for (i = 1; i <= 10; i++)
{
    ...
} // здесь переменная i по-прежнему доступна
```

С другой стороны, можно объявлять переменные, имеющие одинаковое имя в разных циклах `for`, как следует из приведенного ниже примера кода.

```
for (int i = 1; i <= 10; i++)
{
    ...
}
...
for (int i = 11; i <= 20; i++) // переопределение переменной i допустимо
{
    ...
}
```

Цикл `for` является сокращенным и более удобным вариантом цикла `while`. Например, следующий фрагмент кода:

```
for (int i = 10; i > 0; i--)
    System.out.println("Counting down . . . " + i);
```

можно переписать так, как показано ниже. И оба фрагмента кода будут равнозначны.

```
int i = 10;
while (i > 0)
{
    System.out.println("Counting down . . . " + i);
    i--;
}
```

Типичный пример применения оператора цикла `for` в коде приведен в листинге 3.5. Данная программа вычисляет вероятность выигрыша в лотерее. Так, если нужно угадать 6 номеров из 50 , количество возможных вариантов будет равно $(50 \times 49 \times 48 \times 47 \times 46 \times 45) / (1 \times 2 \times 3 \times 4 \times 5 \times 6)$, поэтому шансы на выигрыш равны 1 из 15890700 . Желаем удачи! Вообще говоря, если требуется угадать k номеров из n , количество возможных вариантов определяется следующей формулой:

$$(n \times (n - 1) \times (n - 2) \times \dots \times (n - k + 1) / (1 \times 2 \times 3 \times \dots \times k))$$

Шансы на выигрыш по этой формуле рассчитываются с помощью следующего цикла `for`:

```
int lotteryOdds = 1;
for (int i = 1; i <= k; i++)
    lotteryOdds = lotteryOdds * (n - i + 1) / i;
```



НА ЗАМЕТКУ! В конце этой главы будет рассмотрен обобщенный цикл `for`, называемый также циклом в стиле `for each`. Эта языковая конструкция была введена в версии Java SE 5.0.

Листинг 3.5. Исходный код из файла LotteryOdds/LotteryOdds.java

```
1 import java.util.*;
2 /**
3  * В этой программе демонстрируется применение цикла for
4  * @version 1.20 2004-02-10
5  * @author Cay Horstmann
6  */
7 public class LotteryOdds
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12
13        System.out.print("How many numbers do you need to draw? ");
14        int k = in.nextInt();
15
16        System.out.print("What is the highest number you can draw? ");
17        int n = in.nextInt();
18        /*
19         * вычислить биномиальный коэффициент по формуле:
20         * n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
21         */
22        int lotteryOdds = 1;
23        for (int i = 1; i <= k; i++)
24            lotteryOdds = lotteryOdds * (n - i + 1) / i;
25
26        System.out.println(
27            "Your odds are 1 in " + lotteryOdds + ". Good luck!");
28    }
29 }
```

Оператор `switch` для многовариантного выбора

Языковая конструкция `if/else` может оказаться неудобной, если требуется организовать в коде выбор из многих вариантов. Для этой цели в Java имеется оператор `switch`, полностью соответствующий одноименному оператору в C и C++. Например, организовав выбор из четырех альтернативных вариантов (рис. 3.13), можно написать следующий код:

```
Scanner in = new Scanner(System.in);
System.out.print("Select an option (1, 2, 3, 4) ");
int choice = in.nextInt();
```

```
switch (choice)
{
    case 1:
        . . .
        break;
    case 2:
        . . .
        break;
    case 3:
        . . .
        break;
    case 4:
        . . .
        break;
    default:
        // неверный ввод
        . . .
        break;
}
```

Выполнение начинается с метки ветви `case`, соответствующей значению 1 переменной `choice`, и продолжается до очередного оператора `break` или конца оператора `switch`. Если ни одна из меток ветвей `case` не совпадает со значением переменной, выполняется выражение по метке ветви `default` (если таковое предусмотрено).



ВНИМАНИЕ! Если не ввести оператор `break` в конце ветви `case`, то возможно последовательное выполнение кода по нескольким ветвям `case`. Очевидно, что такая ситуация чревата ошибками, поэтому мы никогда не пользуемся оператором `switch` в своих программах.

Если же вы предпочитаете пользоваться оператором `switch` в своих программах, рассмотрите возможность компиляции их кода с параметром `-Xlint:fallthrough`, как показано ниже.

```
javac -Xlint:fallthrough Test.java
```

В этом случае компилятор выдаст предупреждающее сообщение, если альтернативный выбор не завершается оператором `break`.

А если требуется последовательное выполнение кода по нескольким ветвям `case`, следует поместить охватывающий метод аннотацией `@SuppressWarnings("fallthrough")`. В таком случае никаких предупреждений для данного метода не выдается. (Аннотация служит механизмом для предоставления сведений компилятору или инструментальному средству, обрабатывающему файлы с исходным кодом Java или классами. Более подробно аннотации будут рассматриваться в главе 13 второго тома данной книги.)

В качестве метки ветви `case` может быть следующее.

- Константное выражение типа `char`, `byte`, `short`, `int` (или соответствующие классы-оболочки `Character`, `Byte`, `Short` и `Integer`, которые будут представлены в главе 4).
- Перечисляемая константа.
- Строковый литерал, начиная с версии Java SE 7.

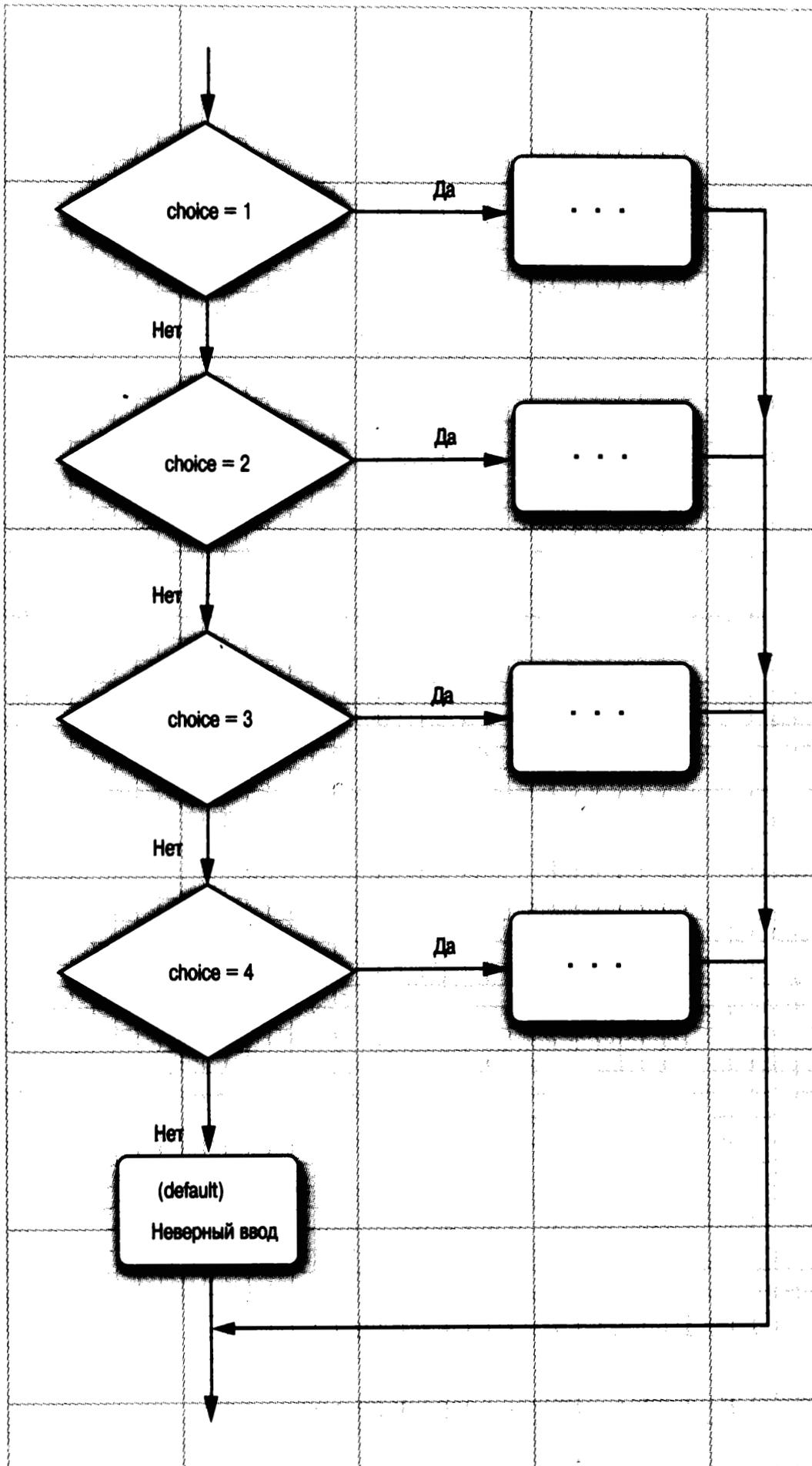


Рис. 3.13. Блок-схема, поясняющая принцип действия оператора switch

Так, в приведенном ниже фрагменте кода указан строковый литерал в ветви case.

```
String input = . . . ;
switch (input.toLowerCase())
{
    case "yes": // допустимо, начиная с версии Java SE 7
        . . .
        break;
    . . .
}
```

Когда оператор switch употребляется в коде с перечислимыми константами, указывать имя перечисления в метке каждой ветви не нужно, поскольку оно выводится из значения переменной оператора switch. Например:

```
Size sz = . . . ;
switch (sz)
{
    case SMALL: // не нужно использовать имя перечисления Size.SMALL
        . . .
        break;
    . . .
}
```

Операторы прерывания логики управления программой

Несмотря на то что создатели Java сохранили зарезервированное слово goto, они решили не включать его в состав языка. В принципе применение операторов goto считается признаком плохого стиля программирования. Некоторые программисты считают, что борьба с использованием оператора goto ведется недостаточно активно (см., например, известную статью Дональда Кнута “Структурное программирование с помощью операторов goto” (Structured Programming with goto statements; http://pplab.snu.ac.kr/courses/adv_p105/papers/p261-knuth.pdf)). Они считают, что применение операторов goto может приводить к ошибкам. Но в некоторых случаях нужно выполнять преждевременный выход из цикла. Создатели Java согласились с их аргументами и даже добавили в язык новый оператор для поддержки такого стиля программирования. Им стал оператор break с меткой.

Рассмотрим обычный оператор break без метки. Для выхода из цикла можно применять тот же самый оператор break, что и для выхода из оператора switch. Ниже приведен пример применения оператора break без метки.

```
while (years <= 100)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    if (balance >= goal) break;
    years++;
}
```

В данном фрагменте кода выход из цикла осуществляется при выполнении одного из двух условий: years > 100 в начале цикла или balance >= goal в теле цикла. Разумеется, то же самое значение переменной years можно было бы вычислить и без применения оператора break, как показано ниже.

```
while (years <= 100 && balance < goal)
{
    balance += payment;
```

```

double interest = balance * interestRate / 100;
balance += interest;
if (balance < goal)
    years++;
}

```

Следует, однако, иметь в виду, что проверка условия `balance < goal` в данном варианте кода повторяется дважды. Этого позволяет избежать оператор `break`.

В отличие от C++, в Java поддерживается оператор `break` с меткой, обеспечивающий выход из вложенных циклов. С его помощью можно организовать прерывание глубоко вложенных циклов при нарушении логики управления программой. А задавать дополнительные условия для проверки каждого вложенного цикла попросту неудобно.

Ниже приведен пример, демонстрирующий применение оператора `break` с меткой в коде. Следует иметь в виду, что метка должна предшествовать тому внешнему циклу, из которого требуется выйти. Кроме того, метка должна оканчиваться двоеточием.

```

Scanner in = new Scanner(System.in);
int n;
read_data:
while ( . . . ) // Этот цикл помечен меткой
{
    . . .
    for ( . . . ) // Этот цикл не помечен
    {
        System.out.print("Enter a number >= 0: ");
        n = in.nextInt();
        if (n < 0) // условие для прерывания цикла
            break read_data;
        // прервать цикл
        . . .
    }
}
// этот оператор выполняется сразу же после
// оператора break с меткой
if (n < 0) // проверить наличие недопустимой ситуации
{
    // принять меры против недопустимой ситуации
}
else
{
    // выполнить действия при нормальном ходе выполнения программы
}
}

```

Если было введено неверное число, оператор `break` с меткой выполнит переход в конец помеченного блока. В этом случае необходимо проверить, нормально ли осуществлен выход из цикла, или он произошел в результате выполнения оператора `break`.



НА ЗАМЕТКУ! Любопытно, что метку можно связать с любым оператором — даже с условным оператором `if` или блоком, как показано ниже.

```

метка:
{
    . . .
}

```

```
if (условие) break метка; // выход из блока
...
}
```

// При выполнении оператора break управление передается в эту точку

Итак, если вам крайне необходим оператор `goto`, поместите блок, из которого нужно немедленно выйти, непосредственно перед тем местом, куда требуется перейти, и примените оператор `break`! Естественно, такой способ не рекомендуется применять на практике. Следует также иметь в виду, что подобным способом можно выйти из блока, но невозможно войти в него.

Существует также оператор `continue`, который, подобно оператору `break`, прерывает нормальный ход выполнения программы. Оператор `continue` передает управление в начало текущего вложенного цикла. Ниже приведен характерный пример применения данного оператора.

```
Scanner in = new Scanner(System.in);
while (sum < goal)
{
    System.out.print("Enter a number: ");
    n = in.nextInt();
    if (n < 0) continue;
    sum += n; // не выполняется, если n < 0
}
```

Если $n < 0$, то оператор `continue` выполняет переход в начало цикла, пропуская оставшуюся часть текущего шага цикла. Если же оператор `continue` применяется в цикле `for`, он передает управление оператору увеличения счетчика цикла. В качестве примера рассмотрим следующий цикл:

```
for (count = 1; count <= 100; count++)
{
    System.out.print("Enter a number, -1 to quit: ");
    n = in.nextInt();
    if (n < 0) continue;
    sum += n; // не выполняется, если n < 0
}
```

Если $n < 0$, то оператор `continue` осуществит переход к оператору `count++`. В Java имеется также оператор `continue` с меткой, передающий управление заголовку оператора цикла, помеченного соответствующей меткой.



СОВЕТ. Многие программирующие на Java считают, что операторы `break` и `continue` неоправданно усложняют текст программы. Применять эти операторы совсем не обязательно, поскольку те же самые действия можно реализовать, не прибегая к ним. В данной книге операторы `break` и `continue` не употребляются нигде, кроме примеров кода, приведенных в этом разделе.

Большие числа

Если для решения задачи недостаточно точности основных типов для представления целых и вещественных чисел, можно обратиться к классам `BigInteger` и `BigDecimal` из пакета `java.math`. Эти классы предназначены для выполнения действий с числами, состоящими из произвольного количества цифр. В классах

`BigInteger` и `BigDecimal` реализуются арифметические операции произвольной точности соответственно для целых и вещественных чисел.

Для преобразования обычного числа в число с произвольной точностью (называемое также *большим числом*) служит статический метод `valueOf()`:

```
BigInteger a = BigInteger.valueOf(100);
```

К сожалению, над большими числами нельзя выполнять обычные математические операции вроде `+` или `*`. Вместо этого следует применять методы `add()` и `multiply()` из классов соответствующих типов больших чисел, как в приведенном ниже примере кода.

```
BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.valueOf(2))); // d = c * (b + 2)
```



НА ЗАМЕТКУ C++! В отличие от C++, перегрузка операций в Java не поддерживается. Поэтому разработчики класса `BigInteger` были лишены возможности переопределить операции `+` и `*` для методов `add()` и `multiply()`, чтобы реализовать в классе `BigInteger` аналоги операций сложения и умножения соответственно. Создатели Java реализовали только перегрузку операции `+` для обозначения сцепления строк. Они решили не перегружать остальные операции и не предоставили такой возможности программирующим на Java в их собственных классах.

В листинге 3.6 приведен видоизмененный вариант программы из листинга 3.5 для подсчета шансов на выигрыш в лотерею. Теперь эта программа может обращаться с большими числами. Так, если вам предложат сыграть в лотерею, в которой нужно угадать 60 чисел из 490 возможных, эта программа сообщит, что ваши шансы на выигрыш составляют 1 из 716395843461995557415116222540092933411717612789263493493351013459481104668848. Желаем удачи!

Программа из листинга 3.5 вычисляла следующее выражение:

```
lotteryOdds = lottery * (n - i + 1) / i;
```

Для обработки больших чисел соответствующая строка кода будет выглядеть следующим образом:

```
lotteryOdds =
    lotteryOdds.multiply(BigInteger.valueOf(n-i+1).divide(BigInteger.valueOf(i)));
```

Листинг 3.6. Исходный код из файла `BigIntegerTest/BigIntegerTest.java`

```
1 import java.math.*;
2 import java.util.*;
3
4 /**
5  * В этой программе используются большие числа для
6  * оценки шансов на выигрыш в лотерею
7  * @version 1.20 2004-02-10
8  * @author Cay Horstmann
9  */
10 public class BigIntegerTest
11 {
12     public static void main(String[] args)
13     {
14         Scanner in = new Scanner(System.in);
15     }
```

```
16 System.out.print("How many numbers do you need to draw? ");
17 int k = in.nextInt();
18
19 System.out.print("What is the highest number you can draw? ");
20 int n = in.nextInt();
21
22 /*
23  * ВЫЧИСЛИТЬ БИНОМИАЛЬНЫЙ КОЭФФИЦИЕНТ ПО ФОРМУЛЕ:
24  *  $n * (n-1) * (n-2) * \dots * (n-k+1) / (1 * 2 * 3 * \dots * k)$ 
25  *
26  */
27
28 BigInteger lotteryOdds = BigInteger.valueOf(1);
29
30 for (int i = 1; i <= k; i++)
31     lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(
32         n - i + 1)).divide(BigInteger.valueOf(i));
33
34 System.out.println("Your odds are 1 in " +
35     lotteryOdds + ". Good luck!");
36 }
37 }
```

java.math.BigInteger 1.1

- `BigInteger subtract(BigInteger other)`
- `BigInteger multiply(BigInteger other)`
- `BigInteger divide(BigInteger other)`
- `BigInteger mod(BigInteger other)`

Возвращают сумму, разность, произведение, частное и остаток от деления, полученные в результате выполнения соответствующих операций над текущим большим числом и значением переменной *other*.

- `int compareTo(BigInteger other)`

Возвращает 0, если текущее большое число равно значению переменной *other*, отрицательное число, если это большое число меньше значения переменной *other*, а иначе — положительное число.

- `static BigInteger valueOf(long x)`

Возвращает большое число, равное значению переменной *x*.

java.math.BigDecimal 1.1

- `BigDecimal add(BigDecimal other)`
- `BigDecimal subtract(BigDecimal other)`
- `BigDecimal multiply(BigDecimal other)`

- `BigDecimal divide(BigDecimal other, int roundingMode)` 5.0
Возвращают сумму, разность, произведение и частное, полученные в результате соответствующих операций над текущим большим числом и значением переменной *other*. Чтобы вычислить частное, следует указать режим округления. Режим `roundingMode.ROUND_HALF_UP` означает округление в сторону уменьшения для цифр 0–4 и в сторону увеличения для цифр 5–9. Для обычных вычислений этого оказывается достаточно. Другие режимы округления описаны в документации.
- `int compareTo(BigDecimal other)`
Возвращает 0, если текущее число типа `BigDecimal` равно значению переменной *other*, отрицательное число, если это число меньше значения переменной *other*, а иначе — положительное число.
- `static BigDecimal valueOf(long x)`
- `static BigDecimal valueOf(long x, int scale)`
Возвращают большое десятичное число, значение которого равно значению переменной *x* или $x/10^{\text{scale}}$.

Массивы

Массив — это структура данных, в которой хранятся величины одинакового типа. Доступ к отдельному элементу массива осуществляется по целочисленному *индексу*. Так, если *a* — массив целых чисел, то `a[i]` — *i*-е целое число в массиве. Массив объявляется следующим образом: сначала указывается тип массива, т.е. тип элементов, содержащихся в нем, затем следует пара пустых квадратных скобок, а после них — имя переменной. Ниже приведено объявление массива, состоящего из целых чисел.

```
int[] a;
```

Но этот оператор лишь объявляет переменную *a*, не инициализируя ее. Чтобы создать массив, нужно выполнить операцию `new`, как показано ниже.

```
int[] a = new int[100];
```

В этой строке кода создается массив, состоящий из **100** целых чисел. Длина массива не обязательно должна быть постоянной. Так, операция `new int[n]` создает массив с длиной *n*.



НА ЗАМЕТКУ! Объявить массив можно двумя способами:

```
int[] a;
```

или

```
int a[];
```

Большинство программистов на Java пользуются первым способом, поскольку в этом случае тип более явно отделяется от имени переменной.

Элементы сформированного выше массива нумеруются от 0 до 99 (а не от 1 до 100). После создания массива его можно заполнить конкретными значениями. В частности, это можно делать в цикле, как показано ниже.

```
int[] a = new int[100];
for (int i = 0; i < 100; i++)
    a[i] = i; // заполнить массив числами от 0 до 99
```

При создании массива чисел все его элементы инициализируются нулями. Массивы значений типа `boolean` инициализируются логическим значением `false`, а массивы объектов — пустым значением `null`, указывающим на то, что массив пока еще не содержит ни одного объекта. Для начинающих это может показаться неожиданным. Так, в приведенной ниже строке кода создается массив из десяти символьных строк, причем все они нулевые, т.е. имеют значение `null`.

```
String[] names = new String[10];
```

Если же требуется создать массив из пустых символьных строк, его придется специально заполнить пустыми строками, как показано ниже.

```
for (int i = 0; i < 10; i++) names[i] = "";
```



ВНИМАНИЕ! Если, создав массив, состоящий из 100 элементов, вы попытаетесь обратиться к элементу `a[100]` (или любому другому элементу, индекс которого выходит за пределы от 0 до 99), программа прекратит работу, поскольку будет сгенерировано исключение в связи с выходом индекса массива за допустимые пределы.

Для определения количества элементов в массиве служит свойство *имя_массива*. `length`. Например, в следующем фрагменте кода свойство `length` используется для вывода его элементов массива:

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

После создания массива изменить его размер нельзя (хотя можно, конечно, изменять его отдельные элементы). Если в ходе выполнения программы требуется часто изменять размер массива, то для этой цели лучше использовать другую структуру данных, называемую *списочным массивом*. (Подробнее о списочных массивах речь пойдет в главе 5.)

Цикл в стиле `for each`

В Java имеется эффективный вид цикла, позволяющий перебирать все элементы массива (а также любого другого набора данных), не применяя счетчик. Эта усовершенствованная разновидность цикла `for` записывается следующим образом:

```
for (переменная : коллекция) оператор
```

При выполнении этого цикла его *переменной* последовательно присваивается каждый элемент *коллекции*, после чего выполняется *оператор* (или блок). В качестве *коллекции* может использоваться массив или экземпляр класса, реализующего интерфейс `Iterable`, например `ArrayList`. Списочные массивы типа `ArrayList` будут обсуждаться в главе 5, а интерфейс `Iterable` — в главе 1 второго тома данной книги.

В приведенном ниже примере рассматриваемый здесь цикл организуется для вывода каждого элемента массива `a` в отдельной строке.

```
for (int element : a)
    System.out.println(element);
```

Действие этого цикла можно кратко описать как обработку каждого элемента из `a`. Создатели Java рассматривали возможность применения ключевых слов `foreach` и `in` для обозначения данного типа цикла. Но такой тип цикла появился намного позже основных языковых средств Java, и введение нового ключевого слова привело бы к необходимости изменять исходный код некоторых уже существовавших приложений, содержащих переменные или методы с подобными именами, как, например, `System.in`.

Конечно, действия, выполняемые с помощью этого типа цикла, можно осуществить и средствами традиционного цикла `for`:

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

Но при использовании цикла в стиле `for each` запись получается более краткой, поскольку отпадает необходимость в начальном выражении и условии завершения цикла. Да и вероятность ошибок при написании кода уменьшается.



НА ЗАМЕТКУ! Переменная цикла в стиле `for each` перебирает элементы массива, а не значения индекса.

Несмотря на появление усовершенствованного цикла в стиле `for each`, упрощающего во многих случаях составление программ, традиционный цикл `for` совсем не устарел. Без него нельзя обойтись, например, в тех случаях, когда требуется обработать не всю коллекцию, а лишь ее часть, или тогда, когда счетчик явно используется в теле цикла.



СОВЕТ. Еще более простой способ вывести все значения из массива состоит в использовании метода `toString()` из класса `Arrays`. Так, в результате вызова `Arrays.toString(a)` будет возвращена символьная строка, содержащая все элементы массива, заключенные в квадратные скобки и разделенные запятыми, например: `"[2, 3, 5, 7, 11, 13]"`. Следовательно, чтобы вывести массив, достаточно сделать вызов `System.out.println(Arrays.toString(a));`.

Инициализация массивов и анонимные массивы

В Java имеется средство для одновременного создания массива и его инициализации. Пример такой синтаксической конструкции приведен ниже.

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13};
```

Обратите внимание на то, что в данном случае операция `new` не требуется. Можно даже инициализировать массив, не имеющий имени или называемый иначе *анонимным*, как показано ниже.

```
new int[] {16, 19, 23, 29, 31, 37}
```

В этом выражении выделяется память для нового массива, а его элементы заполняются числами, указанными в фигурных скобках. При этом подсчитывается их

количество и соответственно определяется размер массива. Такую синтаксическую конструкцию удобно применять для повторной инициализации массива без необходимости создавать новую переменную. Например, выражение

```
smallPrimes = new int{ 17, 19, 23, 29, 31, 37 };
```

представляет собой сокращенную запись следующего фрагмента кода:

```
int[] anonymous = { 17, 19, 23, 29, 31, 37 };  
smallPrimes = anonymous;
```



НА ЗАМЕТКУ! При необходимости можно создать массив нулевого размера. Такой массив может оказаться полезным при написании метода, возвращающего массив, который оказывается в некоторых случаях пустым. Массив нулевой длины объявляется следующим образом:

```
new тип_элементов[0]
```

Следует, однако, иметь в виду, что массив нулевой длины не равнозначен массиву пустых объектов со значениями `null`. (Более подробно этот вопрос будет обсуждаться в главе 4.)

Копирование массивов

При необходимости одну переменную массива можно скопировать в другую, но в этом случае обе переменные будут ссылаться на один и тот же массив:

```
int[] luckyNumbers = smallPrimes;  
luckyNumbers[5] = 12; // теперь элемент smallPrimes[5] также равен 12
```

Результат копирования переменной массива приведен на рис. 3.14.

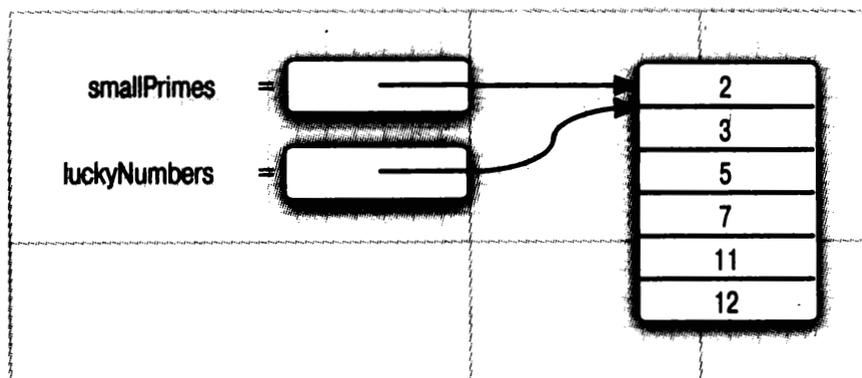


Рис. 3.14. Копирование переменной массива

Если требуется скопировать все элементы одного массива в другой, для этого следует вызвать метод `copyTo()` из класса `Arrays`. Его вызов выглядит следующим образом:

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length);
```

Вторым параметром является длина нового массива. Обычно этот метод применяется для увеличения размера массива следующим образом:

```
luckyNumbers = Arrays.copyOf(luckyNumbers, 2 * luckyNumbers.length);
```

Дополнительные элементы заполняются нулями, если массив содержит числа, либо логическими значениями `false`, если значения типа `boolean`. С другой стороны, если длина нового массива меньше длины исходного, то копируются только начальные элементы.



НА ЗАМЕТКУ C++! Массив в Java значительно отличается от массива в C++, располагающегося в стеке. Но переменную массива можно условно сравнить с указателем на динамически созданный массив. Таким образом, выражение в Java

```
int[] a = new int[100]; //Java
```

можно сравнить со следующим выражением в C++:

```
int* a = new int[100]; // C++
```

но оно существенно отличается от приведенного ниже.

```
int a[100]; // C++
```

В Java при выполнении операции [] по умолчанию проверяются границы массива. Кроме того, в Java не поддерживается арифметика указателей. В частности, нельзя увеличить указатель на массив `a`, чтобы обратиться к следующему элементу этого массива.

Параметры командной строки

В каждом из рассмотренных ранее примеров программ на Java присутствовал метод `main()` с параметром `String[] args`. Этот параметр означает, что метод `main()` получает массив, элементами которого являются параметры, указанные в командной строке. Рассмотрим в качестве примера следующую программу:

```
public class Message
{
    public static void main(String[] args)
    {
        if (args[0].equals("-h"))
            System.out.print("Hello, ");
        else if (args[0].equals("-g"))
            System.out.print("Goodbye, ");
        // вывести остальные параметры командной строки
        for (int i = 1; i < args.length; i++)
            System.out.print(" " + args[i]);
        System.out.println("!");
    }
}
```

При следующем вызове программы из командной строки:

```
java Message -g cruel world
```

массив `args` будет состоять из таких элементов:

```
args[0]: "-g"
args[1]: "cruel"
args[2]: "world"
```

В результате выполнения данной программы будет выведено следующее сообщение:

```
Goodbye, cruel world!
(Прощай, жестокий мир!)
```



НА ЗАМЕТКУ C++! При запуске программы на Java ее имя не сохраняется в массиве `args`, передаваемом методу `main()`. Так, после запуска из командной строки программы **Message** по команде **java Message -h world** элемент `args[0]` будет содержать параметр `"-h"`, а не название программы `"Message"` или команду `"java"`.

Сортировка массива

Если требуется упорядочить массив чисел, для этого достаточно вызвать метод `sort()` из класса `Arrays`:

```
int[] a = new int[10000];
...
Arrays.sort(a);
```

В этом методе используется усовершенствованный вариант алгоритма быстрой сортировки, которая считается наиболее эффективной для большинства наборов данных. Класс `Arrays` содержит ряд удобных методов, предназначенных для работы с массивами. Эти методы приведены в конце раздела.

Программа, исходный код которой представлен в листинге 3.7, создает массив и генерирует случайную комбинацию чисел для лотереи. Так, если нужно угадать 6 чисел из 49, программа может вывести следующее сообщение:

```
Bet the following combination. It'll make you rich!
(Попробуйте следующую комбинацию, чтобы разбогатеть!)
4
7
8
19
30
44
```

Для выбора случайных чисел массив `numbers` сначала заполняется последовательностью чисел 1, 2, ..., n , как показано ниже.

```
int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
    numbers[i] = i + 1;
```

Второй массив служит для хранения сгенерированных чисел:

```
int[] result = new int[k];
```

Затем генерируется k чисел. Метод `Math.random()` возвращает случайное число с плавающей точкой, находящееся в пределах от 0 (включительно) до 1 (исключительно). Умножение результата на число n дает случайное число, находящееся в пределах от 0 до $n-1$, как показано в следующей строке кода:

```
int r = (int) (Math.random() * n);
```

Далее i -е число присваивается i -му элементу массива. Сначала там будет находиться результат $r+1$, но, как будет показано ниже, содержимое массива `number` будет изменяться после генерирования каждого нового числа.

```
result[i] = numbers[r];
```

Теперь следует убедиться, что ни одно число не повторится, т.е. все номера должны быть разными. Следовательно, нужно сохранить в элементе массива `number[r]` последнее число, содержащееся в массиве, и уменьшить n на единицу, как показано ниже.

```
numbers[r] = numbers[n - 1];
n--;
```

Обратите внимание на то, что всякий раз при генерировании чисел получается индекс, а не само число. Это индекс массива, содержащего числа, которые еще не

были выбраны. После генерирования k номеров лотереи сформированный в итоге массив `result` сортируется, чтобы результат выглядел более изящно:

```
Arrays.sort(result);
for (int i = 0; i < result.length; i++)
    System.out.println(result[i]);
```

Листинг 3.7. Исходный код из файла `LotteryDrawing/LotteryDrawing.java`

```
1 import java.util.*;
2
3 /**
4  * В этой программе демонстрируется обращение с массивами
5  * @version 1.20 2004-02-10
6  * @author Cay Horstmann
7  */
8 public class LotteryDrawing
9 {
10     public static void main(String[] args)
11     {
12         Scanner in = new Scanner(System.in);
13
14         System.out.print("How many numbers do you need to draw? ");
15         int k = in.nextInt();
16
17         System.out.print("What is the highest number you can draw? ");
18         int n = in.nextInt();
19
20         // заполнить массив числами 1 2 3 . . . n
21         int[] numbers = new int[n];
22         for (int i = 0; i < numbers.length; i++)
23             numbers[i] = i + 1;
24
25         // выбрать k номеров и поместить их во второй массив
26         int[] result = new int[k];
27         for (int i = 0; i < result.length; i++)
28         {
29             // получить случайный индекс в пределах от 0 до n - 1
30             int r = (int) (Math.random() * n);
31
32             // выбрать элемент из произвольного места
33             result[i] = numbers[r];
34
35             // переместить последний элемент в произвольное место
36             numbers[r] = numbers[n - 1];
37             n--;
38         }
39
40         // вывести отсортированный массив
41         Arrays.sort(result);
42         System.out.println(
43             "Bet the following combination. It'll make you rich!");
44         for (int r : result)
45             System.out.println(r);
46     }
47 }
```

- `static boolean equals(тип[] a, тип[] b)`

Возвращает логическое значение `true`, если массивы имеют равную длину и совпадают все их элементы по индексу.

Параметры: `a, b`

Массивы типа `int, long, short, char, byte, boolean, float` или `double`

Многомерные массивы

Для доступа к элементам многомерного массива применяется несколько индексов. Такие массивы служат для хранения таблиц и более сложных упорядоченных структур данных. Если вы не собираетесь пользоваться многомерными массивами в своей практической деятельности, то можете смело пропустить этот раздел.

Допустим, вам нужно составить таблицу чисел, показывающих, как возрастут первоначальные капиталовложения на сумму 10 тысяч долларов при разных процентных ставках, если прибыль ежегодно выплачивается и повторно вкладывается в дело. Пример таких числовых данных приведен в табл. 3.8.

Таблица 3.8. Рост капиталовложений при разных процентных ставках

10%	11%	12%	13%	14%	15%
10000,00	10000,00	10000,00	10000,00	10000,00	10000,00
11000,00	11100,00	11200,00	11300,00	11400,00	11500,00
12100,00	12321,00	12544,00	12769,00	12996,00	13225,00
13310,00	13676,31	14049,28	14428,97	14815,44	15208,75
14641,00	15180,70	15735,19	16304,74	16889,60	17490,06
16105,10	16850,58	17623,42	18424,35	19254,15	20113,57
17715,61	18704,15	19738,23	20819,52	21949,73	23130,61
19487,17	20761,60	22106,81	23526,05	25022,69	26600,20
21435,89	23045,38	24759,63	26584,44	28525,86	30590,23
23579,48	25580,37	27730,79	30040,42	32519,49	35178,76

Очевидно, что эти данные лучше всего хранить в двумерном массиве (или матрице), например, под названием `balance`. Объявить двумерный массив в Java нетрудно. Это можно, в частности, сделать следующим образом:

```
double[][] balances;
```

Как обычно, массивом нельзя пользоваться до тех пор, пока он не инициализирован с помощью операции `new`. В данном случае инициализация двумерного массива осуществляется следующим образом:

```
balances = new double[NYEARS][NRATES];
```

В других случаях, когда элементы массива известны заранее, можно пользоваться сокращенной записью для его инициализации, не прибегая к операции `new`. Ниже приведен соответствующий тому пример.

```
int[][] magicSquare =
{
    {16, 3, 2, 13},
    {5, 10, 11, 8},
    {9, 6, 7, 12},
    {4, 15, 14, 1}
};
```

После инициализации массива к его отдельным элементам можно обращаться, используя две пары квадратных скобок, например `balances[i][j]`.

В качестве примера рассмотрим программу, сохраняющую процентные ставки в одномерном массиве `interest`, а результаты подсчета остатка на счету ежегодно по каждой процентной ставке — в двумерном массиве `balance`. Сначала первая строка массива инициализируется исходной суммой следующим образом:

```
for (int j = 0; j < balances[0].length; j++)
    balances[0][j] = 10000;
```

Затем подсчитывается содержимое остальных строк, как показано ниже.

```
for (int i = 1; i < balances.length; i++)
{
    for (int j = 0; j < balances[i].length; j++)
    {
        double oldBalance = balances[i - 1][j];
        double interest = . . .;
        balances[i][j] = oldBalance + interest;
    }
}
```

Исходный код данной программы полностью приведен в листинге 3.8.



НА ЗАМЕТКУ! Цикл в стиле `for each` не обеспечивает автоматического перебора элементов двумерного массива. Он лишь перебирает строки, которые, в свою очередь, являются одномерными массивами. Так, для обработки всех элементов двумерного массива `a` требуются два следующих цикла:

```
for (double[] row : a)
    for (double value : row)
        обработать значения value
```



СОВЕТ. Чтобы вывести на скорую руку список элементов двумерного массива, сделайте вызов `System.out.println(Arrays.deepToString(a))`; вывод будет отформатирован следующим образом:

```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 1]]
```

Листинг 3.8. Исходный код из файла CompoundInterest/CompoundInterest.java

```
1  /**
2   * В этой программе демонстрируется сохранение табличных
3   * данных в двумерном массиве
4   * @version 1.40 2004-02-10
5   * @author Cay Horstmann
6   */
7  public class CompoundInterest
8  {
9      public static void main(String[] args)
10     {
11         final double STARTRATE = 10;
12         final int NRATES = 6;
13         final int NYEARS = 10;
14
15         // установить процентные ставки 10 . . . 15%
16         double[] interestRate = new double[NRATES];
17         for (int j = 0; j < interestRate.length; j++)
18             interestRate[j] = (STARTRATE + j) / 100.0;
19
20         double[][] balances = new double[NYEARS][NRATES];
21
22         // установить исходные остатки на счету равными 10000
23         for (int j = 0; j < balances[0].length; j++)
24             balances[0][j] = 10000;
25
26         // рассчитать проценты на следующие годы
27         for (int i = 1; i < balances.length; i++)
28         {
29             for (int j = 0; j < balances[i].length; j++)
30             {
31                 // получить остатки на счету за прошлый год
32                 double oldBalance = balances[i - 1][j];
33
34                 // рассчитать проценты
35                 double interest = oldBalance * interestRate[j];
36
37                 // рассчитать остатки на счету в текущем году
38                 balances[i][j] = oldBalance + interest;
39             }
40         }
41
42         // вывести один ряд процентных ставок
43         for (int j = 0; j < interestRate.length; j++)
44             System.out.printf("%9.0f%%", 100 * interestRate[j]);
45
46         System.out.println();
47         // вывести таблицу остатков на счету
48         for (double[] row : balances)
49         {
50             // вывести строку таблицы
51             for (double b : row)
52                 System.out.printf("%10.2f", b);
53
54             System.out.println();
55         }
56     }
57 }
```

Неровные массивы

Все рассмотренные ранее языковые конструкции мало чем отличались от аналогичных конструкций в других языках программирования. Но механизм массивов в Java имеет особенность, предоставляющую совершенно новые возможности. В этом языке вообще нет многомерных массивов, а только одномерные. Многомерные массивы — это искусственно создаваемые “массивы массивов”.

Например, массив `balances` из предыдущего примера программы фактически представляет собой массив, состоящий из 10 элементов, каждый из которых является массивом из 6 элементов, содержащих числа с плавающей точкой (рис. 3.15).

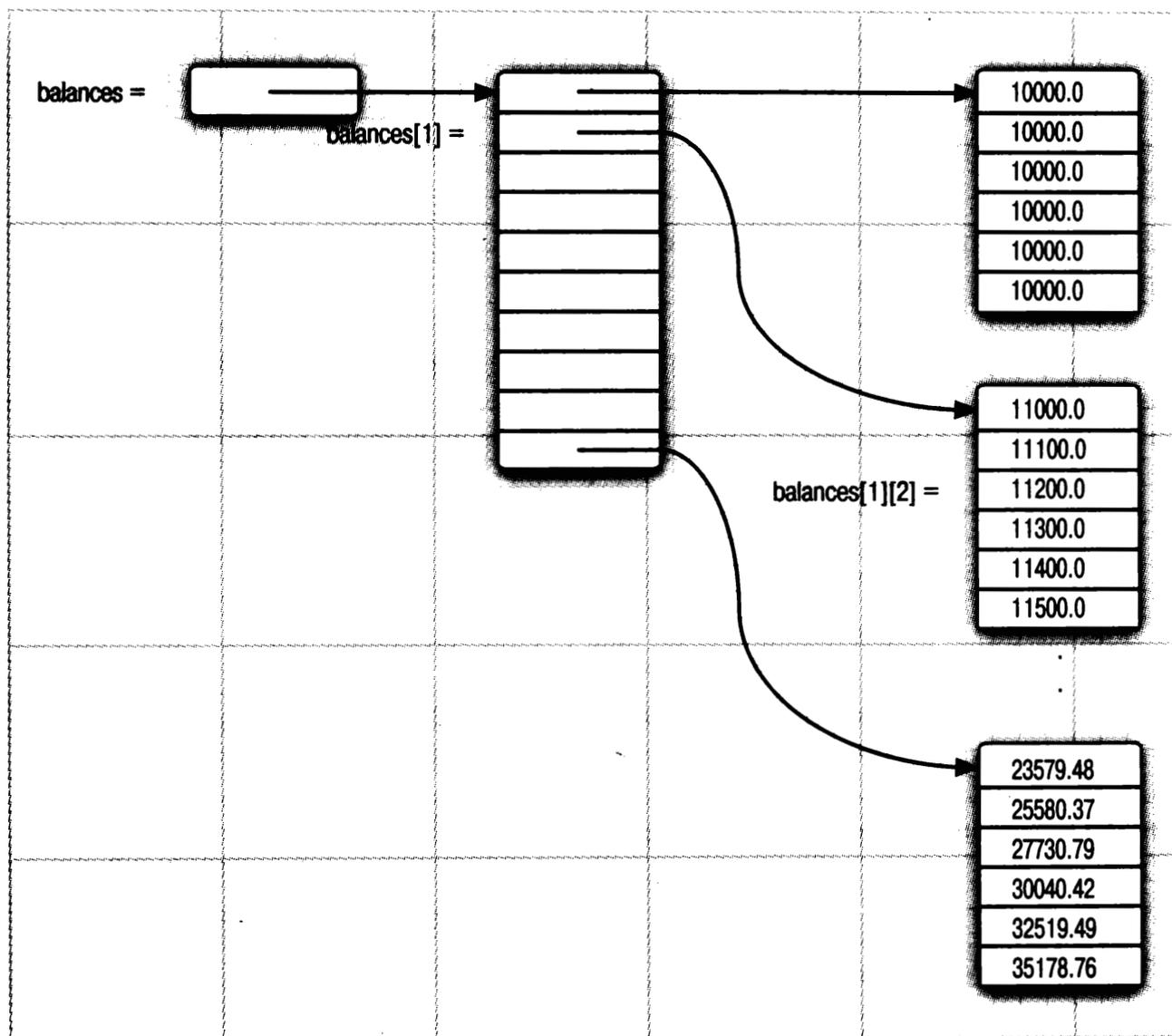


Рис. 3.15. Двумерный массив

Выражение `balance[i]` определяет i -й подмассив, т.е. i -ю строку таблицы. Эта строка сама представляет собой массив, а выражение `balance[i][j]` относится к его j -му элементу. Строки массива доступны из программы, и поэтому их, как показано ниже, можно легко переставлять!

```
double[] temp = balances[i];
balances[i] = balances[i + 1];
balances[i + 1] = temp;
```

Кроме того, в Java легко формируются неровные, “рваные”, массивы, т.е. такие массивы, у которых разные строки имеют разную длину. Рассмотрим типичный пример, создав массив, в котором элемент, стоящий на пересечении i -й строки и j -го столбца, равен количеству вариантов выбора j чисел из i ,

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1

```

Число j не может превышать число i , из-за чего получается треугольная матрица. В i -й строке этой матрицы находится $i+1$ элемент. (Допускается выбрать и 0 элементов, но сделать это можно лишь одним способом.) Чтобы создать неровный массив, следует сначала разместить в памяти массив, хранящий его строки:

```
int[][] odds = new int[NMAX+1][];
```

Затем в памяти размещаются сами строки:

```
for (int n=0; n<=NMAX; n++)
    odds[n] = new int[n+1];
```

Теперь в памяти размещен весь массив, а следовательно, к его элементам можно обращаться, как обычно, но при условии, что индексы не выходят за допустимые пределы. В приведенном ниже фрагменте кода показано, как это делается.

```
for (int n=0; n<odds.length; n++)
    for (int k=0; k<odds[n].length; k++)
    {
        // рассчитать варианты выигрыша в лотерею
        ...
        odds[n][k] = lotteryOdds;
    }
```

Исходный код программы, реализующей рассматриваемый здесь неровный треугольный массив, приведен в листинге 3.9.

Листинг 3.9. Исходный код из файла LotteryArray/LotteryArray.java

```

1 /**
2  * В этой программе демонстрируется применение треугольного массива
3  * @version 1.20 2004-02-10
4  * @author Cay Horstmann
5  */
6 public class LotteryArray
7 {
8     public static void main(String[] args)
9     {
10         final int NMAX = 10;
11
12         // выделить память под треугольный массив
13
14         int[][] odds = new int[NMAX + 1][];
15         for (int n = 0; n <= NMAX; n++)
16             odds[n] = new int[n + 1];
17

```

```
18 // заполнить треугольный массив
19
20 for (int n = 0; n < odds.length; n++)
21     for (int k = 0; k < odds[n].length; k++)
22     {
23         /*
24          * вычислить биномиальный коэффициент:
25          * n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
26          */
27         int lotteryOdds = 1;
28         for (int i = 1; i <= k; i++)
29             lotteryOdds = lotteryOdds * (n - i + 1) / i;
30
31         odds[n][k] = lotteryOdds;
32     }
33
34 // вывести треугольный массив
35 for (int[] row : odds)
36 {
37     for (int odd : row)
38         System.out.printf("%4d", odd);
39     System.out.println();
40 }
41 }
42 }
```



НА ЗАМЕТКУ C++! В Java объявление

```
double[][] balance = new double[10][6]; // Java
```

не равнозначно следующему объявлению:

```
double balance[10][6]; // C++
```

и даже не соответствует приведенному ниже объявлению.

```
double (*balance)[6] = new double[10][6]; // C++
```

Вместо этого в памяти размещается массив, состоящий из десяти указателей. Средствами C++ это можно выразить следующим образом:

```
double** balance = new double*[10]; // C++
```

Затем каждый элемент в массиве указателей заполняется массивом, состоящим из 6 чисел, как показано ниже.

```
for (i = 0; i < 10; i++)
    balance[i] = new double[6];
```

Правда, этот цикл выполняется автоматически при объявлении массива с помощью оператора `new double[10][6]`. Если вам требуется неровный массив, размещайте в памяти массивы его строк по отдельности.

Итак, вы ознакомились со всеми основными языковыми конструкциями Java. А следующая глава посвящена особенностям объектно-ориентированного программирования на Java.

Объекты и классы

В этой главе...

- ▶ Введение в объектно-ориентированное программирование
- ▶ Применение predefined классов
- ▶ Определение собственных классов
- ▶ Статические поля и методы
- ▶ Параметры методов
- ▶ Конструирование объектов
- ▶ Пакеты
- ▶ Путь к классам
- ▶ Комментарии и документирование
- ▶ Рекомендации по разработке классов

В этой главе рассматриваются следующие вопросы.

- Введение в объектно-ориентированное программирование.
- Создание объектов классов из стандартной библиотеки Java.
- Создание собственных классов.

Если вы недостаточно хорошо ориентируетесь в вопросах объектно-ориентированного программирования, внимательно прочитайте эту главу. Для объектно-ориентированного программирования требуется совершенно иной образ мышления по сравнению с подходом, типичным для процедурно-ориентированных языков. Освоить новые принципы создания программ не всегда просто, но сделать это необходимо. Для овладения языком Java нужно хорошо знать основные понятия объектно-ориентированного программирования.

Тем, у кого имеется достаточный опыт программирования на C++, материал этой и предыдущей глав покажется хорошо знакомым. Но между Java и C++ имеются существенные отличия, поэтому последний раздел этой главы следует прочесть очень внимательно. Примечания к C++ помогут вам плавно перейти от C++ к Java.

Введение в объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) в настоящее время стало доминирующей методикой программирования, вытеснив “структурные”, процедурно-ориентированные подходы, разработанные в 1970-х годах. Java представляет собой полностью объектно-ориентированный язык, и для продуктивной работы на этом языке вам необходимо ознакомиться с основными принципами ООП.

Объектно-ориентированная программа состоит из объектов. Каждый объект обладает определенными функциональными возможностями, предоставляемыми в распоряжение пользователей, а также скрытой реализацией. Одни объекты для своих программ вы можете взять в готовом виде из библиотеки, другие вам придется спроектировать самостоятельно. Строить ли свои объекты или приобретать готовые — зависит от вашего бюджета или времени. Но, как правило, до тех пор, пока объекты удовлетворяют вашим требованиям, вам не нужно особенно беспокоиться о том, как реализованы их функциональные возможности.

Традиционное структурное программирование заключается в разработке набора процедур (или алгоритмов) для решения поставленной задачи. Определив эти процедуры, программист должен найти подходящий способ хранения данных. Вот почему создатель языка Pascal Никлаус Вирт (Niklaus Wirth) назвал свою известную книгу по программированию *Алгоритмы + Структуры данных = Программы* (Algorithms + Data Structures = Programs; издательство Prentice Hall, 1975 г). Обратите внимание на то, что в названии этой книги алгоритмы стоят на первом месте, а структуры данных — на втором. Это отражает образ мышления программистов того времени. Сначала они решали, как манипулировать данными, а затем — какую структуру применить для организации этих данных, чтобы с ними было легче работать. Подход ООП в корне изменил ситуацию, поставив на первое место данные и лишь на второе — алгоритмы, предназначенные для их обработки.

Для решения небольших задач процедурный подход оказывается вполне пригодным. Но объекты более приспособлены для решения более крупных задач. Рассмотрим для примера небольшой веб-браузер. Его реализация может потребовать 2000 процедур, каждая из которых манипулирует набором глобальных данных. В стиле ООП та же самая программа может быть составлена всего из 100 классов, в каждом из которых в среднем определено по 20 методов (рис. 4.1). Такая структура программы гораздо удобнее для программирования. В ней легче находить ошибки. Допустим, данные некоторого объекта находятся в неверном состоянии. Очевидно, что намного легче найти причину неполадок среди 20 методов, имеющих доступ к данным, чем среди 2000 процедур.

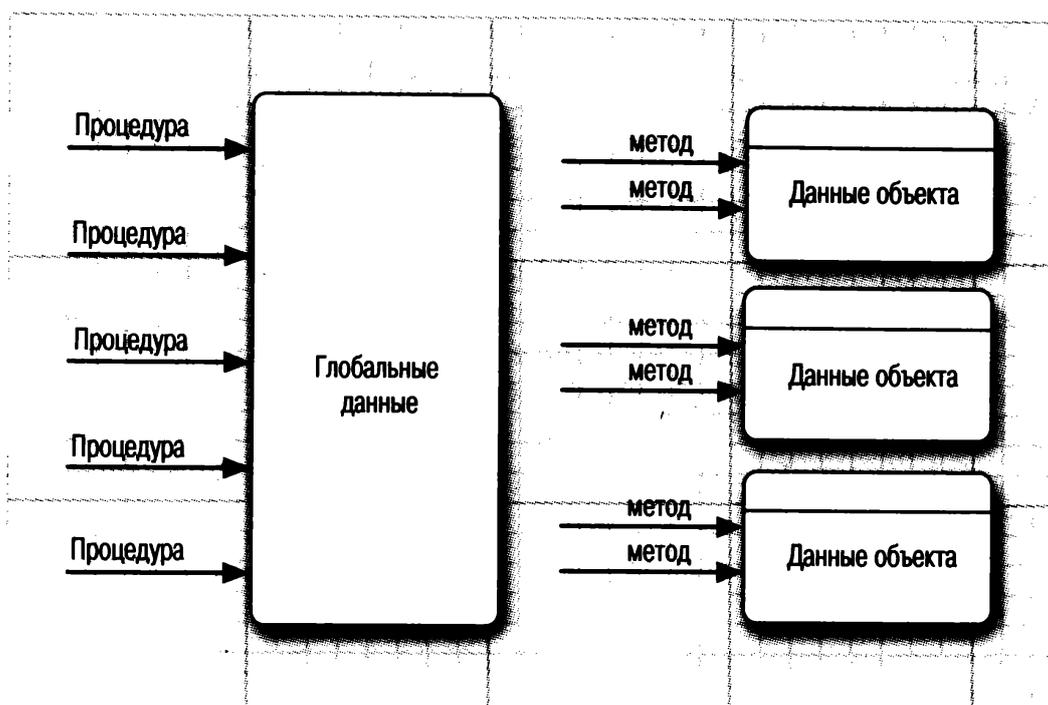


Рис. 4.1. Сравнение процедурного и объектно-ориентированного программирования

Классы

Для дальнейшей работы вам нужно усвоить основные понятия и терминологию ООП. Наиболее важным понятием является класс, применение которого уже продемонстрировалось в примерах программ из предыдущих глав. Класс — это шаблон или образец, по которому будет сделан объект. Обычно класс сравнивают с формой для выпечки печенья, а объект — это само печенье. Конструирование объекта на основе некоторого класса называется получением экземпляра этого класса.

Как следует из примеров программ в предыдущих главах, весь код, написанный на Java, находится в классах. Стандартная библиотека Java содержит несколько тысяч классов, предназначенных для решения самых разных задач, например, для построения пользовательского интерфейса, календарей, установления сетевых соединений и т.д. Несмотря на это, программирующие на Java продолжают создавать свои собственные классы, чтобы формировать объекты, характерные для разрабатываемого приложения, а также приспособлять классы из стандартной библиотеки под свои нужды.

Инкапсуляция (иногда называемая сокрытием данных) — это ключевое понятие для работы с объектами. Формально инкапсуляцией считается обычное объединение данных и операций над ними в одном пакете и сокрытие данных от других объектов. Данные в объекте называются полями экземпляра, а функции и процедуры, выполняющие операции над данными, — его методами. В конкретном объекте, т.е. экземпляре класса, поля экземпляра имеют определенные значения. Множество этих значений называется текущим состоянием объекта. Вызов любого метода для объекта может изменить его состояние.

Следует еще раз подчеркнуть, что основной принцип инкапсуляции заключается в запрещении прямого доступа к полям экземпляра данного класса из других классов. Программы должны взаимодействовать с данными объекта только через методы этого объекта. Инкапсуляция обеспечивает внутреннее поведение объектов, что имеет решающее значение для повторного их использования и надежности работы

программ. Это означает, что в классе можно полностью изменить способ хранения данных. Но поскольку для манипулирования данными используются одни и те же методы, то об этом ничего не известно, да и не особенно важно другим объектам.

Еще один принцип ООП облегчает разработку собственных классов в Java: один класс можно построить на основе других классов. В этом случае говорят, что новый класс *расширяет* тот класс, на основе которого он создан. Язык Java, по существу, создан на основе “глобального суперкласса”, называемого `Object`. Все остальные объекты расширяют его. В следующей главе мы рассмотрим этот вопрос подробнее.

Если вы разрабатываете класс на основе уже существующего, то новый класс содержит все свойства и методы расширяемого класса. Кроме того, в него добавляются новые методы и поля данных. Расширение класса и получение на его основе нового называется *наследованием*. Более подробно принцип наследования будет рассмотрен в следующей главе.

Объекты

В объектно-ориентированном программировании определены следующие ключевые свойства объектов.

- *Поведение* объекта — что с ним можно делать и какие методы к нему можно применять.
- *Состояние* объекта — как этот объект реагирует на применение методов.
- *Идентичность* объекта — чем данный объект отличается от других, характеризующихся таким же поведением и состоянием.

Все объекты, являющиеся экземплярами одного и того же класса, ведут себя одинаково. *Поведение* объекта определяется методами, которые можно вызвать. Каждый объект сохраняет информацию о своем *состоянии*. Со временем состояние объекта может измениться, но спонтанно это произойти не может. Состояние объекта может изменяться только в результате вызовов методов. (Если состояние объекта изменилось вследствие иных причин, значит, принцип инкапсуляции не соблюден.)

Состояние объекта не полностью описывает его, поскольку каждый объект имеет свою собственную *идентичность*. Например, в системе обработки заказов два заказа могут отличаться друг от друга, даже если они относятся к одним и тем же товарам. Заметим, что индивидуальные объекты, представляющие собой экземпляры класса, *всегда* отличаются своей идентичностью и, как правило, — своим состоянием.

Эти основные свойства объектов могут оказывать взаимное влияние. Например, состояние объекта может оказывать влияние на его поведение. (Если заказ выполнен или оплачен, объект может отказаться выполнить вызов метода, требующего добавить или удалить товар. И наоборот, если заказ пуст, т.е. ни одна единица товара не была заказана, он не может быть выполнен.)

Идентификация классов

В традиционной процедурной программе выполнение начинается сверху, т.е. с функции `main()`. При проектировании объектно-ориентированной системы понятия “верха” как такового не существует, и поэтому начинающие осваивать ООП часто интересуются, с чего же следует начинать. Ответ таков: сначала нужно идентифицировать классы, а затем добавить методы в каждый класс.

Простое эмпирическое правило для идентификации классов состоит в том, чтобы выделить для них имена существительные при анализе проблемной области. С другой стороны, методы соответствуют глаголам, обозначающим действие. Например, при описании системы обработки заказов используются следующие имена существительные.

- Изделие.
- Заказ.
- Адрес доставки.
- Оплата.
- Счет.

Этим именам соответствуют классы `Item`, `Order` и т.д.

Далее выбираются глаголы. Изделия *вводятся* в заказы. Заказы *выполняются* или *отменяются*. Оплата заказа *осуществляется*. Используя эти глаголы, можно определить объект, выполняющий такие действия. Так, если поступил новый заказ, ответственность за его обработку должен нести объект `Order` (Заказ), поскольку именно в нем содержится информация о способе хранения и сортировке заказываемых предметов. Следовательно, в классе `Order` должен существовать метод `add()` — добавить, получающий объект `Item` (Товар) в качестве параметра.

Разумеется, упомянутое выше правило выбора имен существительных и глаголов является не более чем рекомендацией. И только опыт может помочь программисту решить, какие именно существительные и глаголы следует выбрать при создании класса.

Отношения между классами

Между классами существуют три общих вида отношений.

- Зависимость (“использует — что-то”).
- Агрегирование (“содержит — что-то”).
- Наследование (“является — чем-то”).

Отношение *зависимости* наиболее очевидное и распространенное. Например, класс `Order` использует класс `Account`, поскольку объекты типа `Order` должны иметь доступ к объектам типа `Account`, чтобы проверить кредитоспособность заказчика. Но класс `Item` не зависит от класса `Account`, потому что объекты `Item` вообще не интересуют состояние счета заказчика. Следовательно, класс зависит от другого класса, если его методы выполняют какие-либо действия над экземплярами этого класса. Старайтесь свести к минимуму количество взаимозависимых классов. Если класс `A` не знает о существовании класса `B`, то он тем более ничего не знает о любых изменениях в нем! (Это означает, что любые изменения в классе `B` не повлияют на поведение объектов класса `A`.)

Отношение *агрегирования* понять нетрудно, потому что оно конкретно. Например, объект типа `Order` может содержать объекты типа `Item`. Агрегирование означает, что объект класса `A` содержит объекты класса `B`.



НА ЗАМЕТКУ! Некоторые специалисты не признают понятие агрегирования и предпочитают использовать отношение связи, или ассоциации. С точки зрения моделирования это разумно. Но для программистов гораздо удобнее отношение, при котором один объект содержит другой. Мы предпочитаем использовать понятие агрегирования еще по одной причине: его обозначение проще для восприятия, чем обозначение отношения связи (табл. 4.1).

Наследование выражает отношение между более конкретным и более общим классом. Например, класс `RushOrder` наследует от класса `Order`. Специализированный класс `RushOrder` содержит особые методы для обработки приоритетов и разные методы для вычисления стоимости доставки, в то время как другие его методы, например, для заказа товара и выписывания счета, унаследованы от класса `Order`. Вообще говоря, если класс `A` расширяет класс `B`, класс `A` наследует методы класса `B` и, кроме них, имеет дополнительные возможности. (Более подробно наследование рассматривается в следующей главе.)

Многие программисты пользуются средствами UML (Unified Modeling Language — унифицированный язык моделирования) для составления диаграмм классов, описывающих отношения между классами. Пример такой диаграммы приведен на рис. 4.2. Здесь классы показаны прямоугольниками, а отношения между ними — различными стрелками. В табл. 4.1 приведены основные обозначения, принятые в языке UML.

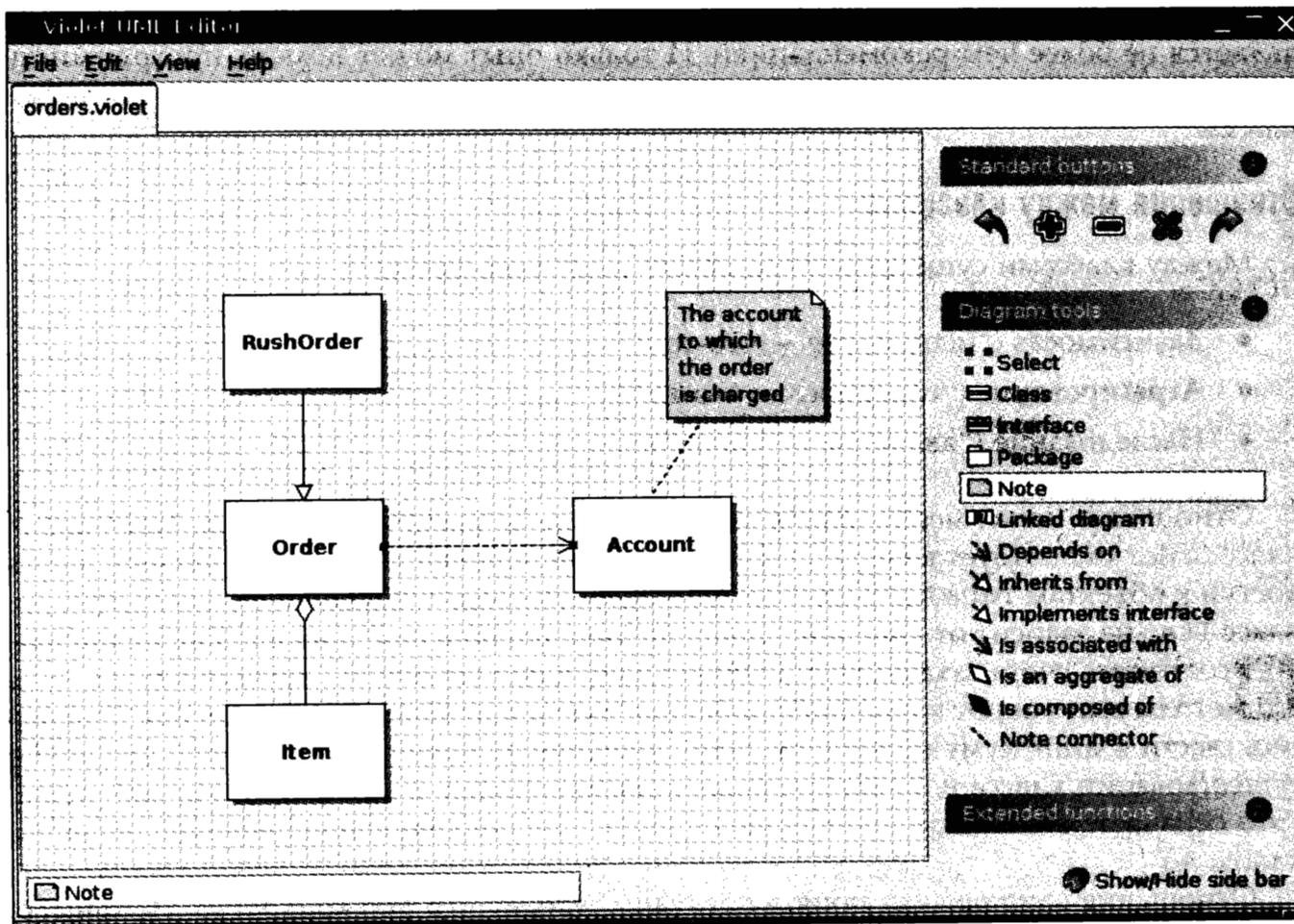


Рис. 4.2. Диаграмма классов

Таблица 4.1. Обозначение отношений между классами в UML

Отношение	Обозначение в UML
Наследование	
Реализация интерфейса	
Зависимость	
Агрегирование	
Связь	
Направленная связь	

Применение predefined классов

В Java ничего нельзя сделать без классов, поэтому мы вкратце обсудили в предыдущих разделах, каким образом действуют некоторые из них. Но в Java имеются также классы, к которым не совсем подходят приведенные выше рассуждения. Характерным тому примером служит класс `Math`. Как упоминалось в предыдущей главе, методы из класса `Math`, например метод `Math.random()`, можно вызывать, вообще ничего не зная об их реализации. Для обращения к методу достаточно знать его имя и параметры (если они предусмотрены). Это признак инкапсуляции, который, безусловно, справедлив для всех классов. Но в классе `Math` отсутствуют данные; он инкапсулирует *только* функциональные возможности, не требуя ни данных, ни их сокрытия. Это означает, что можно и не заботиться о создании объектов и инициализации их полей, поскольку в классе `Math` ничего подобного нет!

В следующем разделе будет рассмотрен класс `Date`. На примере этого класса будет показано, каким образом создаются экземпляры и вызываются методы класса.

Объекты и объектные переменные

Чтобы работать с объектами, их нужно сначала создать и задать их исходное состояние. Затем к этим объектам можно применять методы. В Java для создания новых экземпляров служат *конструкторы*. Конструктор — это специальный метод, предназначенный для создания и инициализации экземпляра класса. В качестве примера можно привести класс `Date`, входящий в стандартную библиотеку Java. С помощью объектов этого класса можно описать текущий или любой другой момент времени, например "December 31, 1999, 23:59:59 GMT".



НА ЗАМЕТКУ! У вас может возникнуть вопрос: почему для представления даты и времени применяются классы, а не встроенные типы, как в ряде других языков программирования? Такой подход применяется, например, в Visual Basic, где дата задается в формате #6/1/1995#. На первый взгляд это удобно — программист может использовать встроенный тип и не заботиться о классах. Но не является ли такое удобство кажущимся? В одних странах даты записываются в формате месяц/день/год, а в других — год/месяц/день. Могут ли создатели языка предусмотреть все возможные варианты? Даже если это и удастся сделать, соответствующие средства будут слишком сложны, причем программисты будут вынуждены применять их. Использование классов позволяет переложить ответственность за решение этих проблем с создателей языка на разработчиков библиотек. Если системный класс не годится, то разработчики всегда могут написать свой собственный. (В качестве аргумента в пользу такого подхода отметим, что библиотека дат на Java довольно запутана, и ее переделка уже началась; см. <http://jcp.org/en/jsr/detail?id=310>.)

Имя конструктора всегда совпадает с именем класса. Следовательно, конструктор класса `Date` называется `Date`. Чтобы создать объект типа `Date`, конструктор следует объединить с операцией `new`, как показано ниже.

```
new Date()
```

В этом выражении создается новый объект, который инициализируется текущими датой и временем.

При желании объект можно передать методу:

```
System.out.println(new Date());
```

И наоборот, можно вызвать метод вновь созданного объекта. Среди методов класса `Date` есть метод `toString()`, позволяющий представить дату в виде символьной строки. Его можно вызвать следующим образом:

```
String = new Date().toString();
```

В этих двух примерах созданный объект использовался только один раз. Но, как правило, объект приходится использовать неоднократно. Чтобы это стало возможным, необходимо связать объект с некоторым идентификатором, иными словами, присвоить объект переменной, как показано ниже.

```
Date birthday = new Date();
```

На рис. 4.3 наглядно показано, каким образом переменная `birthday` ссылается на вновь созданный объект.

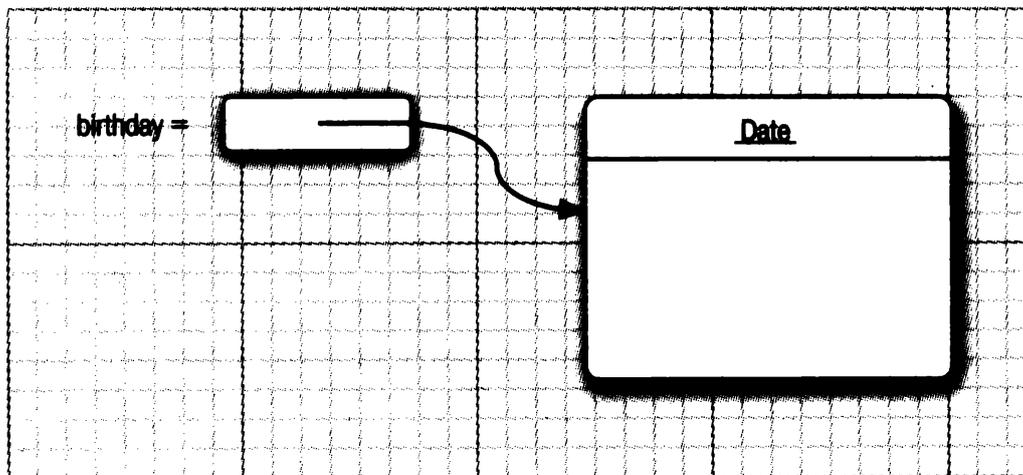


Рис. 4.3. Создание нового объекта

Между объектами и объектными переменными есть существенная разница. Например, в приведенном ниже выражении определяется объектная переменная `deadline`, которая может ссылаться на объекты типа `Date`.

```
Date deadline; // переменная deadline не ссылается ни на один из объектов
```

Важно понимать, что на данном этапе сама переменная `deadline` объектом не является и даже не ссылается ни на один из объектов. Поэтому ни один из методов класса `Date` пока еще нельзя вызывать по этой переменной. Попытка сделать это приведет к появлению сообщения об ошибке.

```
s = deadline.toString(); // вызывать метод еще рано!
```

Сначала переменную `deadline` нужно инициализировать. Для этого имеются две возможности. Прежде всего, переменную можно инициализировать вновь созданным объектом:

```
deadline = new Date();
```

Кроме того, можно заставить переменную ссылаться на существующий объект, как показано ниже.

```
deadline = birthday;
```

Теперь переменные `deadline` и `birthday` ссылаются на *один и тот же* объект (рис. 4.4).

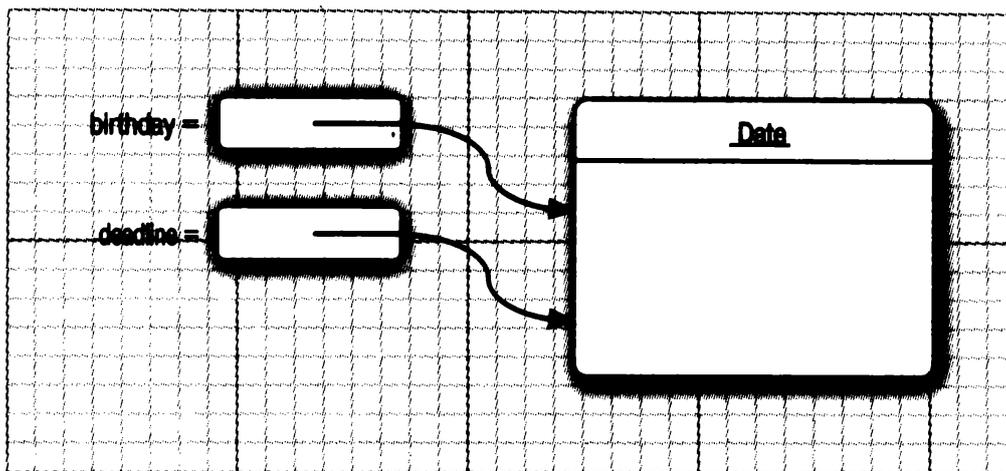


Рис. 4.4. Объектные переменные, ссылающиеся на один и тот же объект

Важно помнить, что объектная переменная фактически не содержит никакого объекта. Она лишь ссылается на него. В Java значение любой объектной переменной представляет собой ссылку на объект, размещенный в другом месте. Операция `new` также возвращает ссылку. Например, приведенная ниже строка кода состоит из двух частей: выражение `new Date()` создает объект типа `Date`, а значение переменной представляет собой ссылку на вновь созданный объект.

```
Date deadline = new Date();
```

Объектной переменной можно явно присвоить пустое значение `null`, чтобы указать на то, что она пока не ссылается ни на один из объектов:

```
deadline = null;
...
if (deadline != null)
    System.out.println(deadline);
```

Если попытаться вызвать метод, ссылаясь по переменной, значение которой равно `null`, то при выполнении программы возникнет ошибка:

```
birthday = null;
String s = birthday.toString(); // Ошибка при выполнении!
```

Локальные переменные не инициализируются автоматически пустым значением `null`. Программирующий должен сам инициализировать переменную, выполнив операцию `new` или присвоив пустое значение `null`.



НА ЗАМЕТКУ C++! Многие ошибочно полагают, что объектные переменные в Java похожи на ссылки в C++. Но в C++ пустые ссылки не допускаются и не присваиваются. Объектные переменные в Java следует считать аналогами указателей на объекты. Например, выражение

```
Date birthday; // Java
```

почти эквивалентно следующей строке кода:

```
Date* birthday; // C++
```

Такая аналогия все расставляет на свои места. Разумеется, указатель `Date*` не инициализируется до тех пор, пока не будет выполнена операция `new`. Синтаксис подобных выражений в C++ и Java почти совпадает.

```
Date* birthday = new Date(); // C++
```

При копировании одной переменной в другую в обеих переменных появляется ссылка на один и тот же объект. В C++ эквивалентом пустой ссылки типа `null` в Java служит указатель `NULL`.

Все объекты в Java располагаются в динамической области памяти, называемой "кучей". Если объект содержит другую объектную переменную, она представляет собой всего лишь указатель на другой объект, расположенный в этой области памяти.

В C++ указатели доставляют немало хлопот, поскольку они часто приводят к ошибкам. Очень легко создать неверный указатель или потерять управление памятью. А в Java подобные осложнения вообще не возникают. Если вы используете неинициализированный указатель, то исполняющая система обязательно сообщит об ошибке, а не продолжит выполнение некорректной программы, выдавая случайные результаты. Вам не нужно беспокоиться об управлении памятью, поскольку механизм "сборки мусора" выполняет все необходимые операции с памятью автоматически.

В C++ большое внимание уделено автоматическому копированию объектов с помощью копирующих конструкторов и операций присваивания. Например, копией связного списка является новый связный список, который, имея старое содержимое, содержит совершенно другие связи. Иными словами, копирование объектов осуществляется так же, как и копирование встроенных типов. А в Java для получения полной копии объекта служит метод `clone()`.

Класс `GregorianCalendar` из библиотеки Java

В предыдущих примерах использовался класс `Date`, входящий в состав стандартной библиотеки Java. Экземпляр класса `Date` находится в состоянии, которое отражает конкретный момент времени.

При использовании класса `Date` совсем не обязательно знать о формате даты, тем не менее, время представлено количеством миллисекунд (положительным или отрицательным), отсчитанным от фиксированного момента времени, так называемого начала эпохи, т.е. от момента времени 00:00:00 UTC, 1 января 1970 г. Сокращение UTC означает Universal Coordinated Time (универсальное скоординированное время) — научный стандарт времени. Стандарт UTC применяется наряду с более известным стандартом GMT (Greenwich Mean Time — среднее время по Гринвичу).

Но класс `Date` не очень удобен для работы с датами. Разработчики библиотеки Java считали, что представление даты, например "December 31, 1999, 23:59:59", является совершенно произвольным и должно зависеть от календаря. Данное конкретное представление подчиняется григорианскому календарю, самому распространенному в мире. Но тот же самый момент времени совершенно иначе представляется в китайском или еврейском лунном календаре, не говоря уже о календаре, которым будут пользоваться потенциальные заказчики с Марса.



НА ЗАМЕТКУ! Вся история человечества сопровождалась созданием календарей — систем именования различных моментов времени. Как правило, основой для календарей был солнечный или лунный цикл. Если вас интересуют подобные вопросы, обратитесь за справкой, например, к книге Наума Дершовица (Nachum Dershowitz) и Эдварда М. Рейнгольда (Edward M. Reingold) *Calendrical Calculations* (издательство Cambridge University Press, 2nd ed., 2001 г.). Там вы найдете исчерпывающие сведения о календаре французской революции, календаре Майя и других экзотических системах отсчета времени.

Разработчики библиотеки Java решили разделить вопросы, связанные с отслеживанием моментов времени, от вопросов их представления. Таким образом, стандартная библиотека Java содержит два отдельных класса: класс `Date`, представляющий момент времени, и класс `GregorianCalendar`, расширяющий более общий класс `Calendar`, описывающий свойства календаря в целом. Кроме того, в стандартной библиотеке Java реализованы тайский буддистский и японский имперский календари.

Отделение измерения времени от календарей является грамотным решением, вполне отвечающим принципам ОПП. Класс `Date` содержит лишь небольшое количество методов, позволяющих сравнивать два момента времени. Например, методы `before()` и `after()` определяют, предшествует ли один момент времени другому или же следует за ним, как показано ниже.

```
if (today.before(birthday))
    System.out.println("Still time to shop for a gift.");
```



НА ЗАМЕТКУ! На самом деле в классе `Date` имеются такие методы, как `getDay()`, `getMonth()` и `getYear()`, но использовать их без крайней необходимости не рекомендуется. Метод объявляется не рекомендованным к применению, когда разработчики библиотеки решают, что его не стоит больше применять в новых программах.

Эти методы были частью класса `Date` еще до того, как разработчики библиотеки Java поняли, что классы, реализующие разные календари, разумнее было бы разделить друг от друга. Сделав это, они поместили эти методы из класса `Date` как не рекомендованные к применению. Вы можете и далее пользоваться ими в своих программах, получая при этом предупреждения от компилятора, а еще лучше вообще отказаться от их применения, поскольку они могут быть удалены из последующих версий библиотеки.

Класс `GregorianCalendar` содержит намного больше методов, чем класс `Date`. В частности, в нем есть несколько полезных конструкторов. Так, приведенное ниже выражение формирует новый объект, представляющий дату и момент времени, когда он был создан.

```
new GregorianCalendar()
```

Объект, представляющий полночь даты, указанной в формате год/месяц/день, можно создать так:

```
new GregorianCalendar(1999, 11, 31)
```

Любопытно, что количество месяцев отсчитывается от нуля. Так, число **11** означает декабрь. Для большей ясности можно использовать константу `Calendar.DECEMBER` следующим образом:

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31)
```

Кроме того, с помощью конструктора типа `GregorianCalendar` можно задать время, как показано ниже.

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31, 23, 59, 59)
```

Чаще всего ссылка на созданный объект присваивается переменной следующим образом:

```
GregorianCalendar deadline = new GregorianCalendar(...);
```

В классе `GregorianCalendar` инкапсулированы поля экземпляра, в которых хранится указанная дата. Не имея доступа к исходному тексту, невозможно определить, какое именно представление даты и времени используется в этом классе. Но в силу принципа инкапсуляции это и не важно. А важнее другое: какие методы доступны в данном классе.

Модифицирующие методы и методы доступа

После знакомления с классом `GregorianCalendar` у вас могут возникнуть следующие вопросы: как получить текущий день, месяц или год, исходя из даты, инкапсулированной в объекте данного класса, и каким образом можно изменить эти значения? Ответы на подобные вопросы вы найдете в этом разделе, а дополнительные сведения — в документации на прикладной интерфейс API. Рассмотрим наиболее важные методы данного класса.

Календарь должен вычислять атрибуты, соответствующие указанному моменту времени, например, дату, день недели, месяц или год. Получить одно из этих значений можно, используя метод `get()` из класса `GregorianCalendar`. Чтобы выбрать желаемый атрибут, нужно передать методу константу, определенную в классе `Calendar`, например `Calendar.MONTH` или `Calendar.DAY_OF_WEEK`, следующим образом:

```
GregorianCalendar now = new GregorianCalendar();  
int month = now.get(Calendar.MONTH);  
int weekday = now.get(Calendar.DAY_OF_WEEK);
```

В конце этого раздела перечислены все константы, которые можно использовать для получения требуемой информации из объекта типа `GregorianCalendar`. Состояние объекта можно изменить с помощью метода `set()` следующим образом:

```
deadline.set(Calendar.YEAR, 2001);  
deadline.set(Calendar.MONTH, Calendar.APRIL);  
deadline.set(Calendar.DAY_OF_WEEK, 15);
```

Существует способ установить год, месяц и день с помощью одного вызова:

```
deadline.set(2001, Calendar.APRIL, 15);
```

По желанию можно добавить к заданной дате определенное количество дней, недель, месяцев и т.д.

```
deadline.add(Calendar.MONTH, 3); // сдвинуть крайний срок на 3 месяца
```

Если же добавить отрицательное число, то момент времени, описываемый объектом, будет сдвинут назад.

У метода `get()`, с одной стороны, и методов `set()` и `add()`, с другой, имеется принципиальное отличие. Метод `get()` только просматривает состояние объекта и сообщает о нем, в то время как методы `set()` и `add()` модифицируют состояние объекта. Методы, способные изменять поля экземпляра, называются *модифицирующими*, а методы, которые могут лишь просматривать поля экземпляра, не изменяя их, — *методами доступа*.



НА ЗАМЕТКУ C++! В C++ для обозначения метода доступа служит суффикс `const`. Метод, не объявленный с помощью ключевого слова `const`, считается модифицирующим. Но в Java нет специальных синтаксических конструкций, позволяющих отличать модифицирующие методы от методов доступа.

В именах методов доступа принято использовать префикс `get`, а в именах модифицирующих методов — префикс `set`. Например, класс `GregorianCalendar` обеспечивает получение и установку момента времени, представленного объектом, с помощью методов `getTime()` и `setTime()`:

```
Date time = calendar.getTime();
calendar.setTime(time);
```

Эти методы особенно полезны для преобразований объектов типа `GregorianCalendar` в объекты типа `Date`, и наоборот. Допустим, известны год, месяц и день и требуется создать объект типа `Date`, поля которого были бы заполнены этими значениями. В классе `Date` ничего не известно о календарях, поэтому сначала создается объект типа `GregorianCalendar`, а затем вызывается метод `getTime()` для получения даты:

```
GregorianCalendar calendar = new GregorianCalendar(year, month, day);
Date hireDay = calendar.getTime();
```

И наоборот, если требуется определить год, месяц или день, хранящиеся в объекте типа `Date`, с этой целью создается объект типа `GregorianCalendar`, устанавливается время, а затем вызывается метод `get()`, как показано ниже.

```
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(hireDay);
int year = calendar.get(Calendar.YEAR);
```

В конце этого раздела приведена программа, в которой демонстрируется применение класса `GregorianCalendar`. Эта программа выводит на экран календарь текущего месяца в следующем формате:

```
Sun Mon Tue Wed Thu Fri Sat
                1
 2   3   4   5   6   7   8
 9  10  11  12  13  14  15
16  17  18  19* 20  21  22
23  24  25  26  27  28  29
30  31
```

Текущий день помечен звездочкой. Как видите, программа должна знать, как вычисляется длина месяца и текущий день недели. Рассмотрим основные стадии выполнения данной программы. Сначала в ней создается объект типа `GregorianCalendar`, инициализированный текущей датой.

```
GregorianCalendar d = new GregorianCalendar();
```

Затем определяется текущий день и месяц, для чего дважды вызывается метод `get()`.

```
int today = d.get(Calendar.DAY_OF_MONTH);
int month = d.get(Calendar.MONTH);
```

В переменной `weekday` устанавливается значение `Calendar.SUNDAY`, если первый день месяца — воскресенье, `Calendar.MONDAY` — если понедельник, и т.д. (Эти

значения на самом деле являются целыми числами 1, 2, ..., 7, но ими лучше не пользоваться при написании кода, поскольку это не способствует его удобочитаемости.)

Обратите внимание на то, что первая строка календаря выведена с отступом, чтобы первый день месяца выпадал на соответствующий день недели. Сделать это не просто, поскольку существуют разные соглашения относительно первого дня недели. В Соединенных Штатах неделя начинается с воскресенья и заканчивается субботой, в то время как в Европе она начинается с понедельника и оканчивается воскресеньем.

Виртуальной машине Java известны *региональные настройки* текущего пользователя. Региональные настройки описывают местные соглашения относительно форматирования, включая начало недели и названия дней недели.



СОВЕТ. Если результат выполнения программы требуется вывести с учетом других региональных настроек, добавьте в начале метода `main()` строку кода, подобную следующей:

```
Locale.setDefault(Locale.ITALY);
```

Метод `getFirstDayOfWeek()` получает начальный день недели с учетом текущих региональных настроек. Чтобы определить размер необходимого отступа, отнимается 1 от дня, описанного объектом календаря, до тех пор, пока не будет достигнут первый день недели:

```
int firstDayOfWeek = d.getFirstDayOfWeek();
int indent = 0;
while (weekday != firstDayOfWeek)
{
    indent++;
    d.add(Calendar.DAY_OF_MONTH, -1);
    weekday = d.get(Calendar.DAY_OF_WEEK);
}
```

Затем выводится заголовок с наименованиями дней недели. Они получаются из класса `DateFormatSymbols` следующим образом:

```
String [] weekdayNames = new DateFormatSymbols().getShortWeekdays();
```

Метод `getShortWeekdays()` возвращает символьную строку с сокращенными наименованиями дней недели на языке пользователя (на английском — "Sun", "Mon" и т.д.). Массив проиндексирован значениями дней недели. Ниже показано, как будет выглядеть цикл, выводящий заголовок.

```
do
{
    System.out.printf("%4s", weekdayNames[weekday]);
    d.add(Calendar.DAY_OF_MONTH, 1);
    weekday = d.get(Calendar.DAY_OF_WEEK);
}
while (weekday != firstDayOfWeek);
System.out.println();
```

Теперь все готово для вывода самого календаря. Сначала выравнивается первая строка и устанавливается объект даты в начало месяца. Затем начинается цикл, где дата `d` перебирается по всем дням месяца. На каждом шаге цикла выводится значение даты. Если дата `d` приходится на сегодня, этот день помечается знаком *. Когда в цикле достигается начало новой недели, выводится знак перевода строки. Затем дата `d` устанавливается на следующий день, как показано ниже.

```
d.add(Calendar.DAY_OF_MONTH, 1);
```

Когда же следует остановиться? Заранее неизвестно, сколько в месяце дней: **31, 30, 29** или **28**. Поэтому цикл продолжается до тех пор, пока объект даты **d** остается в пределах текущего месяца.

```
do
{
    ...
}
while (d.get(Calendar.MONTH) == month);
```

И как только объект даты **d** перейдет на следующий месяц, программа завершит свою работу. Исходный код этой программы полностью приведен в листинге 4.1.

Как видите, класс `GregorianCalendar` позволяет легко создавать программы для работы с календарем, выполняя такие сложные действия, как отслеживание дней недели и учет продолжительности месяцев. Программирующему не нужно ничего знать, каким образом в классе `GregorianCalendar` вычисляются месяцы и дни недели. Ему достаточно пользоваться *интерфейсом* данного класса, включая методы `get()`, `set()` и `add()`. Основное назначение рассмотренной здесь программы — показать, как пользоваться интерфейсом класса для решения сложных задач, не вникая в подробности реализации.

Листинг 4.1. Исходный код из файла `CalendarTest/CalendarTest.java`

```
1 import java.text.DateFormatSymbols;
2 import java.util.*;
3 /**
4  * @version 1.4 2007-04-07
5  * @author Cay Horstmann
6  */
7
8 public class CalendarTest
9 {
10     public static void main(String[] args)
11     {
12         // построить объекты d текущей даты
13         GregorianCalendar d = new GregorianCalendar();
14
15         int today = d.get(Calendar.DAY_OF_MONTH);
16         int month = d.get(Calendar.MONTH);
17
18         // установить объект d на начало месяца
19         d.set(Calendar.DAY_OF_MONTH, 1);
20
21         int weekday = d.get(Calendar.DAY_OF_WEEK);
22
23         // получить первый день недели (воскресенье в США)
24         int firstDayOfWeek = d.getFirstDayOfWeek();
25
26         // определить отступ, требующийся в первой строке
27         int indent = 0;
28         while (weekday != firstDayOfWeek)
29         {
30             indent++;
31             d.add(Calendar.DAY_OF_MONTH, -1);
```

```
32     weekday = d.get(Calendar.DAY_OF_WEEK);
33 }
34
35 // вывести названия дней недели
36 String[] weekdayNames = new DateFormatSymbols().getShortWeekdays();
37 do
38 {
39     System.out.printf("%4s", weekdayNames[weekday]);
40     d.add(Calendar.DAY_OF_MONTH, 1);
41     weekday = d.get(Calendar.DAY_OF_WEEK);
42 }
43 while (weekday != firstDayOfWeek);
44 System.out.println();
45 for (int i = 1; i <= indent; i++)
46     System.out.print(" ");
47
48 d.set(Calendar.DAY_OF_MONTH, 1);
49 do
50 {
51     // вывести день недели
52     int day = d.get(Calendar.DAY_OF_MONTH);
53     System.out.printf("%3d", day);
54
55     // пометить текущий день знаком *
56     if (day == today) System.out.print("*");
57     else System.out.print(" ");
58
59     // продвинуть объект d к следующему дню
60     d.add(Calendar.DAY_OF_MONTH, 1);
61     weekday = d.get(Calendar.DAY_OF_WEEK);
62
63     // начать очередную неделю с новой строки
64     if (weekday == firstDayOfWeek) System.out.println();
65 }
66 while (d.get(Calendar.MONTH) == month);
67 // завершить цикл, когда в объекте d устанавливается
68 // первый день следующего месяца
69
70 // перевести строку, если требуется
71 if (weekday != firstDayOfWeek) System.out.println();
72 }
73 }
```

java.util.GregorianCalendar 1.1

- `GregorianCalendar()`
Создает объект типа `GregorianCalendar`, представляющий текущее время для часового пояса с учетом заданных по умолчанию региональных настроек.
- `GregorianCalendar(int year, int month, int day)`
- `GregorianCalendar(int year, int month, int day, int hour, int minutes, int seconds)`

Создают объект типа `GregorianCalendar`, представляющий указанную дату и время.

Параметры:

<code>year</code>	Год
<code>month</code>	Месяц (отсчет начинается с нуля, т.е. 0 — это январь)
<code>day</code>	День
<code>hour</code>	Час (от 0 до 23)
<code>minutes</code>	Минуты (от 0 до 59)
<code>seconds</code>	Секунды (от 0 до 59)

- `int get(int field)`

Возвращает значение из указанного поля.

Параметры:

<code>field</code>	Принимает значение одной из следующих констант: <code>Calendar.ERA</code> , <code>Calendar.YEAR</code> , <code>Calendar.MONTH</code> , <code>Calendar.WEEK_OF_YEAR</code> , <code>Calendar.DAY_OF_YEAR</code> , <code>Calendar.DAY_OF_WEEK</code> , <code>Calendar.DAY_OF_WEEK_IN_MONTH</code> , <code>Calendar.AM_PM</code> , <code>Calendar.HOUR</code> , <code>Calendar.HOUR_OF_DAY</code> , <code>Calendar.MINUTE</code> , <code>Calendar.SECOND</code> , <code>Calendar.MILLISECOND</code> , <code>Calendar.ZONE_OFFSET</code> или <code>Calendar.DST_OFFSET</code>
--------------------	---

- `void set(int field, int value)`

Устанавливает значения в указанном поле.

Параметры:

<code>field</code>	Одна из констант, допустимых для метода <code>get()</code>
<code>value</code>	Новое значение

- `void set(int year, int month, int day)`
 - `void set(int year, int month, int day, int hour, int minutes, int seconds)`
- Устанавливают новые значения полей.

Параметры:

<code>year</code>	Год
<code>month</code>	Месяц (отсчет начинается с нуля, т.е. 0 — это январь)
<code>day</code>	День
<code>hour</code>	Час (от 0 до 23)
<code>minutes</code>	Минуты (от 0 до 59)
<code>seconds</code>	Секунды (от 0 до 59)

- `void add(int field, int amount)`

Метод для арифметических операций с датой и временем. Добавляет к полям заданную величину. Например, чтобы прибавить 7 дней к текущей дате, следует сделать вызов `c.add(Calendar.DAY_OF_MONTH, 7)`.

Параметры: *field* Изменяемое поле (должно содержать значение одной из констант, предусмотренных для метода `get()`)

amount Добавляемая величина, на которую изменяется поле (может быть отрицательной)

- `int getFirstDayOfWeek()`

Получает первый день недели с учетом региональных настроек текущего пользователя. Например, для США это будет `Calendar.SUNDAY`.

`void setTime(Date time)`

Устанавливает заданный момент времени.

Параметры: *time* Момент времени

- `Date getTime()`

Возвращает момент времени, представленный текущим значением данного календарного объекта.

java.text.DateFormatSymbols 1.1

- `String[] getShortWeekDays()`
- `String[] getShortMonths()`
- `String[] getWeekDays()`
- `String[] getMonths()`

Получают наименования дней недели и месяцев с учетом текущих региональных настроек. Используют константы дней недели и месяцев из класса `Calendar` в качестве индексов массива.

Определение собственных классов

В примерах кода из главы 3 уже предпринималась попытка создавать простые классы. Но все они состояли из единственного метода `main()`. Теперь настало время показать, как создаются “рабочие” классы для более сложных приложений. Как правило, в этих классах метод `main()` отсутствует. Вместо этого они содержат другие методы и поля. Чтобы написать полностью завершённую программу, нужно объединить несколько классов, один из которых содержит метод `main()`.

Класс `Employee`

Простейшая форма определения класса в Java выглядит следующим образом:

```
class ИмяКласса
{
    поле_1
    поле_2

    конструктор_1
    конструктор_2
    ...
    метод_1
    метод_2
    ...
}
```

Рассмотрим следующую, весьма упрощенную версию класса `Employee`, который можно использовать для составления платежной ведомости:

```
class Employee
{
    // поля экземпляра
    private String name;
    private double salary;
    private Date hireDay;

    // конструктор
    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1,
day);
        hireDay = calendar.getTime();
    }

    // метод
    public String getName()
    {
        return name;
    }

    // другие методы
    . . .
}
```

Более подробно реализация этого класса будет проанализирована в последующих разделах. А до тех пор рассмотрим код, приведенный в листинге 4.2 и демонстрирующий практическое применение класса `Employee`.

Листинг 4.2. Исходный код из файла `EmployeeTest/EmployeeTest.java`

```
1 import java.util.*;
2
3 /**
4  * В этой программе проверяется класс Employee
5  * @version 1.11 2004-02-19
```

```
6  * @author Cay Horstmann
7  */
8  public class EmployeeTest
9  {
10     public static void main(String[] args)
11     {
12         // заполнить массив staff тремя объектами типа Employee
13         Employee[] staff = new Employee[3];
14         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
15         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
16         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
17
18         // увеличить зарплату на 5%
19         for (Employee e : staff)
20             e.raiseSalary(5);
21
22         // вывести данные обо всех объектах типа Employee
23         for (Employee e : staff)
24             System.out.println("name=" + e.getName() + ",salary=" +
25                 e.getSalary() + ",hireDay=" + e.getHireDay());
26     }
27 }
28
29 class Employee
30 {
31     private String name;
32     private double salary;
33     private Date hireDay;
34
35     public Employee(String n, double s, int year, int month, int day)
36     {
37         name = n;
38         salary = s;
39         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
40         // в классе GregorianCalendar январь обозначается нулем
41         hireDay = calendar.getTime();
42     }
43
44     public String getName()
45     {
46         return name;
47     }
48
49     public double getSalary()
50     {
51         return salary;
52     }
53
54     public Date getHireDay()
55     {
56         return hireDay;
57     }
58
59     public void raiseSalary(double byPercent)
60     {
61         double raise = salary * byPercent / 100;
62         salary += raise;
63     }
64 }
```

В данной программе сначала создается массив типа `Employee`, в который заносятся три объекта сотрудников следующим образом:

```
Employee[] staff = new Employee[3];
staff[0] = new Employee("Carl Cracker", . . .);
staff[1] = new Employee("Harry Hacker", . . .);
staff[2] = new Employee("Tony Tester", . . .);
```

Затем вызывается метод `raiseSalary()` из класса `Employee`, чтобы поднять зарплату каждого сотрудника на 5%, как показано ниже.

```
for (Employee e : staff)
    e.raiseSalary(5);
```

И наконец, с помощью методов `getName()`, `getSalary()` и `getHireDay()` выводятся данные о каждом сотруднике.

```
for (Employee e : staff)
    System.out.println("name=" + e.getName()
        + ", salary=" + e.getSalary()
        + ", hireDay=" + e.getHireDay());
```

Следует заметить, что данный пример программы состоит из двух классов: `Employee` и `EmployeeTest`, причем последний объявлен открытым с модификатором доступа `public`. Метод `main()` с описанными выше операторами содержится в классе `EmployeeTest`. Исходный код данной программы содержится в файле `EmployeeTest.java`, поскольку его имя должно совпадать с именем открытого класса. В исходном файле может быть только один класс, объявленный как `public`, а также любое количество классов, в объявлении которых данное ключевое слово отсутствует.

При компиляции исходного кода данной программы создаются два файла классов: `EmployeeTest.class` и `Employee.class`. Затем начинается выполнение программы, для чего интерпретатору байт-кода указывается имя класса, содержащего основной метод `main()` данной программы:

```
java EmployeeTest
```

Интерпретатор начинает обрабатывать метод `main()` из класса `EmployeeTest`. В результате выполнения кода создаются три новых объекта типа `Employee` и отображается их состояние.

Использование нескольких исходных файлов

Рассмотренная выше программа из листинга 4.2 состоит из двух классов в одном исходном файле. Многие программирующие на Java предпочитают размещать каждый класс в отдельном файле. Например, класс `Employee` можно разместить в файле `Employee.java`, а класс `EmployeeTest` — в файле `EmployeeTest.java`.

Существуют различные способы компиляции программы, код которой содержится в двух исходных файлах. Например, при вызове компилятора можно использовать шаблонный символ следующим образом:

```
javac Employee*.java
```

В результате все исходные файлы, имена которых соответствуют указанному шаблону, будут скомпилированы в файлы классов. С другой стороны, можно также ограничиться приведенной ниже командой.

```
javac EmployeeTest.java
```

Как ни странно, файл `Employee.java` также будет скомпилирован. Обнаружив, что в файле `EmployeeTest.java` используется класс `Employee`, компилятор Java начнет искать файл `Employee.class`. Если он его не найдет, то автоматически будет скомпилирован файл `Employee.java`. Более того, если файл `Employee.java` создан позже, чем существующий файл `Employee.class`, компилятор языка Java также *автоматически* выполнит повторную компиляцию и создаст исходный файл данного класса.



НА ЗАМЕТКУ! Если вы знакомы с утилитой `make`, доступной в Unix и других операционных системах, то такое поведение компилятора не станет для вас неожиданностью. Дело в том, что в компиляторе Java реализованы функциональные возможности этой утилиты.

Анализ класса `Employee`

Проанализируем класс `Employee`, начав с его методов. Изучая исходный код, нетрудно заметить, что в классе `Employee` реализованы один конструктор и четыре метода, перечисляемые ниже.

```
public Employee(String n, double s, int year, int month, int day)
public String getName()
public double getSalary()
public Date getHireDay()
public void raiseSalary(double byPercent)
```

Все методы этого класса объявлены как `public`, т.е. обращение к этим методам может осуществляться из любого класса. (Существуют четыре возможных уровня доступа. Все они рассматриваются в этой и следующей главах.)

Далее следует заметить, что в классе имеются три поля экземпляра для хранения данных, обрабатываемых в объекте `Employee`, как показано ниже.

```
private String name;
private double salary;
private Date hireDay;
```

Ключевое слово `private` означает, что к данным полям имеют доступ только методы самого класса `Employee`. Ни один внешний метод не может читать или записывать данные в эти поля.



НА ЗАМЕТКУ! Поля экземпляра могут быть объявлены как `public`, однако делать этого не следует. Ведь в этом случае любые компоненты программы (классы и методы) могут обратиться к открытым полям и видоизменить их содержимое, и, как показывает опыт, всегда найдется какой-нибудь код, который непременно воспользуется этими правами доступа в самый неподходящий момент. Поэтому мы настоятельно рекомендуем всегда закрывать доступ к полям экземпляра с помощью ключевого слова `private`.

И наконец, следует обратить внимание на то, что два из трех полей экземпляра сами являются объектами. В частности, поля `name` и `hireDay` являются ссылками на экземпляры классов `String` и `Date`. В ООП это довольно распространенное явление: одни классы часто содержат поля с экземплярами других классов.

Первые действия с конструкторами

Рассмотрим конструктор класса `Employee`.

```
public Employee(String n, double s, int year, int month, int day)
{
    name = n;
    salary = s;
    GregorianCalendar calendar = new GregorianCalendar(year, month - 1,
day);
    hireDay = calendar.getTime();
}
```

Как видите, имя конструктора совпадает с именем класса. Этот конструктор выполняется при создании объекта типа `Employee`, заполняя поля экземпляра заданными значениями. Например, при создании экземпляра класса `Employee` с помощью оператора

```
new Employee("James Bond", 100000, 1950, 1, 1);
```

поля экземпляра заполняются следующими значениями:

```
name = "James Bond";
salary = 100000;
hireDay = January 1, 1950;
```

У конструкторов имеется существенное отличие от других методов: конструктор можно вызывать только в сочетании с операцией `new`. Конструктор нельзя применить к существующему объекту, чтобы изменить информацию в его полях. Например, приведенный ниже вызов приведет к ошибке во время компиляции.

```
james.Employee("James Bond", 250000, 1950, 1, 1); // ОШИБКА!
```

Мы еще вернемся в этой главе к конструкторам. А до тех пор запомните следующее.

- Имя конструктора совпадает с именем класса.
- Класс может иметь несколько конструкторов.
- Конструктор может иметь один параметр или больше или же вообще их не иметь.
- Конструктор не возвращает никакого значения.
- Конструктор всегда вызывается совместно с операцией `new`.



НА ЗАМЕТКУ C++! Конструкторы в Java и C++ действуют одинаково. Но учтите, что все объекты в Java размещаются в динамической памяти и конструкторы вызываются только вместе с операцией `new`. Те, у кого имеется опыт программирования на C++, часто допускают следующую ошибку:

```
Employee number007("James Bond", 10000, 1950, 1, 1)
// допустимо в C++, но не в Java
```

Это выражение в C++ допустимо, а в Java — нет.



ВНИМАНИЕ! Будьте осмотрительны, чтобы не присваивать локальным переменным такие же имена, как и полям экземпляра. Например, приведенный ниже конструктор не сможет установить зарплату сотрудника.

```
public Employee(String n, double s, ...)
{
    String name = n; // ОШИБКА!
    double salary = s; // ОШИБКА!
}
```

В конструкторе объявляются *локальные* переменные `name` и `salary`. Доступ к этим переменным возможен только внутри конструктора. Они *скрывают* поля экземпляра с аналогичными именами. Некоторые программисты — например, авторы этой книги — могут написать такой код совершенно автоматически. Подобные ошибки очень трудно обнаружить. Поэтому нужно быть внимательными, чтобы не присваивать переменным имена полей экземпляра.

Явные и неявные параметры

Методы объекта имеют доступ ко всем его полям. Рассмотрим следующий метод:

```
public void raiseSalary(double byPercent)
{
    double raise = salary * byPercent / 100;
    salary += raise;
}
```

В этом методе устанавливается новое значение в поле `salary` того объекта, для которого этот метод вызывается. Например, следующий вызов данного метода:

```
number007.raiseSalary(5);
```

приведет к увеличению на **5%** значения в поле `number007.salary` объекта `number007`. Строго говоря, вызов данного метода приводит к выполнению следующих двух операторов:

```
double raise = number007.salary * 5 / 100;
number007.salary += raise;
```

У метода `raiseSalary()` имеются два параметра. Первый параметр, называемый *явным*, представляет собой ссылку на объект типа `Employee`, который указывается перед именем метода. А второй параметр — число, указанное в скобках после имени метода, — называется *явным*.

Нетрудно заметить, что явные параметры перечисляются в объявлении метода, например `double byPercent`. Неявный параметр в объявлении метода не приводится. В каждом методе ключевое слово `this` обозначает неявный параметр. По желанию метод `raiseSalary()` можно было бы переписать следующим образом:

```
public void raiseSalary(double byPercent)
{
    double raise = this.salary * byPercent / 100;
    this.salary += raise;
}
```

Некоторые предпочитают именно такой стиль программирования, поскольку в нем более отчетливо различаются поля экземпляра и локальные переменные.



НА ЗАМЕТКУ C++! В C++ методы обычно определяются за пределами класса, как показано ниже.

```
void Employee::raiseSalary(double byPercent) // В C++, но не в Java
{
```

```
    ...  
}
```

Если определить метод в классе, он автоматически станет встраиваемым.

```
class Employee  
{  
    ...  
    int getName() { return name; } // встраиваемая функция в C++  
}
```

В Java все методы определяются в пределах класса, но это не делает их встраиваемыми. Виртуальная машина Java анализирует, как часто производится обращение к методу, и принимает решение, должен ли метод быть встраиваемым. Динамический компилятор находит краткие, частые вызовы методов, которые не являются переопределенными, и оптимизирует их соответствующим образом.

Преимущества инкапсуляции

Рассмотрим далее очень простые методы `getName()`, `getSalary()` и `getHireDay()` из класса `Employee`. Их исходный код приведен ниже.

```
public String getName()  
{  
    return name;  
}  
  
public double getSalary()  
{  
    return salary;  
}  
  
public Date getHireDay()  
{  
    return hireDay;  
}
```

Они служат характерным примером *методов доступа*. А поскольку они лишь возвращают значения полей экземпляра, то иногда их еще называют *методами доступа к полям*. Но не проще ли было сделать поля `name`, `salary` и `hireDay` открытыми для доступа (т.е. объявить их как `public`) и не создавать отдельные методы доступа к ним?

Но дело в том, что поле `name` доступно лишь для чтения. После того как значение этого поля будет установлено конструктором, ни один метод не сможет его изменить. А это дает гарантию, что данные, хранящиеся в этом поле, не будут искажены.

Поле `salary` доступно не только для чтения, но и для записи, но изменить значение в нем способен только метод `raiseSalary()`. И если окажется, что в поле записано неверное значение, то отладить нужно будет только один метод. Если бы поле `salary` было открытым, причина ошибки могла бы находиться где угодно.

Иногда требуется иметь возможность читать и видоизменять содержимое поля. Для этого придется реализовать в составе класса следующие три компонента.

- **Закрытое** (`private`) поле данных.
- **Открытый** (`public`) метод доступа.
- **Открытый** (`public`) модифицирующий метод.

Конечно, сделать это намного труднее, чем просто объявить открытым единственное поле данных. Но такой подход дает немалые преимущества. Во-первых, внутреннюю реализацию класса можно изменять совершенно независимо от других классов. Допустим, что имя и фамилия сотрудника хранятся отдельно:

```
String firstName;
String lastName;
```

Тогда в методе `getName()` возвращаемое значение должно быть сформировано следующим образом:

```
firstName + " " + lastName
```

И такое изменение оказывается совершенно незаметным для остальной части программы. Разумеется, методы доступа и модифицирующие методы должны быть переработаны, чтобы учесть новое представление данных. Но это дает еще одно преимущество: модифицирующие методы могут выполнять проверку ошибок, тогда как при непосредственном присваивании открытому полю некоторого значения ошибки не выявляются. Например, в методе `setSalary()` можно проверить, не стала ли зарплата отрицательной величиной.



ВНИМАНИЕ! Будьте осмотрительны при создании методов доступа, возвращающих ссылки на изменяемый объект. Создавая класс `Employee`, мы нарушили это правило: метод `getHireDay()` возвращает объект класса `Date`, как показано ниже.

```
class Employee
{
    private Date hireDay;
    ...
    public Date getHire();
    {
        return hireDay;
    }
    ...
}
```

Это нарушает принцип инкапсуляции! Рассмотрим следующий пример неверного кода:

```
Employee harry = ...;
Date d = harry.getHireDay();
double tenYearsInMilliseconds = 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long) tenYearsInMilliseconds);
// значение в объекте изменено
```

Причина ошибки в этом коде проста: обе ссылки, `d` и `harry.hireDay`, делаются на один и тот же объект (рис. 4.5). В результате применения модифицирующих методов к объекту `d` автоматически изменяется открытое состояние объекта сотрудника типа `Employee`!

Чтобы вернуть ссылку на изменяемый объект, его нужно сначала клонировать. Клон — это точная копия объекта, находящаяся в другом месте памяти. Подробнее о клонировании речь пойдет в главе 6. Ниже приведен исправленный код.

```
class Employee
{
    . . .
    public Date getHireDay()
```

```

    {
        return hireDay.clone();
    }
    . . .
}

```

В качестве эмпирического правила пользуйтесь методом `clone()`, если вам нужно скопировать изменяемое поле данных.

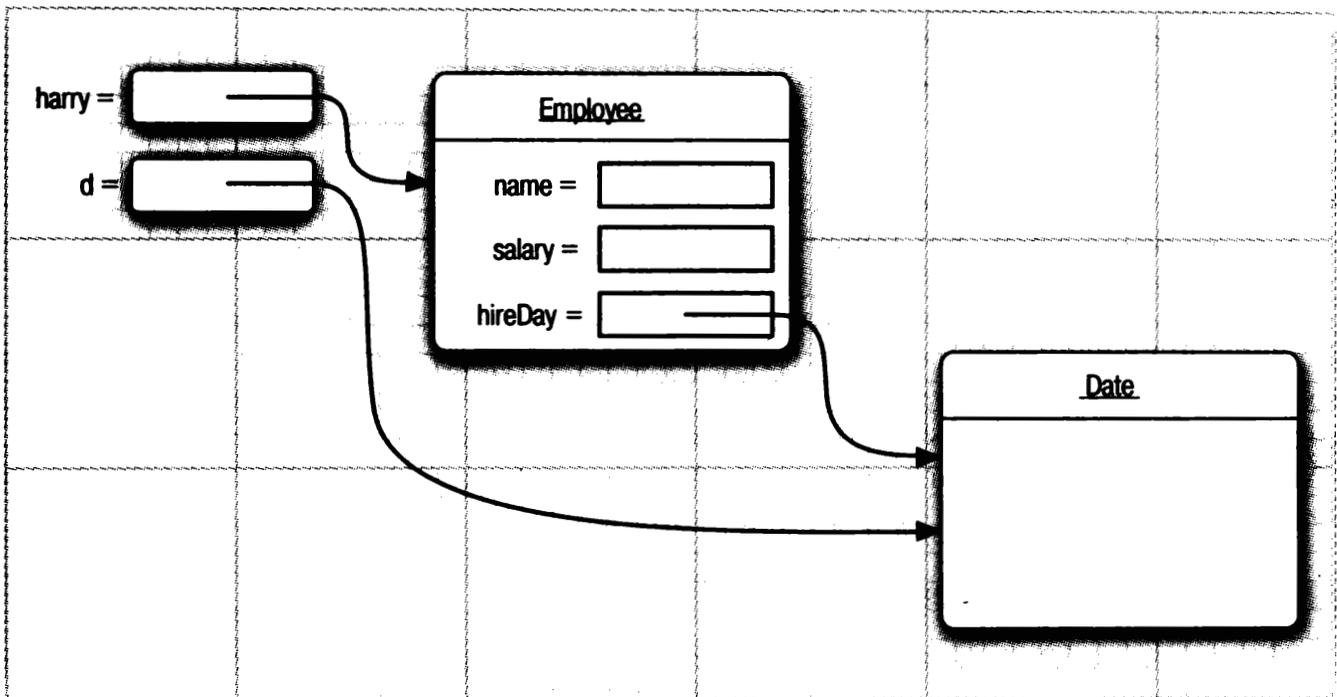


Рис. 4.5. Ссылки на изменяемый объект

Привилегии доступа к данным в классе

Как вам должно быть уже известно, метод имеет доступ к закрытым данным того объекта, для которого он вызывается. Но он также может обращаться к закрытым данным *всех* объектов своего класса! Рассмотрим в качестве примера метод `equals()`, сравнивающий два объекта типа `Employee`.

```

class Employee
{
    . . .
    public boolean equals(Employee other)
    {
        return name.equals(other.name);
    }
}

```

Типичный вызов этого метода выглядит следующим образом:

```

if (harry.equals(boss)) ...

```

Этот метод имеет доступ к закрытым полям объекта `harry`, что и не удивительно. Но он также имеет доступ к полям объекта `boss`. И это вполне объяснимо, поскольку `boss` — объект типа `Employee`, а методы, принадлежащие классу `Employee`, могут обращаться к закрытым полям *любого* объекта этого класса.



НА ЗАМЕТКУ C++! В языке C++ действует такое же правило. Метод имеет доступ к переменным и функциям любого объекта своего класса.

Закрытые методы

При реализации класса все поля данных делаются закрытыми, поскольку предоставлять к ним доступ из других классов весьма рискованно. А как поступить с методами? Для взаимодействия с другими объектами требуются открытые методы. Но в ряде случаев для вычислений нужны вспомогательные методы. Как правило, эти вспомогательные методы не являются частью интерфейса, поэтому указывать при их объявлении ключевое слово `public` нет необходимости. И чаще всего они объявляются как `private`, т.е. как закрытые. Чтобы сделать метод закрытым, достаточно изменить ключевое слово `public` на `private` в его объявлении.

Сделав метод закрытым, совсем не обязательно его сохранять при переходе к другой реализации. Такой метод труднее реализовать, а возможно, он окажется вообще ненужным, если изменится представление данных, что, в общем, несущественно. Важнее другое: до тех пор, пока метод является закрытым (`private`), разработчики класса могут быть уверены в том, что он никогда не будет использован в операциях, выполняемых за пределами класса, а следовательно, они могут просто удалить его. Если же метод является открытым (`public`), его нельзя просто так опустить, поскольку от него может зависеть другой код.

Неизменяемые поля экземпляра

Поля экземпляра можно объявить с помощью ключевого слова `final`. Такое поле должно инициализироваться при создании объекта, т.е. необходимо гарантировать, что значение поля будет установлено по завершении каждого конструктора. После этого его значение изменить уже нельзя. Например, поле `name` из класса `Employee` можно объявить неизменяемым, поскольку после создания объекта оно уже не изменится, а метода `setName()` для этого не существует.

```
class Employee
{
    ...
    private final String name;
}
```

Модификатор `final` удобно применять при объявлении полей простых типов или полей, типы которых задаются *неизменяемыми классами*. Неизменяемым называется такой класс, методы которого не позволяют изменить состояние объекта. Например, неизменяемым является класс `String`. Если класс допускает изменения, то ключевое слово `final` может стать источником недоразумений. Рассмотрим следующий оператор:

```
private final Date hiredate;
```

Он означает, что переменная `hiredate` не изменяется после создания объекта. Но это не означает, что состояние объекта, на который ссылается переменная, остается неизменным. В любой момент можно вызвать метод `setTime()`.

Статические поля и методы

При объявлении метода `main()` во всех рассматривавшихся до сих пор примерах программ использовался модификатор `static`. Рассмотрим назначение этого модификатора доступа.

Статические поля

Поле с модификатором доступа `static` существует в одном экземпляре для всего класса. Но если поле не статическое, то каждый объект содержит его копию. Допустим, требуется присвоить уникальный идентификационный номер каждому сотруднику. Для этого достаточно добавить в класс `Employee` поле `id` и статическое поле `nextId`, как показано ниже.

```
class Employee
{
    ...
    private int id;
    private static int nextId = 1;
}
```

Теперь у каждого объекта типа `Employee` имеется свое поле `id`, а также поле `nextId`, которое одновременно принадлежит всем экземплярам данного класса. Иными словами, если существует тысяча объектов типа `Employee`, то в них есть тысяча полей `id`: по одному на каждый объект. В то же время существует только один экземпляр статического поля `nextId`. Даже если не создано ни одного объекта типа `Employee`, статическое поле `nextId` все равно существует. Оно принадлежит классу, а не конкретному объекту.



НА ЗАМЕТКУ! В большинстве объектно-ориентированных языков статические поля называются полями класса. Термин *статический* унаследован как малозначащий пережиток от языка C++.

Реализуем следующий простой метод:

```
public void setId()
{
    id = nextId;
    nextId++;
}
```

Допустим, требуется задать идентификационный номер объекта `harry` следующим образом:

```
harry.setId();
```

Затем устанавливается текущее значение в поле `id` объекта `harry`, а значение статического поля `nextId` увеличивается на единицу, как показано ниже.

```
harry.id = Employee.nextId;
Employee.nextId++;
```

Статические константы

Статические переменные используются довольно редко. В то же время статические константы применяются намного чаще. Например, статическая константа, задающая число π , определяется в классе `Math` следующим образом:

```
public class Math
{
    ...
    public static final double PI = 3.14159265358979323846;
    ...
}
```

Обратиться к этой константе в программе можно с помощью выражения `Math.PI`. Если бы ключевое слово `static` было пропущено, константа `PI` была бы обычным полем экземпляра класса `Math`. Это означает, что для доступа к такой константе нужно было бы создать объект типа `Math`, причем каждый такой объект имел бы свою копию константы `PI`.

Еще одной часто употребляемой является статическая константа `System.out`. Она объявляется в классе `System` следующим образом:

```
public class System
{
    ...
    public static final PrintStream out = ...;
    ...
}
```

Как уже упоминалось не раз, делать поля открытыми в коде не рекомендуется, поскольку любой объект сможет изменить их значения. Но открытыми константами (т.е. полями, объявленными с ключевым словом `final`) можно пользоваться смело. Так, если поле `out` объявлено как `final`, ему нельзя присвоить другой поток вывода:

```
System.out = new PrintStream(...); // ОШИБКА: поле out изменить нельзя!
```



НА ЗАМЕТКУ! Анализируя исходный код класса `System`, можно обнаружить в нем метод `setOut()`, позволяющий присвоить полю `System.out` другой поток. Как же этот метод может изменить переменную, объявленную как `final`? Дело в том, что метод `setOut()` является собственным, т.е. он реализован средствами конкретной платформы, а не языка Java. Собственные методы способны обходить механизмы контроля, предусмотренные в Java. Это очень необычный обходной прием, который ни в коем случае не следует применять в своих программах.

Статические методы

Статическими называются методы, которые не оперируют объектами. Например, метод `pow()` из класса `Math` является статическим. Выражение `Math.pow(x, a)` вычисляет x^a . При выполнении этого метода не используется ни один из экземпляров класса `Math`. Иными словами, у него нет неявного параметра `this`. Это означает, что в статических методах не используется текущий объект по ссылке `this`. (А в нестатических методах неявный параметр `this` ссылается на текущий объект; см. раздел “Явные и неявные параметры” выше в этой главе).

В связи с тем что статические методы не оперируют объектами, из них нельзя получить доступ к полям экземпляра. Но статические методы имеют доступ к статическим полям класса. Ниже приведен пример статического метода.

```
public static int getNextId()
{
    return nextId; // вернуть статическое поле
}
```

Чтобы вызвать этот метод, нужно указать имя класса следующим образом:

```
int n = Employee.getNextId();
```

Можно ли пропустить ключевое слово `static` при обращении к этому методу? Можно, но тогда для его вызова потребуется ссылка на объект типа `Employee`.



НА ЗАМЕТКУ! Для вызова статического метода можно использовать и объекты. Так, если `harry` — это объект типа `Employee`, то вместо вызова `Employee.getNextId()` можно сделать вызов `harry.getNextId()`. Но такое обозначение усложняет восприятие программы, поскольку для вычисления результата метод `getNextId()` не обращается к объекту `harry`. Поэтому для вызова статических методов рекомендуется использовать имена их классов, а не объекты.

Статические методы следует применять в двух случаях.

- Когда методу не требуется доступ к данным о состоянии объекта, поскольку все необходимые параметры задаются явно (например, в методе `Math.pow()`).
- Когда методу требуется доступ лишь к статическим полям класса (например, при вызове метода `Employee.getNextId()`).



НА ЗАМЕТКУ C++! Статические поля и методы в Java и C++, по существу, отличаются только синтаксически. В C++ для доступа к статическому полю или методу, находящемуся вне области действия, можно использовать операцию `::`, например `Math::PI`. Любопытно происхождение термина *статический*. Сначала ключевое слово `static` было внедрено в C для обозначения локальных переменных, которые не уничтожались при выходе из блока. В этом контексте термин *статический* имеет смысл: переменная продолжает существовать после выхода из блока, а также при повторном входе в него. Затем термин *статический* приобрел в C второе значение для глобальных переменных и функций, к которым нельзя получить доступ из других файлов. Ключевое слово `static` было просто использовано повторно, чтобы не вводить новое. И наконец, в C++ это ключевое слово было применено в третий раз, получив совершенно новую интерпретацию. Оно обозначает переменные и методы, принадлежащие классу, но не принадлежащие ни одному из объектов этого класса. Именно этот смысл ключевое слово `static` имеет и в Java.

Фабричные методы

Рассмотрим еще одно применение статических методов. В классе `NumberFormat` применяются так называемые *фабричные методы* для создания объектов, соответствующих различным стилям форматирования, как показано ниже.

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // вывести $0.10
System.out.println(percentFormatter.format(x)); // вывести 10%
```

Почему же не использовать для этой цели конструктор? На то есть две причины.

- Конструктору нельзя присвоить произвольное имя. Его имя всегда должно совпадать с именем класса. Так, в классе `NumberFormat` имеет смысл применять разные имена для разных типов форматирования.
- При использовании конструктора тип объекта фиксирован. Если же применяются фабричные методы, они возвращают объект типа `DecimalFormat`, наследующий свойства из класса `NumberFormat`. (Подробнее вопросы наследования будут обсуждаться в главе 5.)

Метод `main()`

Отметим, что статические методы можно вызывать, не имея ни одного объекта данного класса. Например, для того чтобы вызвать метод `Math.pow()`, объекты типа `Math` не нужны. По той же причине метод `main()` объявляется как статический:

```
public class Application
{
    public static void main(String[] args)
    {
        // здесь создаются объекты
    }
}
```

Метод `main()` не оперирует никакими объектами. На самом деле при запуске программы еще нет никаких объектов. Статический метод `main()` выполняется и конструирует объекты, необходимые программе.



СОВЕТ. Каждый класс может содержать метод `main()`. С его помощью удобно организовать блочное тестирование классов. Например, метод `main()` можно добавить в класс `Employee` следующим образом:

```
class Employee
{
    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        GregorianCalendar calendar =
            new GregorianCalendar(year, month - 1, day);
        hireDay = calendar.getTime();
    }
    . . .
    public static void main(String[] args) // блочный тест
    {
        Employee e = new Employee ("Romeo", 50000, 2003, 3, 31);
        e.raiseSalary(10);
        System.out.println(e.getName() + " " + e.getSalary());
    }
    . . .
}
```

Если требуется протестировать только класс `Employee`, то для этого достаточно ввести команду **java Employee**. А если класс `Employee` является частью крупного приложения, то последнее можно запустить на выполнение по команде **java Application**. В этом случае метод `main()` из класса `Employee` вообще не будет выполнен.

Программа, приведенная в листинге 4.3, содержит простую версию класса `Employee` со статическим полем `nextId` и статическим методом `getNextId()`. В этой программе массив заполняется тремя объектами типа `Employee`, а затем выводятся данные о сотрудниках. И наконец, для демонстрации статического метода на экран выводится очередной доступный идентификационный номер.

Следует заметить, что в классе `Employee` имеется также статический метод `main()` для блочного тестирования. Попробуйте выполнить метод `main()` по командам `java Employee` и `java StaticTest`.

Листинг 4.3. Исходный код из файла `StaticTest/StaticTest.java`

```
1  /**
2   * В этой программе демонстрируются статические методы
3   * @version 1.01 2004-02-19
4   * @author Cay Horstmann
5   */
6  public class StaticTest
7  {
8      public static void main(String[] args)
9      {
10         // заполнить массив staff тремя объектами типа Employee
11         Employee[] staff = new Employee[3];
12
13         staff[0] = new Employee("Tom", 40000);
14         staff[1] = new Employee("Dick", 60000);
15         staff[2] = new Employee("Harry", 65000);
16
17         // вывести данные обо всех объектах типа Employee
18         for (Employee e : staff)
19         {
20             e.setId();
21             System.out.println("name=" + e.getName() + ",id=" +
22                 e.getId() + ",salary=" + e.getSalary());
23         }
24
25         int n = Employee.getNextId(); // вызвать статический метод
26         System.out.println("Next available id=" + n);
27     }
28 }
29
30 class Employee
31 {
32     private static int nextId = 1;
33
34     private String name;
35     private double salary;
36     private int id;
37
38     public Employee(String n, double s)
39     {
40         name = n;
41         salary = s;
42         id = 0;
43     }
44     public String getName()
45     {
```

```
46     return name;
47 }
48
49 public double getSalary()
50 {
51     return salary;
52 }
53
54 public int getId()
55 {
56     return id;
57 }
58
59 public void setId()
60 {
61     id = nextId; // установить следующий доступный идентификатор
62     nextId++;
63 }
64
65 public static int getNextId()
66 {
67     return nextId; // вернуть статическое поле
68 }
69
70 public static void main(String[] args) // выполнить блочный тест
71 {
72     Employee e = new Employee("Harry", 50000);
73     System.out.println(e.getName() + " " + e.getSalary());
74 }
75 }
```

Параметры методов

Рассмотрим термины, которые употребляются для описания способа передачи параметров методам (или функциям) в языках программирования. Термин *вызов по значению* означает, что метод получает значение, переданное ему из вызывающей части программы. Напротив, *вызов по ссылке* означает, что метод получает из вызывающей части программы *местоположение* переменной. Таким образом, метод может *видоизменить* значение переменной, передаваемой по ссылке, но не переменной, передаваемой по значению. Фраза “вызов по...” относится к стандартной компьютерной терминологии, описывающей способ передачи параметров в различных языках программирования, а не только в Java. (На самом деле существует еще и третий способ передачи параметров — *вызов по имени*, представляющий в основном исторический интерес, поскольку он был применен в языке Algol, который относится к числу самых старых языков программирования высокого уровня.)

В Java всегда используется *только* вызов по значению. Это означает, что метод получает копии значений всех своих параметров. По этой причине метод не может видоизменить содержимое ни одной из переменных, передаваемых ему в качестве параметров.

Рассмотрим для примера следующий вызов:

```
double percent = 10;
harry.raiseSalary(percent);
```

Каким бы образом ни был реализован метод, после его вызова значение переменной `percent` все равно останется равным **10**.

Проанализируем эту ситуацию подробнее. Допустим, в методе предпринимается попытка утроить значение параметра, как показано ниже.

```
public static void tripleValue(double x); // не работает!
{
    x = 3 * x;
}
```

Если вызвать этот метод следующим образом:

```
double percent = 10;
tripleValue(percent);
```

такой прием не сработает. После вызова метода значение переменной `percent` по-прежнему остается равным **10**. В данном случае происходит следующее.

1. Переменная `x` инициализируется копией значения параметра `percent` (т.е. числом **10**).
2. Значение переменной `x` утраивается, и теперь оно равно **30**. Но значение переменной `percent` по-прежнему остается равным **10** (рис. 4.6).
3. Метод завершает свою работу, и его переменный параметр `x` больше не используется.

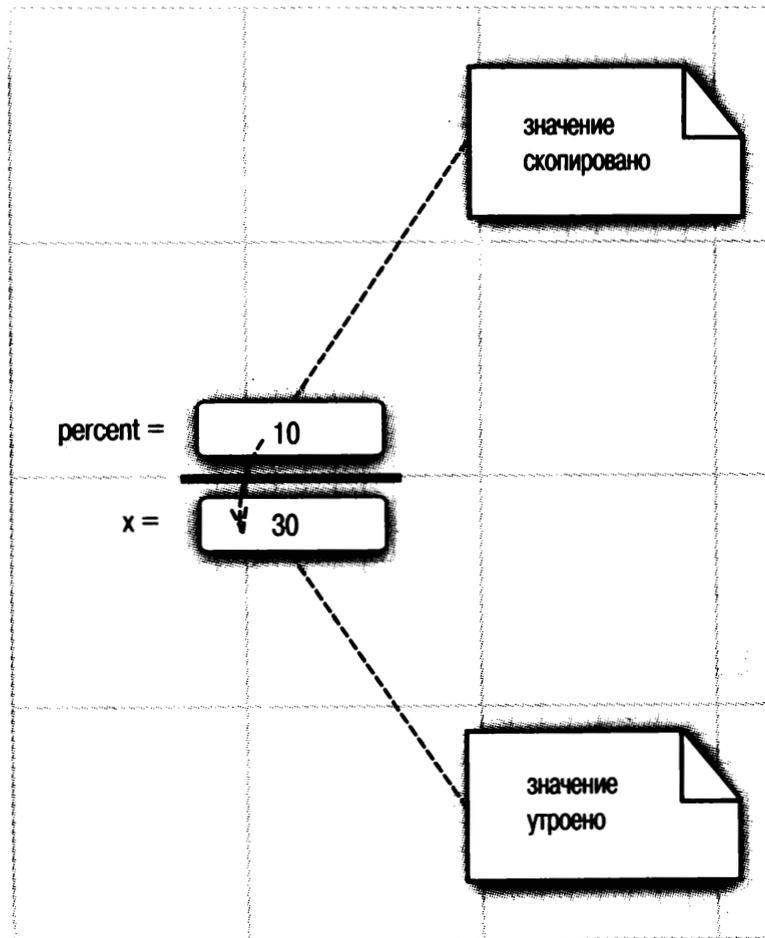


Рис. 4.6. Видоизменение значения в вызываемом методе не оказывает никакого влияния на параметр, передаваемый из вызывающей части программы

Но существуют два следующих типа параметров методов.

- Прimitивные типы (т.е. числовые и логические значения).
- Ссылки на объекты.

Как было показано выше, методы не могут видоизменить параметры примитивных типов. Совсем иначе дело обстоит с объектами. Нетрудно реализовать метод, утраивающий зарплату сотрудников, следующим образом:

```
public static void tripleSalary(Employee x) // сработает!  
{  
    x.raiseSalary(200);  
}
```

При выполнении следующего фрагмента кода происходят перечисленные ниже действия.

```
harry = new Employee(...);  
tripleSalary(harry);
```

1. Переменная *x* инициализируется копией значения переменной *harry*, т.е. ссылкой на объект.
2. Метод `raiseSalary()` применяется к объекту по этой ссылке. В частности, объект типа `Employee`, доступный по ссылкам *x* и *harry*, получает сумму зарплаты сотрудников, увеличенную на **200%**.
3. Метод завершает свою работу, и его параметр *x* больше не используется. Разумеется, переменная *harry* продолжает ссылаться на объект, в котором зарплата увеличена втрое (рис. 4.7).

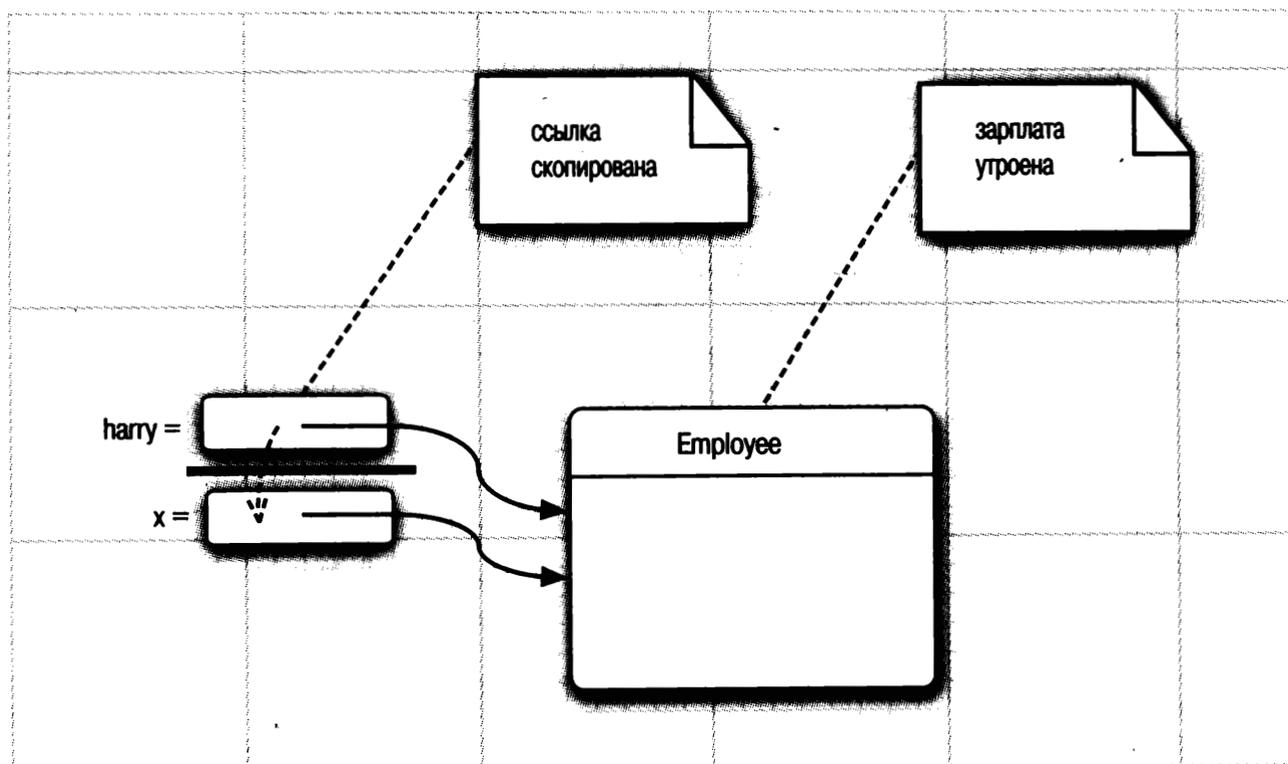


Рис. 4.7. Если параметр ссылается на объект, последний может быть видоизменен

Как видите, реализовать метод, изменяющий состояние объекта, передаваемого как параметр, совсем не трудно. В действительности такие изменения вносятся очень часто по следующей простой причине: метод получает копию ссылки на объект, поэтому копия и оригинал ссылки указывают на один и тот же объект.

Во многих языках программирования (в частности, C++ и Pascal) предусмотрены два способа передачи параметров: вызов по значению и вызов по ссылке. Некоторые программисты (и, к сожалению, даже авторы некоторых книг) утверждают, что в Java при передаче объектов используется только вызов по ссылке. Но это совсем не так. Для того чтобы развеять это бытующее заблуждение, обратимся к конкретному примеру. Ниже приведен метод, выполняющий обмен двух объектов типа `Employee`.

```
public static void swap(Employee x, Employee y) // не работает!
{
    Employee temp = x;
    x = y;
    y = temp;
}
```

Если бы в Java для передачи объектов в качестве параметров использовался вызов по ссылке, этот метод действовал бы следующим образом:

```
Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// ссылается ли теперь переменная a на Bob, а переменная b — на Alice?
```

Но на самом деле этот метод не меняет местами ссылки на объекты, хранящиеся в переменных `a` и `b`. Сначала параметры `x` и `y` метода `swap()` инициализируются *копиями* этих ссылок, а затем эти копии меняются местами в данном методе, как показано ниже.

```
// переменная x ссылается на Alice, а переменная y — на Bob
Employee temp = x;
x = y;
y = temp;
// теперь переменная x ссылается на Bob, а переменная y — на Alice
```

В конце концов, следует признать, что все было напрасно. По завершении работы данного метода переменные `x` и `y` уничтожаются, а исходные переменные `a` и `b` продолжают ссылаться на прежние объекты (рис. 4.8).

Таким образом, в Java для передачи объектов не применяется вызов по ссылке. Вместо этого ссылки на объекты передаются *по значению*. Ниже описано, что может и чего он не может метод делать со своими параметрами.

- Метод не может изменять параметры примитивных типов (т.е. числовые и логические значения).
- Метод может изменять *состояние* объекта, передаваемого в качестве параметра.
- Метод не может делать в своих параметрах ссылки на новые объекты.

Все эти положения демонстрируются в примере программы из листинга 4.4. Сначала в ней предпринимается безуспешная попытка утроить значение числового параметра.

```
Testing tripleValue:
(Тестирование метода tripleValue())
```

Before: percent=10.0
 (До выполнения)
 End of method: x=30.0
 (В конце метода)
 After: percent=10.0
 (После выполнения)

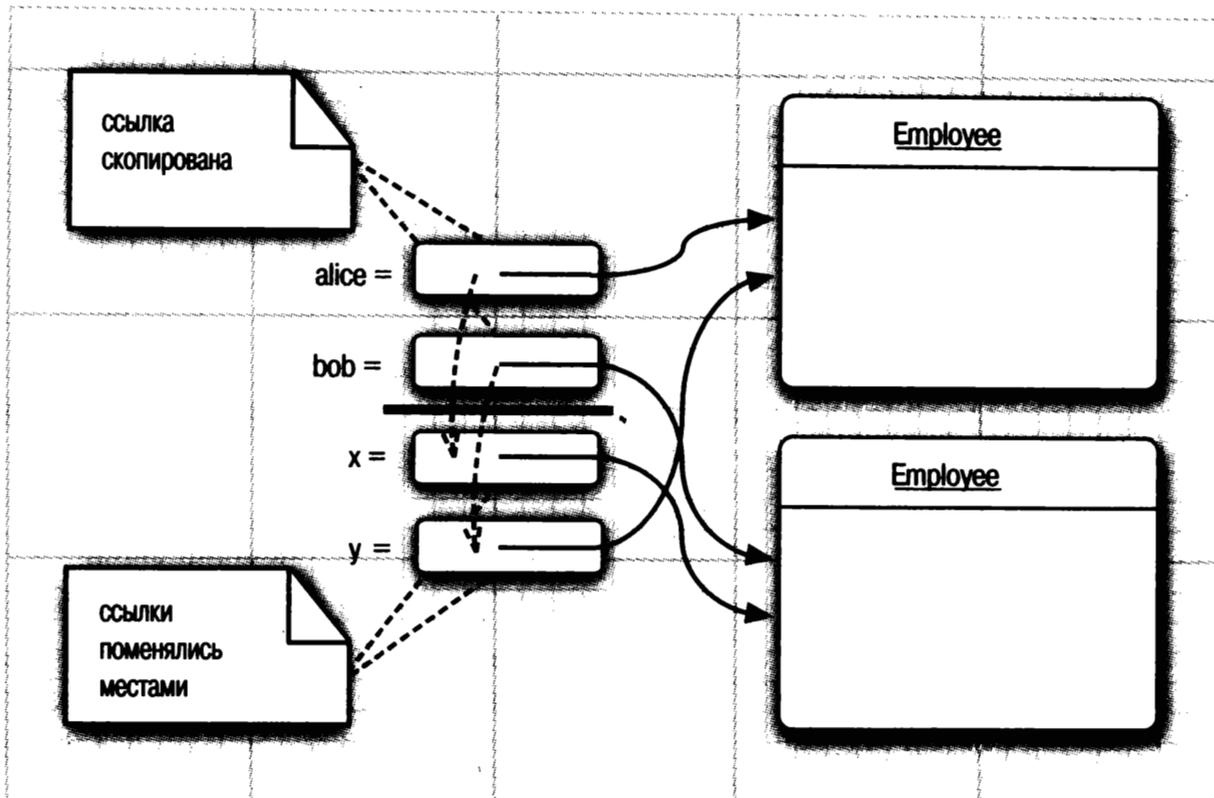


Рис. 4.8. Перестановка ссылок в вызываемом методе не имеет никаких последствий для вызывающей части программы

Затем в программе успешно утраивается зарплата сотрудника.

```
Testing tripleSalary:
(Тестирование метода tripleSalary())
Before: salary=50000.0
End of method: salary=150000.0
After: salary=150000.0
```

После выполнения метода состояние объекта, на который ссылается переменная `harry`, изменяется. Ничего невероятного здесь нет, поскольку в данном методе состояние объекта было видоизменено по копии ссылки на него. И наконец, в программе демонстрируется безрезультатность работы метода `swap()`.

```
Testing swap:
(Тестирование метода swap())
Before: a=Alice
Before: b=Bob
End of method: x=Bob
End of method: y=Alice
After: a=Alice
After: b=Bob
```

Как видите, переменные параметры `x` и `y` меняются местами, но исходные переменные `a` и `b` остаются без изменения.

Листинг 4.4. Исходный код из файла ParamTest/ParamTest.java

```
1  /**
2   * В этой программе демонстрируется передача параметров в Java
3   * @version 1.00 2000-01-27
4   * @author Cay Horstmann
5   */
6  public class ParamTest
7  {
8      public static void main(String[] args)
9      {
10         /*
11          * Тест 1: методы не могут видоизменять числовые параметры
12          */
13         System.out.println("Testing tripleValue:");
14         double percent = 10;
15         System.out.println("Before: percent=" + percent);
16         tripleValue(percent);
17         System.out.println("After: percent=" + percent);
18
19         /*
20          * Тест 2: методы могут изменять состояние объектов,
21          * передаваемых в качестве параметров
22          */
23         System.out.println("\nTesting tripleSalary:");
24         Employee harry = new Employee("Harry", 50000);
25         System.out.println("Before: salary=" + harry.getSalary());
26         tripleSalary(harry);
27         System.out.println("After: salary=" + harry.getSalary());
28
29         /*
30          * Тест 3: методы не могут присоединять новые объекты
31          * к объектным параметрам
32          */
33         System.out.println("\nTesting swap:");
34         Employee a = new Employee("Alice", 70000);
35         Employee b = new Employee("Bob", 60000);
36         System.out.println("Before: a=" + a.getName());
37         System.out.println("Before: b=" + b.getName());
38         swap(a, b);
39         System.out.println("After: a=" + a.getName());
40         System.out.println("After: b=" + b.getName());
41     }
42
43     public static void tripleValue(double x) // не работает!
44     {
45         x = 3 * x;
46         System.out.println("End of method: x=" + x);
47     }
48
49     public static void tripleSalary(Employee x) // работает!
50     {
51         x.raiseSalary(200);
52         System.out.println("End of method: salary=" + x.getSalary());
53     }
54
55     public static void swap(Employee x, Employee y)
```

```
56 {
57     Employee temp = x;
58     x = y;
59     y = temp;
60     System.out.println("End of method: x=" + x.getName());
61     System.out.println("End of method: y=" + y.getName());
62 }
63 }
64 class Employee // упрощенный класс Employee
65 {
66     private String name;
67     private double salary;
68
69     public Employee(String n, double s)
70     {
71         name = n;
72         salary = s;
73     }
74
75     public String getName()
76     {
77         return name;
78     }
79
80     public double getSalary()
81     {
82         return salary;
83     }
84
85     public void raiseSalary(double byPercent)
86     {
87         double raise = salary * byPercent / 100;
88         salary += raise;
89     }
90 }
```

С **НА ЗАМЕТКУ C++!** В C++ применяется как вызов по значению, так и вызов по ссылке. Например, можно без особого труда реализовать методы `void tripleValue(double& x)` или `void swap(Employee& x, Employee& y)`. Эти методы видоизменяют свои ссылочные параметры.

Конструирование объектов

Ранее было показано, как писать простые конструкторы, определяющие начальные состояния объектов. Но конструирование объектов — очень важная операция, и поэтому в Java предусмотрены самые разные механизмы написания конструкторов. Все эти механизмы рассматриваются ниже.

Перегрузка

Напомним, что в классе `GregorianCalendar` предусмотрено несколько конструкторов. Создать объект этого класса можно двумя способами:

```
GregorianCalendar today = new GregorianCalendar();
```

или

```
GregorianCalendar deadline =  
    new GregorianCalendar(2099, Calendar, DECEMBER, 31);
```

Оба способа носят общее название *перегрузки*. О перегрузке говорят в том случае, если у нескольких методов (в данном случае нескольких конструкторов) имеются одинаковые имена, но разные параметры. Компилятор должен сам определить, какой метод вызвать, сравнивая типы параметров, определяемых при объявлении методов, с типами значений, указанных при вызове методов. Если ни один из методов не соответствует вызову или же если одному вызову одновременно соответствует несколько вариантов, возникает ошибка компиляции. (Этот процесс называется *разрешением перегрузки*.)



НА ЗАМЕТКУ! В Java можно перегрузить любой метод, а не только конструкторы. Следовательно, для того чтобы полностью описать метод, нужно указать его имя и типы параметров. Подобное написание называется *сигнатурой* метода. Например, в классе `String` имеются четыре открытых метода под названием `indexOf()`. Они имеют следующие сигнатуры:

```
indexOf(int)  
indexOf(int, int)  
indexOf(String)  
indexOf(String, int)
```

Тип возвращаемого методом значения не входит в его сигнатуру. Следовательно, нельзя создать два метода, имеющих одинаковые имена и типы параметров и отличающихся лишь типом возвращаемого значения.

Инициализация полей по умолчанию

Если значение поля в конструкторе явно не задано, ему автоматически присваивается значение по умолчанию: числам — нули, логическим переменным — логическое значение `false`, а ссылкам на объект — пустое значение `null`. Но полагаться на действия по умолчанию не следует. Если поля инициализируются неявно, программа становится менее понятной.



НА ЗАМЕТКУ! Между полями и локальными переменными имеется существенная разница. Локальные переменные всегда должны явно инициализироваться в методе. Но если поле не инициализируется в классе явно, то ему автоматически присваивается значение, задаваемое по умолчанию (`0`, `false` или `null`).

Рассмотрим в качестве примера класс `Employee`. Допустим, в конструкторе инициализация значений некоторых полей не задана. По умолчанию поле `salary` должно инициализироваться нулем, а поля `name` и `hireDay` — пустым значением `null`. Но при вызове методов `getName()` или `getHireDay()` пустая ссылка `null` может оказаться совершенно нежелательной, как показано ниже.

```
Date h = harry.getHireDay();  
calendar.setTime(h); // Если h = null, генерируется исключение
```

Конструктор без аргументов

Многие классы содержат конструктор без аргументов, создающий объект, состояние которого устанавливается соответствующим образом по умолчанию. В качестве примера ниже приведен конструктор без аргументов для класса `Employee`.

```
public Employee()
{
    name = "";
    salary = 0;
    hireDay = new Date();
}
```

Если в классе совсем не определены конструкторы, то автоматически создается конструктор без аргументов. В этом конструкторе *всем* полям экземпляра присваиваются их значения, предусмотренные по умолчанию. Так, все числовые значения, содержащиеся в полях экземпляра, окажутся равными нулю, логические переменные — `false`, а объектные переменные — `null`.

Если же в классе есть хотя бы один конструктор и явно не определен конструктор без параметров, то создавать объекты, не предоставляя аргументы, нельзя. Например, у класса `Employee` из листинга 4.2 имеется один следующий конструктор:

```
Employee(String name, double salary, int y, int m, int d)
```

В этой версии данного класса нельзя создать объект, поля которого принимали бы значения по умолчанию. Иными словами, следующий вызов приведет к ошибке:

```
e = new Employee();
```



ВНИМАНИЕ! Следует иметь в виду, что конструктор без аргументов вызывается только в том случае, если в классе не определены другие конструкторы. Если же в классе имеется хотя бы один конструктор с параметрами и требуется создавать экземпляры класса с помощью приведенного ниже выражения, следует явно определить конструктор без параметров.

```
new ИмяКласса()
```

Разумеется, если значения по умолчанию во всех полях вполне устраивают, можно создать следующий конструктор:

```
public ИмяКласса()
{
}
}
```

Явная инициализация полей

Конструкторы можно перегружать в классе, как любые другие методы, а следовательно, задать начальное состояние полей его экземпляров можно несколькими способами. Каждое поле экземпляра следует всегда снабжать осмысленными значениями независимо от вызова конструктора.

В определении класса имеется возможность присвоить каждому полю соответствующее значение, как показано ниже.

```
class Employee
{
    ...
    private String name = "";
}
```

Это присваивание выполняется до вызова конструктора. Такой подход оказывается особенно полезным в тех случаях, когда требуется, чтобы поле имело конкретное значение независимо от вызова конструктора класса. При инициализации поля совсем не обязательно использовать константу. Ниже приведен пример, в котором поле инициализируется с помощью вызова метода.

```
class Employee
{
    private static int nextId;
    private int id = assignId();
    . . .
    private static int assignId()
    {
        int r = nextId;
        nextId++;
        return r;
    }
    . . .
}
```



НА ЗАМЕТКУ C++! В C++ нельзя инициализировать поля экземпляра непосредственно в описании класса. Значения всех полей должны задаваться в конструкторе. Но в C++ имеется синтаксическая конструкция, называемая *списком инициализации*, как показано ниже.

```
Employee::Employee(String n, double s, int y, int m, int d) // C++
: name(n),
  salary(s),
  hireDay(y, m, d)
{
}
```

В C++ эта специальная синтаксическая конструкция служит для вызова конструкторов полей. А в Java поступать подобным образом нет никакой необходимости, поскольку объекты не могут содержать подобъекты, но разрешается иметь только ссылки на них.

Имена параметров

Создавая даже элементарный конструктор (а большинство из них таковыми и являются), трудно выбрать подходящие имена для его параметров. Обычно в качестве имен параметров служат отдельные буквы, как показано ниже.

```
public Employee(String n, double s)
{
    name = n;
    salary = s;
}
```

Но недостаток такого подхода заключается в том, что, читая программу, невозможно понять, что же означают параметры `n` и `s`. Некоторые программисты добавляют к осмысленным именам параметров префикс "a".

```
public Employee(String aName, double aSalary)
{
    name = aName;
    salary = aSalary
}
```

Такой код вполне понятен. Любой читающий его может сразу определить, в чем заключается смысл параметра. Имеется еще один широко распространенный прием. Чтобы воспользоваться им, следует знать, что параметры *скрывают* поля экземпляра с такими же именами. Так, если вызвать метод с параметром `salary`, то ссылка `salary` будет делаться на параметр, а не на поле экземпляра. Доступ к полю экземпляра осуществляется с помощью выражения `this.salary`. Напомним, что ключевое слово `this` обозначает неявный параметр, т.е. конструируемый объект. Следующий пример иллюстрирует данный подход:

```
public Employee(String name, double salary)
{
    this.name = name;
    this.salary = salary;
}
```



НА ЗАМЕТКУ C++! В C++ к именам полей экземпляра обычно добавляются префиксы, обозначаемые знаком подчеркивания или буквой (нередко для этой цели служит буква `m` или `x`). Например, поле, в котором хранится сумма зарплаты, может называться `_salary`, `mSalary` или `xSalary`. Программирующие на Java, как правило, так не поступают.

Вызов одного конструктора из другого

Ключевым словом `this` обозначается неявный параметр метода. Но у этого слова имеется еще одно назначение. Если *первый оператор* конструктора имеет вид `this(...)`, то вызывается другой конструктор этого же класса. Ниже приведен типичный тому пример.

```
public Employee(double s)
{
    // вызвать конструктор Employee(String, double)
    this("Employee " + nextId, s);
    nextId++;
}
```

Если используется выражение `new Employee(60000)`, то конструктор `Employee(double)` вызывает конструктор `Employee(String, double)`. Применять ключевое слово `this` для вызова другого конструктора очень удобно — нужно лишь один раз написать общий код для конструирования объекта.



НА ЗАМЕТКУ C++! Ссылка `this` в Java сродни указателю `this` в C++. Но в C++ нельзя вызвать один конструктор из другого. Для того чтобы реализовать общий код инициализации объекта в C++, нужно создать отдельный метод.

Блоки инициализации

Ранее мы рассмотрели два способа инициализации поля:

- установка его значения в конструкторе;
- присваивание значения при объявлении.

На самом деле в Java существует еще и третий механизм: использование блока инициализации. Такой блок выполняется каждый раз, когда создается объект данного класса. Рассмотрим следующий пример кода:

```
class Employee
{
    private static int nextId;

    private int id;
    private String name;
    private double salary;

    // блок инициализации
    {
        id = nextId;
        nextId++;
    }

    public Employee(String n, double s)
    {
        name = n;
        salary = s;
    }

    public Employee()
    {
        name = "";
        salary = 0;
    }

    ...
}
```

В этом примере начальное значение поля `id` задается в блоке инициализации объекта, причем неважно, какой именно конструктор используется для создания экземпляра класса. Блок инициализации выполняется первым, а вслед за ним — тело конструктора. Этот механизм совершенно не обязателен и обычно не применяется. Намного чаще применяются более понятные способы задания начальных значений полей.



НА ЗАМЕТКУ! В блоке инициализации допускается обращение к полям, определения которых находятся после данного блока. Несмотря на то что инициализация полей, определяемых после блока, формально допустима, поступать так не рекомендуется во избежание циклических определений. Конкретные правила изложены в разделе 8.3.2.3 спецификации Java (<http://docs.oracle.com/javase/specs>). Эти правила достаточно сложны, и учесть их в реализации компилятора крайне трудно. Так, в ранних версиях компилятора они были реализованы не без ошибок. Поэтому в коде блоки инициализации рекомендуется размещать после определений полей.

При таком многообразии способов инициализации полей данных довольно трудно отследить все возможные пути процесса конструирования объектов. Поэтому рассмотрим подробнее те действия, которые происходят при вызове конструктора.

1. Все поля инициализируются значениями, предусмотренными по умолчанию (0, false или null).

2. Инициализаторы всех полей и блоки инициализации выполняются в порядке их следования в объявлении класса.
3. Если в первой строке кода одного конструктора вызывается другой конструктор, то выполняется вызываемый конструктор.
4. Выполняется тело конструктора.

Всякий согласится, что код, отвечающий за инициализацию полей, нужно организовать так, чтобы в нем было легко разобраться. Например, было бы странным, если бы вызов конструкторов класса зависел от порядка объявления полей. Такой подход чреват ошибками.

Инициализировать статическое поле следует, задавая его начальное значение или используя статический блок инициализации. Первый механизм уже рассматривался ранее, а его пример приведен ниже.

```
static int nextId = 1;
```

Если для инициализации статических полей класса требуется сложный код, то удобнее использовать статический блок инициализации. Для этого следует разместить код в блоке и пометить его ключевым словом `static`. Допустим, идентификационные номера сотрудников должны начинаться со случайного числа, не превышающего **10000**. Соответствующий блок инициализации будет выглядеть следующим образом:

```
// Статический блок инициализации
static
{
    Random generator = new Random();
    nextId = generator.nextInt(10000);
}
```

Статическая инициализация выполняется в том случае, если класс загружается впервые. Аналогично полям экземпляра, статические поля принимают значения `0`, `false` или `null`, если не задать другие значения явным образом. Все операторы, задающие начальные значения статических полей, а также статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.



НА ЗАМЕТКУ! Оказывается, что элементарную программу, выводящую символьную строку "Hello, World", можно создать без метода `main()`, как показано ниже.

```
public class Hello
{
    static
    {
        System.out.println("Hello, World!");
    }
}
```

При выполнении команды `java Hello` загружается класс `Hello`, статический блок инициализации выводит строку "Hello, World!" и лишь затем появляется сообщение о том, что метод `main()` не определен. Это сообщение можно подавить, вызвав метод `System.exit(0)` в конце статического блока инициализации.

В программе, приведенной в листинге 4.5, наглядно демонстрируются многие языковые средства Java, обсуждавшиеся в этом разделе, включая следующие.

- Перегрузка конструкторов.
- Вызов одного конструктора из другого по ссылке `this(...)`.
- Применение конструктора без аргументов.
- Применение блока инициализации.
- Применение статической инициализации.
- Инициализация полей экземпляра.

Листинг 4.5. Исходный код из файла `ConstructorTest/ConstructorTest.java`

```
1 import java.util.*;
2 /**
3  * В этой программе демонстрируется конструирование объектов
4  * @version 1.01 2004-02-19
5  * @author Cay Horstmann
6  */
7 public class ConstructorTest
8 {
9     public static void main(String[] args)
10    {
11        // заполнить массив staff тремя объектами типа Employee
12
13        Employee[] staff = new Employee[3];
14
15        staff[0] = new Employee("Harry", 40000);
16        staff[1] = new Employee(60000);
17        staff[2] = new Employee();
18
19        // вывести данные обо всех объектах типа Employee
20
21        for (Employee e : staff)
22            System.out.println("name=" + e.getName() + ",id=" +
23                e.getId() + ",salary="+ e.getSalary());
24    }
25 }
26 class Employee
27 {
28     private static int nextId;
29
30     private int id;
31     private String name = ""; // инициализация поля экземпляра
32     private double salary;
33
34     // статический блок инициализации
35
36     static
37     {
38         Random generator = new Random();
39         // задать произвольное число 0-999 в поле nextId
40         nextId = generator.nextInt(10000);
41     }
```

```
42
43 // блок инициализации объекта
44
45 {
46     id = nextId;
47     nextId++;
48 }
49
50 // три перегружаемых конструктора
51
52 public Employee(String n, double s)
53 {
54     name = n;
55     salary = s;
56 }
57
58 public Employee(double s)
59 {
60     // вызвать конструктор Employee(String, double)
61     this("Employee #" + nextId, s);
62 }
63
64 // конструктор без аргументов
65
66 public Employee()
67 {
68     // поле name инициализируется пустой строкой "" - см. ниже
69     // поле salary не устанавливается явно, а инициализируется нулем
70     // поле id инициализируется в блоке инициализации
71 }
72 public String getName()
73 {
74     return name;
75 }
76
77 public double getSalary()
78 {
79     return salary;
80 }
81
82 public int getId()
83 {
84     return id;
85 }
86 }
```

java.util.Random 1.0

- Random()
Создает новый генератор случайных чисел.
- int nextInt(int n) 1.2
Возвращает случайное число в пределах от 0 до n-1.

Уничтожение объектов и метод `finalize()`

В некоторых объектно-ориентированных языках программирования и, в частности, в C++ имеются явные деструкторы, предназначенные для уничтожения объектов. Их основное назначение — освободить память, занятую объектами. В Java реализован механизм автоматической “сборки мусора”, освободить память вручную нет никакой необходимости, и поэтому в этом языке деструкторы отсутствуют.

Разумеется, некоторые объекты используют кроме памяти и другие ресурсы, например файлы, или оперируют другими объектами, которые, в свою очередь, обращаются к системным ресурсам. В этом случае очень важно, чтобы ресурсы освобождались вовремя. С этой целью в любой класс можно ввести метод `finalize()`, который будет вызван перед тем, как система “сборки мусора” уничтожит объект. Но на практике, если требуется возобновить ресурсы и сразу использовать их повторно, нельзя полагаться только на метод `finalize()`, поскольку заранее неизвестно, когда именно этот метод будет вызван.



НА ЗАМЕТКУ! Существует метод `System.runFinalizerOnExit(true)`, гарантирующий, что метод `finalize()` будет вызван до того, как программа завершит свою работу. Но и этот метод крайне ненадежен и не рекомендован к применению. В качестве альтернативы можно воспользоваться методом `Runtime.addShutdownHook()`. Дополнительные сведения о нем можно найти в документации на прикладной интерфейс API.

Если ресурс должен быть освобожден сразу после его использования, в таком случае придется написать соответствующий код самостоятельно. С этой целью следует предоставить метод `close()`, выполняющий необходимые операции по очистке памяти, вызвав его, когда соответствующий объект больше не нужен. В разделе “Оператор `try` с ресурсами” главы 11 будет показано, каким образом обеспечивается автоматический вызов такого метода.

Пакеты

Язык Java позволяет объединять классы в наборы, называемые *пакетами*. Пакеты облегчают организацию работы и позволяют отделить классы, созданные одним разработчиком, от классов, разработанных другими. Стандартная библиотека Java содержит большое количество пакетов, в том числе `java.lang`, `java.util`, `java.net` и т.д. Стандартные пакеты Java представляют собой иерархические структуры. Подобно каталогам на диске компьютера, пакеты могут быть вложены один в другой. Все стандартные пакеты принадлежат иерархиям пакетов `java` и `javax`.

Пакеты служат в основном для обеспечения однозначности имен классов. Допустим, двух программистов осенила блестящая идея создать класс `Employee`. Если оба класса будут находиться в разных пакетах, конфликт не возникнет. Чтобы обеспечить абсолютную однозначность имени пакета, рекомендуется использовать доменное имя компании в Интернете (оно по определению уникально), записанное в обратном порядке. В составе пакета можно создавать подпакеты и использовать их в разных проектах. Так, у одного из авторов этой книги имеется зарегистрированный домен `horstmann.com`. Записав это имя в обратном порядке, можно использовать его как название пакета — `com.horstmann`. В дальнейшем в этом пакете можно создать подпакет, например `com.horstmann.corejava`.

Единственная цель вложенных пакетов — гарантировать однозначность имен. С точки зрения компилятора между вложенными пакетами нет никакой связи. Например, пакеты `java.util` и `java.util.jar` вообще не связаны друг с другом. Каждый из них представляет собой независимую коллекцию классов.

Импорт классов

Класс может использовать все классы из собственного пакета и все *открытые* классы из других пакетов. Доступ к классам из других пакетов можно получить двумя способами. Во-первых, перед именем каждого класса можно указать полное имя пакета, как показано ниже.

```
java.util.Date today = new java.util.Date();
```

Очевидно, что этот способ слишком трудоемок. Второй, более простой и распространенный способ предусматривает применение ключевого слова `import`. В этом случае имя пакета перед именем класса записывать не обязательно.

Импортировать можно как один конкретный класс, так и весь пакет. Операторы `import` следует поместить в начало исходного файла (после всех операторов `package`). Например, все классы из пакета `java.util` можно импортировать следующим образом:

```
import java.util.*;
```

После этого имя пакета не указывается, как показано ниже.

```
Date today = new Date();
```

Отдельный класс можно также импортировать из пакета следующим образом:

```
import java.util.Date;
```

Импортировать все классы проще. На объем кода это не оказывает никакого влияния. Но если указать импортируемый класс явным образом, то читающему исходный код программы станет ясно, какие именно классы будут в ней использоваться.



СОВЕТ. Работая в ИСР Eclipse, можно выбрать команду меню `Source⇒Organize Imports` (Исходный код⇒Организовать импорт). В результате выражения типа `import java.util.*;` будут автоматически преобразованы в последовательности строк, предназначенных для импорта отдельных классов:

```
import java.util.ArrayList;
import java.util.Date;
```

Такая возможность очень удобна при написании кода.

Следует, однако, иметь в виду, что оператор `import` со звездочкой можно применять для импорта только одного пакета. Нельзя использовать обозначение `import java.*` или `import java.*.*`, чтобы импортировать все пакеты, имена которых содержат префикс **java**. В большинстве случаев импортируется весь пакет, независимо от его размера. Единственный случай, когда следует обратить особое внимание на пакет, возникает при конфликте имен. Например, оба пакета, `java.util` и `java.sql`, содержат класс `Date`. Допустим, разрабатывается программа, в которой оба эти пакета импортируются следующим образом:

```
import java.util.*;
import java.sql.*;
```

Если теперь попытаться воспользоваться классом `Date`, возникнет ошибка компиляции:

```
Date today; // ОШИБКА: неясно, какой пакет: java.util.Date или java.sql.date?
```

Компилятор не в состоянии определить, какой именно класс `Date` требуется в программе. Разрешить это затруднение можно, добавив конкретный оператор `import` следующим образом:

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

А что, если на самом деле нужны оба класса `Date`? В этом случае нужно указать полное имя пакета перед именем каждого класса, как показано ниже.

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

Обнаружение классов в пакетах является задачей компилятора. В байт-кодах, находящихся в файлах классов, всегда содержатся полные имена пакетов.



НА ЗАМЕТКУ C++! Программирующие на C++ считают, что оператор `import` является аналогом директивы `#include`. Но у них нет ничего общего. В языке C++ директиву `#include` приходится применять в объявлениях внешних ресурсов потому, что компилятор C++ не просматривает файлы, кроме компилируемого, а также файлы, указанные в самой директиве `#include`. А компилятор Java просматривает содержимое всех файлов при условии, если известно, где их искать. В Java можно и не применять механизм импорта, явно называя все пакеты, например `java.util.Date`. А в C++ избежать использования директивы `#include` нельзя.

Единственное преимущество оператора `import` — удобство. Он позволяет использовать более короткие имена классов, не указывая полное имя пакета. Например, после оператора `import java.util.*` (или `import java.util.Date`) к классу `java.util.Date` можно обращаться по имени `Date`.

Аналогичный механизм работы с пакетами в C++ реализован в виде директивы `namespace`. Операторы `package` и `import` в Java можно считать аналогами директив `namespace` и `using` в C++.

Статический импорт

Имеется форма оператора `import`, позволяющая импортировать не только классы, но и статические методы и поля. Допустим, в начале исходного файла введена следующая строка кода:

```
import static java.lang.System.*;
```

Это позволит использовать статические методы и поля, определенные в классе `System`, не указывая имени класса, следующим образом:

```
out.println("Goodbye, World!"); // вместо System.out
exit(0); // вместо System.exit
```

Статические методы или поля можно также импортировать явным образом:

```
import static java.lang.System.out;
```

Но в практике программирования на Java такие выражения, как `System.out` или `System.exit`, обычно не сокращаются. Ведь в этом случае исходный код станет более трудным для восприятия. С другой стороны, следующая строка кода:

```
sqrt(pow(x, 2) + pow(y, 2))
```

выглядит более ясной, чем эта:

```
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
```

Ввод классов в пакеты

Чтобы ввести класс в пакет, следует указать имя пакета в начале исходного файла *перед* определением класса. Например, исходный файл `Employee.java` из листинга 4.7 начинается следующими строками кода:

```
package com.horstmann.corejava;
```

```
public class Employee
{
    ...
}
```

Если оператор `package` в исходном файле не указан, то классы, описанные в этом файле, вводятся в *пакет по умолчанию*. У пакета по умолчанию нет имени. Все рассмотренные до сих пор классы принадлежали пакету по умолчанию.

Пакеты следует размещать в подкаталоге, путь к которому соответствует полному имени пакета. Например, все файлы классов из пакета `com.horstmann.corejava` должны находиться в подкаталоге `com/horstmann/corejava` (или `com\horstmann\corejava` в Windows). Компилятор размещает файлы классов в той же самой структуре каталогов.

Программы из листингов 4.6 и 4.7 распределены по двум пакетам: класс `PackageTest` принадлежит пакету по умолчанию, а класс `Employee` — пакету `com.horstmann.corejava`. Следовательно, файл `Employee.class` должен находиться в подкаталоге `com/horstmann/corejava`. Иными словами, структура каталогов должна выглядеть следующим образом:

```
. (базовый каталог)
├─ PackageTest.java
├─ PackageTest.class
├─ com/
│   └─ horstmann/
│       └─ corejava/
│           ├── Employee.java
│           └─ mployee.class
```

Чтобы скомпилировать программу из листинга 4.6, перейдите в каталог, содержащий файл `PackageTest.java`, и выполните следующую команду:

```
javac Package.java
```

Компилятор автоматически найдет файл `com/horstmann/corejava/Employee.java` и скомпилирует его.

Рассмотрим более практический пример. В данном примере пакет по умолчанию не используется. Вместо этого классы распределены по разным пакетам (`com.horstmann.corejava` и `com.mycompany`), как показано ниже.

```

. (базовый каталог)
├── com/
│   ├── horstmann/
│   │   ├── corejava/
│   │   │   ├── Employee.java
│   │   │   └── Employee.class
│   │   └── mycompany/
│   │       ├── PayrollApp.java
│   │       └── PayrollApp.class

```

И в этом случае классы следует компилировать и запускать из базового каталога, т.е. того каталога, в котором содержится подкаталог `com`, как показано ниже.

```

javac com/mycompany/PayrollApp.java
java com.mycompany.PayrollApp

```

Не следует также забывать, что компилятор работает с файлами (при указании файла задаются путь и расширение `.java`). А интерпретатор оперирует классами (имя каждого класса указывается в пакете через точку).

Листинг 4.6. Исходный код из файла `PackageTest/PackageTest.java`

```

1 import com.horstmann.corejava.*;
2 // В этом пакете определен класс Employee
3
4 import static java.lang.System.*;
5 /**
6  * В этой программе демонстрируется применение пакетов
7  * @version 1.11 2004-02-19
8  * @author Cay Horstmann
9  */
10 public class PackageTest
11 {
12     public static void main(String[] args)
13     {
14         // здесь не нужно указывать полное имя com.horstmann.corejava.Employee
15         // поскольку используется оператор import
16         Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
17
18         harry.raiseSalary(5);
19
20         // здесь не нужно указывать полное имя System.out,
21         // поскольку используется оператор static import
22         out.println("name=" + harry.getName() + ", salary=" + harry.getSalary());
23     }
24 }

```

Листинг 4.7. Исходный код из файла `PackageTest/com/horstmann/corejava/Employee.java`

```

1 package com.horstmann.corejava;
2 // классы из этого файла входят в данный пакет
3
4 import java.util.*;
5 // операторы import следуют после оператора package
6
7 /**

```

```
8  * @version 1.10 1999-12-18
9  * @author Cay Horstmann
10 */
11 public class Employee
12 {
13     private String name;
14     private double salary;
15     private Date hireDay;
16
17     public Employee(String n, double s, int year, int month, int day)
18     {
19         name = n;
20         salary = s;
21         GregorianCalendar calendar =
22             new GregorianCalendar(year, month - 1, day);
23         // В классе GregorianCalendar январь обозначается нулем
24         hireDay = calendar.getTime();
25     }
26
27     public String getName()
28     {
29         return name;
30     }
31
32     public double getSalary()
33     {
34         return salary;
35     }
36
37     public Date getHireDay()
38     {
39         return hireDay;
40     }
41
42     public void raiseSalary(double byPercent)
43     {
44         double raise = salary * byPercent / 100;
45         salary += raise;
46     }
47 }
48 }
```



СОВЕТ. Начиная со следующей главы в исходном коде приводимых примеров программ будут использоваться пакеты. Это даст вам возможность создавать проекты в ИСП по отдельным главам, а не по разделам.



ВНИМАНИЕ! Компилятор не проверяет структуру каталогов. Допустим, исходный файл начинается со следующей директивы:

```
package com.mycompany;
```

Этот файл можно скомпилировать, даже если он не находится в каталоге `com/mycompany`. Исходный файл будет скомпилирован без ошибок, если он не зависит от других пакетов, но при попытке выполнить эту программу виртуальная машина не найдет нужные классы.

Область действия пакетов

В приведенных ранее примерах кода уже встречались модификаторы доступа `public` и `private`. Открытые компоненты, помеченные ключевым словом `public`, могут использоваться любым классом. А закрытые компоненты, в объявлении которых указано ключевое слово `private`, могут использоваться только тем классом, в котором они были определены. Если же ни один из модификаторов доступа не указан, то компонент программы (класс, метод или переменная) доступен всем методам в том же самом *пакете*.

Обратимся снова к примеру программы из листинга 4.2. Класс `Employee` не определен в ней как открытый. Следовательно, любой другой класс из того же самого пакета (в данном случае пакета, заданного по умолчанию), например класс `EmployeeTest`, может получить к нему доступ. Для классов такой подход следует признать вполне разумным. Но для переменных подобный способ доступа не годится. Переменные должны быть явно обозначены как `private`, иначе их область действия будет по умолчанию расширена до пределов пакета, что, безусловно, нарушает принцип инкапсуляции. В процессе работы над программой разработчики часто забывают указать ключевое слово `private`. Ниже приведен пример из класса `Window`, принадлежащего пакету `java.awt`, который входит в состав JDK.

```
public class Window extends Container
{
    String warningString;
    . . .
}
```

Обратите внимание на то, что у переменной `warningString` отсутствует модификатор доступа `private`! Это означает, что методы всех классов из пакета `java.awt` могут обращаться к ней и изменить ее значение, например, присвоить ей строку "Trust me!" (Доверьтесь мне!). Фактически все методы, имеющие доступ к переменной `warningString`, принадлежат классу `Window`, поэтому эту переменную можно было бы смело объявить как закрытую. Мы подозреваем, что те, кто писал этот код, торопились и просто забыли указать ключевое слово `private`. (Не станем упоминать имени автора этого кода, чтобы не искать виноватого, — вы сами можете заглянуть в исходный код.)



НА ЗАМЕТКУ! Как ни странно, этот недостаток не был устранен даже после того, как мы указали на него в восьми изданиях этой книги. Очевидно, что разработчики, данной библиотеки не читают нашу книгу. Более того, со временем в класс `Window` были добавлены новые поля, и снова половина из них не была объявлена как `private`.

А может, это совсем и не оплошность? Однозначно ответить на этот вопрос нельзя. По умолчанию пакеты не являются открытыми, а это означает, что всякий может добавлять в пакеты свои классы. Разумеется, злонамеренные или невежественные программисты могут написать код, модифицирующий переменные, область действия которых ограничена пакетом. Например, в ранних версиях Java можно было легко проникнуть в другой класс пакета `java.awt`. Для этого достаточно было начать определение нового класса со следующей строки:

```
package java.awt;
```

а затем разместить полученный файл класса в подкаталоге `java\awt`. В итоге содержимое пакета `java.awt` становилось открытым для доступа. Подобным ловким способом можно было изменить строку предупреждения, отображаемую на экране (рис. 4.9).

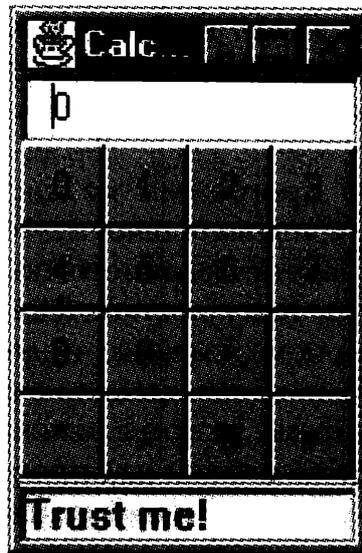


Рис. 4.9. Изменение строки предупреждения в окне апплета

В версии 1.2 создатели JDK запретили загрузку классов, определенных пользователем, если их имя начиналось с "java. "! Разумеется, эта защита не распространяется на ваши собственные классы. Вместо этого вы можете воспользоваться другим механизмом, который называется *герметизацией пакета* и ограничивает доступ к пакету. Произведя герметизацию пакета, вы запрещаете добавлять в него классы. В главе 10 будет показано, как создать файл формата JAR, содержащий герметичные пакеты.

Путь к классам

Как вам должно быть уже известно, классы хранятся в подкаталогах файловой системы. Путь к классу должен совпадать с именем пакета. Кроме того, можно воспользоваться утилитой `jar`, чтобы разместить классы в архивном файле формата JAR (архиве Java в дальнейшем — просто JAR-файле). В одном архивном файле многие файлы классов и подкаталогов находятся в сжатом виде, что позволяет экономить память и сокращать время доступа к ним. Когда вы пользуетесь библиотекой от независимых разработчиков, вы обычно получаете в свое распоряжение один JAR-файл или более. В состав JDK также входит целый ряд JAR-файлов, как, например, файл `jre/lib/rt.jar`, содержащий тысячи библиотечных классов. Из главы 10 вы узнаете о том, как создавать собственные JAR-файлы.



СОВЕТ. Для организации файлов и подкаталогов в архивных JAR-файлах используется формат ZIP. Обращаться с файлом `rt.jar` и другими JAR-файлами можно с помощью любой утилиты, поддерживающей формат ZIP.

Чтобы обеспечить совместный доступ программ к классам, выполните следующие действия.

1. Разместите файлы классов в одном или более специальном каталоге, например `/home/user/classdir`. Следует иметь в виду, что этот каталог является *базовым* по отношению к дереву пакета. Если потребуется добавить класс `com.horstmann.corejava.Employee`, то файл класса следует разместить в подкаталоге `/home/user/classdir/com/horstmann/corejava`.
2. Разместите все JAR-файлы в одном каталоге, например `/home/user/archives`.
3. Задайте путь к классу. *Путь к классу* — это совокупность всех базовых каталогов, которые могут содержать файлы классов.

В Unix составляющие пути к классу отделяются друг от друга двоеточиями:

```
/home/user/classdir:./home/user/archives/archive.jar
```

А в Windows они разделяются точками с запятой:

```
c:\clasdir;.;c:\archives\archive.jar
```

В обоих случаях точка означает текущий каталог.

Путь к классу содержит:

- имя базового каталога `/home/user/classdir` или `c:\classes`;
- обозначение текущего каталога `(.)`;
- имя JAR-файла `/home/user/archives/archive.jar` или `c:\archives\archive.jar`.

Начиная с версии Java SE 6 для обозначения каталога с JAR-файлами можно указывать шаблонные символы одним из следующих способов:

```
/home/user/classdir:./home/user/archives/'*'
```

или

```
c:\classdir;.;c:\archives\*
```

В UNIX шаблонный символ `*` должен быть экранирован, чтобы исключить его интерпретацию командной оболочкой. Все JAR-файлы (но не файлы с расширением `.class`), находящиеся в каталоге `archives`, включаются в путь к классам. Файлы из библиотеки рабочих программ (`rt.jar` и другие файлы формата JAR в каталогах `jre/lib` и `jre/lib/ext`) всегда участвуют в поиске классов, поэтому их не нужно включать явно в путь к классам.



ВНИМАНИЕ! Компилятор `javac` всегда ищет файлы в текущем каталоге, а загрузчик виртуальной машины `java` обращается к текущему каталогу только в том случае, если в пути к классам указана точка `(.)`. Если путь к классам не указан, никаких осложнений не возникает — по умолчанию в него включается текущий каталог `(.)`. Если же вы задали путь к классам и забыли указать точку, ваша программа будет скомпилирована без ошибок, но выполняться не будет.

В пути к классам перечисляются все каталоги и архивные файлы, которые служат исходными точками для поиска классов. Рассмотрим следующий простой путь к классам:

```
/home/user/classdir:./home/user/archives/archive.jar
```

Допустим, интерпретатор осуществляет поиск файла, содержащего класс `com.horstmann.corejava.Employee`. Сначала он ищет его в системных файлах, которые хранятся в архивах, находящихся в каталогах `jre/lib` и `jre/lib/ext`. В этих файлах искомый класс отсутствует, поэтому интерпретатор анализирует пути к классам, по которым он осуществляет поиск следующих файлов:

- `/home/user/classdir/com/horstmann/corejava/Employee.class`;
- `com.horstmann/corejava/Employee.class`, начиная с текущего каталога;
- `com.horstmann/corejava/Employee.class` в каталоге `/home/user/archives/archive.jar`.

На поиск файлов компилятор затрачивает больше времени, чем виртуальная машина. Если вы указали класс, не назвав пакета, которому он принадлежит, компилятор сначала должен сам определить, какой именно пакет содержит данный класс. В качестве возможных источников для классов рассматриваются пакеты, указанные в директивах `import`. Допустим, исходный файл содержит приведенные ниже директивы, а также код, в котором происходит обращение к классу `Employee`.

```
import java.util.*  
import com.horstmann.corejava.*;
```

Компилятор попытается найти классы `java.lang.Employee` (поскольку пакет `java.lang` всегда импортируется по умолчанию), `java.util.Employee`, `com.horstmann.corejava.Employee` и `Employee` в текущем пакете. Он ищет каждый из этих файлов во всех каталогах, указанных в пути к классам. Если найдено несколько таких классов, возникает ошибка компиляции. (Классы должны быть определены однозначно, и поэтому порядок следования директив `import` особого значения не имеет.)

Компилятор делает еще один шаг. Он просматривает *исходные* файлы, чтобы проверить, были ли они созданы позже, чем файлы классов. Если проверка дает положительный результат, исходный файл автоматически перекомпилируется. Напомним, что из других пакетов можно импортировать лишь открытые классы. Исходный файл может содержать только один открытый класс, а кроме того, имена файла и класса должны совпадать. Следовательно, компилятор может легко определить, где находятся исходные файлы, содержащие открытые классы. Из текущего пакета можно импортировать также классы, не определенные как открытые (т.е. `public`). Исходные коды классов могут содержаться в файлах с разными именами. При импорте класса из текущего пакета компилятор проверяет все исходные файлы, чтобы выяснить, в каком из них определен требуемый класс.

Указание пути к классам

Путь к классам лучше всего указать с помощью параметра `-classpath` (или `-cp`) следующим образом:

```
java -classpath /home/user/classdir:./home/user/archives/archive.jar MyProg
```

или

```
java -classpath c:\classdir;.;c:\archives\archive.jar MyProg.java
```

Вся команда должна быть набрана в одной строке. Лучше всего ввести такую длинную команду в сценарий командной оболочки или командный файл. Применение параметра `-classpath` считается более предпочтительным способом установки путей к классам. Альтернативный способ — переменная окружения `CLASSPATH`. А далее все зависит от конкретной командной оболочки. Так, в командной оболочке Bourne Again (bash) для установки путей к классам используется следующая команда:

```
export CLASSPATH=/home/user/classdir:./home/user/archives/archive.jar
```

В командной оболочке C shell для этой цели применяется следующая команда:

```
setenv CLASSPATH /home/user/classdir:./home/user/archives/archive.jar
```

А в командной строке Windows — команда

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

Путь к классам действителен до завершения работы командной оболочки.



ВНИМАНИЕ! Некоторые советуют устанавливать переменную окружения `CLASSPATH` на постоянной основе. В целом это неудачная идея. Сначала разработчики забывают о глобальных установках, а затем удивляются, когда классы не загружаются правильно. Характерным тому примером служит программа установки приложения QuickTime от компании Apple в Windows. Эта программа глобально устанавливает переменную окружения `CLASSPATH` таким образом, чтобы она указывала на нужный ей JAR-файл, не включая в путь текущий каталог. В итоге очень многие программирующие на Java попадают в тупиковую ситуацию, когда их программы сначала компилируются, а затем не запускаются.



ВНИМАНИЕ! Некоторые советуют вообще не указывать путь к классам, сбрасывая все файлы формата JAR в каталог `jre/lib/ext`. Это очень неудачное решение по двум причинам. Архивы, из которых другие классы загружаются вручную, не работают правильно, когда они размещены в каталоге расширений (подробнее о загрузчиках классов речь пойдет в главе 9 второго тома данной книги). Более того, программистам свойственно забывать о файлах, которые они разместили там три месяца назад. А затем они недоумевают, почему загрузчик файлов игнорирует их тщательно продуманный путь к классам, загружая на самом деле давно забытые классы из каталога расширений.

Комментарии и документирование

В состав JDK входит полезное инструментальное средство — утилита `javadoc`, генерирующая документацию в формате HTML из исходных файлов. Фактически интерактивная документация на прикладной интерфейс API является результатом применения утилиты `javadoc` к исходному коду стандартной библиотеки Java.

Добавив к исходному коду комментарии, начинающиеся с последовательности символов `/**`, нетрудно создать документацию, имеющую профессиональный вид. Это очень удобный способ, поскольку он позволяет совместно хранить как код, так и документацию к нему. Если же поместить документацию в отдельный файл, то со временем она перестанет соответствовать коду. В то же время документацию нетрудно обновить, повторно запустив на выполнение утилиту `javadoc`, поскольку комментарии являются неотъемлемой частью исходного файла.

Вставка комментариев

Утилита `javadoc` извлекает сведения о следующих компонентах программы.

- Пакеты.
- Классы и интерфейсы, объявленные как `public`.
- Методы, объявленные как `public` или `protected`.
- Поля, объявленные как `public` или `protected`.

Защищенные компоненты программы, для объявления которых используется ключевое слово `protected`, будут рассмотрены в главе 5, а интерфейсы — в главе 6. Разрабатывая программу, можно (и даже нужно) комментировать каждый из перечисленных выше компонентов. Комментарии размещаются непосредственно перед тем компонентом, к которому они относятся. Комментарии начинаются символами `/**` и оканчиваются символами `*/`. Комментарии вида `/** ... */` содержат произвольный текст, за которым следует дескриптор. Дескриптор начинается с символа `@`, например `@author` или `@param`.

Первое предложение в тексте комментариев должно быть кратким описанием. Утилита `javadoc` автоматически генерирует страницы, состоящие из кратких описаний. В самом тексте можно использовать элементы HTML-разметки, например, `...` — для выделения текста курсивом, `<code>...</code>` — для форматирования текста моноширинного шрифта, `...` — для выделения текста полужирным и даже `` — для вставки рисунков. Следует, однако, избегать применения заголовков (`<h1>` — `<h6>`) и горизонтальных линий (`<hr>`), поскольку они могут помешать нормальному форматированию документа.



НА ЗАМЕТКУ! Если комментарии содержат ссылки на другие файлы, например рисунки (диаграммы или изображения компонентов пользовательского интерфейса), разместите эти файлы в каталоге `doc-files`, содержащем исходный файл. Утилита `javadoc` скопирует эти каталоги вместе с находящимися в них файлами из исходного каталога в данный каталог, выделяемый для документации. Поэтому в своих ссылках вы должны непременно указать каталог `doc-files`, например ``.

Комментарии к классам

Комментарии к классу должны размещаться после директив `import`, непосредственно перед определением класса. Ниже приведен пример подобных комментариев.

```
/**
 *Объект класса Card имитирует игральную карту,
 *например даму червей. Карта имеет масть и ранг
 *(1=туз, 2...10, 11=валет, 12=дама, 13=король).
 */
public class Card
{
    ...
}
```



НА ЗАМЕТКУ! Начинать каждую строку документации звездочкой нет никакой нужды. Например, следующий комментарий вполне корректен:

```
/**
 * Объект класса Card имитирует игральную карту,
 * например, даму червей. Карта имеет масть и ранг
 * ( 1=туз, 2...10, 11=валет, 12=дама, 13=король).
 */
```

Но в большинстве ИСР звездочки в начале строки комментария устанавливаются автоматически и перестраиваются при изменении расположения переносов строк.

Комментарии к методам

Комментарии должны непосредственно предшествовать методу, который они описывают. Кроме дескрипторов общего назначения, можно использовать перечисленные ниже специальные дескрипторы.

- *@param описание переменной*

Добавляет в описание метода раздел параметров. Раздел параметров можно развернуть на несколько строк. Кроме того, можно использовать элементы HTML-разметки. Все дескрипторы *@param*, относящиеся к одному методу, должны быть сгруппированы вместе,

- *@return описание*

Добавляет в описание метода раздел возвращаемого значения. Этот раздел также может занимать несколько строк и допускает форматирование с помощью дескрипторов HTML-разметки.

- *@throws описание класса*

Указывает на то, что метод способен генерировать исключение. Исключения будут рассмотрены в главе 11.

Рассмотрим пример комментариев к методу.

```
/**
 * Увеличивает зарплату сотрудников
 * @param Переменная byPercent содержит величину
 * в процентах, на которую повышается зарплата
 * (например, 10 = 10%).
 * @return Величина, на которую повышается зарплата
 */
public double raiseSalary(double byPercent)
{
    double raise = salary * byPercent / 100;
    salary += raise;
    return raise;
}
```

Комментарии к полям

Документировать нужно лишь открытые поля. Они, как правило, являются статическими константами. Ниже приведен пример комментария к полю.

```
/**
 * Масть черви
 */
public static final int HEARTS = 1;
```

Комментарии общего характера

В комментариях к классам можно использовать следующие дескрипторы.

- @author *имя*

Создает раздел автора программы. В комментариях может быть несколько таких дескрипторов — по одному на каждого автора.

- @version *текст*

Создает раздел версии программы. В данном случае *текст* означает произвольное описание текущей версии программы.

При написании любых комментариев, предназначенных для составления документации, можно также использовать следующие дескрипторы.

- @since *текст*

Создает раздел начальной точки отсчета. Здесь *текст* означает описание версии программы, в которой впервые был внедрен данный компонент. Например, @since version 1.7.1.

- @deprecated *текст*

Добавляет сообщение о том, что класс, метод или переменная не рекомендуется к применению. Подразумевается, что после дескриптора @deprecated следует некоторое выражение. Например:

```
@deprecated Use setVisible(true) instead
```

С помощью дескрипторов @see и @link можно указывать гипертекстовые ссылки на соответствующие внешние документы или на отдельную часть того же документа, сформированного с помощью утилиты javadoc.

Дескриптор @see *ссылка* добавляет ссылку в раздел “См. также”. Его можно использовать в комментариях к классам и методам. Здесь *ссылка* означает одну из следующих конструкций:

```
пакет.класс#метка_компонента
<a href="...">метка</a>
"текст"
```

Первый вариант встречается чаще всего. Здесь нужно указать имена класса, метода или переменной, а утилита javadoc вставит в документацию соответствующую гипертекстовую ссылку. Например, в приведенной ниже строке кода создается ссылка на метод raiseSalary(double) из класса com.horstmann.corejava.Employee. Если

опустить имя только пакета или же имя пакета и класса, то комментарии к данному компоненту будут размещены в текущем пакете или классе.

```
@see com.horstmann.corejava.Employee#raiseSalary(double)
```

Обратите внимание на то, что для отделения имени класса от имени метода или переменной служит символ #, а не точка. Компилятор javac корректно обрабатывает точки, выступающие в роли разделителей имен пакетов, подпакетов, классов, внутренних классов, методов и переменных. Но утилита javadoc не настолько развита логически, и поэтому возникает потребность в специальном синтаксисе.

Если за дескриптором @see следует символ <, то необходимо указать гипертекстовую ссылку. Ссылаться можно на любой веб-адрес в формате URL, как показано ниже.

```
@see <a href="www.horstmann.com/corejava.html">Web-страница книги Core Java</a>
```

В каждом из рассматриваемых здесь вариантов составления гипертекстовых ссылок можно указать необязательную *метку*, которая играет роль точки привязки для ссылки. Если метку не указать, точкой привязки для ссылки будет считаться имя программы или URL.

Если за дескриптором @see следует символ ", то текст отображается в разделе "См. также", как показано ниже.

```
@see "Core Java volume 2"
```

Для комментирования одного и того же элемента можно использовать несколько дескрипторов @ see, но их следует сгруппировать вместе.

По желанию в любом комментарии можно разместить гипертекстовые ссылки на другие классы или методы. Для этого в любом месте документации достаточно ввести специальный дескриптор вида {link пакет.класс#метка_элемента}. Описание элемента подчиняется тем же правилам, что и для дескриптора @see.

Комментарии к пакетам и обзорные

Комментарии к классам, методам и переменным, как правило, размещаются непосредственно в исходных файлах и выделяются символами /** ... */. Но для формирования комментариев к *пакетам* следует добавить отдельный файл в каталог каждого пакета. Для этого имеются два варианта выбора.

1. Применить HTML-файл под именем package.html. Весь текст, содержащийся между дескрипторами <body> ... </body>, извлекается утилитой javadoc.
2. Подготовить файл под именем package-info.java. Этот файл должен содержать начальный комментарий Javadoc, отделенный символами /** и */, за которым следует оператор package, но больше никакого кода или комментариев.

Кроме того, все исходные файлы можно снабдить обзорными комментариями. Они размещаются в файле overview.html, расположенном в родительском каталоге со всеми исходными файлами. Весь текст, находящийся между дескрипторами <body> ... </body>, извлекается утилитой javadoc. Эти комментарии отображаются на экране, когда пользователь выбирает вариант Overview (Обзор) в панели навигации.

Извлечение комментариев

Допустим, что `docDirectory` — это имя каталога, в котором должны храниться HTML-файлы. Для извлечения комментариев выполните описанные ниже действия.

1. Перейдите в каталог с исходными файлами, которые подлежат документированию. Если документированию подлежат вложенные пакеты, например `com.horstmann.corejava`, перейдите в каталог, содержащий подкаталог `com`. (Именно в этом каталоге должен храниться файл `overview.html`.)

2. Для документирования одного пакета выполните следующую команду:

```
javadoc -d docDirectory имя_пакета
```

3. Для документирования нескольких пакетов выполните следующую команду:

```
javadoc -d docDirectory имя_пакета_1 имя_пакета_2 ...
```

4. Если файлы находятся в пакете, заданном по умолчанию, вместо приведенных выше команд выполните следующую команду:

```
javadoc -d docDirectory *.java
```

Если опустить параметр `-d docDirectory`, HTML-файлы будут извлечены в текущий каталог, что вызовет путаницу. Поэтому делать этого не рекомендуется.

При вызове утилиты `javadoc` можно указывать различные параметры. Например, чтобы включить в документацию дескрипторы `@author` и `@version`, можно выбрать параметры `-author` и `-version` (по умолчанию они опущены). Параметр `-link` позволяет включать в документацию ссылки на стандартные классы. Например, приведенная ниже команда автоматически создаст ссылку на документацию, находящуюся на веб-сайте компании Oracle.

```
javadoc -link http://docs.oracle.com/javase/7/docs/api *.java
```

Если же выбрать параметр `-linksource`, то каждый исходный файл будет преобразован в формат HTML (без цветового кодирования, но с номерами строк), а имя каждого класса и метода превратится в гиперссылку на исходный код. Другие параметры описаны в документации на `javadoc`, доступной по адресу <http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc>.



НА ЗАМЕТКУ! Если требуется выполнить более тонкую настройку, например, создать документацию в формате, отличающемся от HTML, можно разработать свой собственный доклет — приложение, позволяющее формировать документацию произвольного вида. Подробнее о таком способе можно узнать на веб-странице по адресу <http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/doclet/overview.html>.

Рекомендации по разработке классов

В завершение этой главы приведем некоторые рекомендации, которые призваны помочь вам в разработке классов в стиле ООП.

1. *Всегда храните данные в переменных, объявленных как private.*

Первое и главное требование: всеми средствами избегайте нарушения инкапсуляции. Иногда приходится писать методы доступа к полю или модифицирующие методы, но предоставлять доступ к полям не следует. Как показывает горький опыт, способ представления данных может изменяться, но порядок их использования изменяется много реже. Если данные закрыты, их представление не влияет на использующий их класс, что упрощает выявление ошибок.

2. *Всегда инициализируйте данные.*

В Java локальные переменные не инициализируются, но поля в объектах инициализируются. Не полагайтесь на действия по умолчанию, инициализируйте переменные явным образом с помощью конструкторов.

3. *Не используйте в классе слишком много простых типов.*

Несколько связанных между собой полей простых типов следует объединять в новый класс. Такие классы проще для понимания, а кроме того, их легче видоизменить. Например, следующие четыре поля из класса `Customer` нужно объединить в новый класс `Address`:

```
private String street;  
private String city;  
private String state;  
private int zip;
```

Представленный таким образом адрес легче изменить, как, например, в том случае, если требуется указать интернациональные адреса.

4. *Не для всех полей нужно создавать методы доступа и модификации.*

Очевидно, что при выполнении программы расчета зарплаты требуется получать сведения о зарплате сотрудника, а кроме того, ее приходится время от времени изменять. Но вряд ли придется менять дату его приема на работу после того, как объект сконструирован. Иными словами, существуют поля, которые после создания объекта совсем не изменяются. К их числу относятся, в частности, массив сокращенных названий штатов США в классе `Address`.

5. *Разбивайте на части слишком крупные классы.*

Это, конечно, слишком общая рекомендация: то, что кажется “слишком крупным” одному программисту, представляется нормальным другому. Но если есть очевидная возможность разделить один сложный класс на два класса попроще, то воспользуйтесь ею. (Только опасайтесь другой крайности. Вряд ли оправданы десять классов, в каждом из которых имеется только один метод.) Ниже приведен пример неудачного составления класса.

```
public class CardDeck // неудачная конструкция  
{  
    private int[] value;  
    private int[] suit;  
  
    public CardDeck() { . . . }  
    public void shuffle() { . . . }  
    public int getTopValue() { . . . }  
    public int getTopSuit() { . . . }  
    public void draw() { . . . }  
}
```

На самом деле в этом классе реализованы два разных понятия: во-первых, *карточный стол* с методами `shuffle()` (тасование) и `draw()` (раздача) и, во-вторых, *игральная карта* с методами для проверки ранга и масти карты. Разумнее было бы ввести класс `Card`, представляющий собой отдельную игральную карту. В итоге имелись бы два класса, каждый из которых отвечал бы за свое, как показано ниже.

```
public class CardDeck
{
    private Card[] cards;
    public CardDeck() { . . . }
    public void shuffle() { . . . }
    public Card getTop() { . . . }
    public void draw() { . . . }
}

public class Card
{
    private int value;
    private int suit;

    public Card(int aValue, int aSuit) { . . . }
    public int getValue() { . . . }
    public int getSuit() { . . . }
}
```

6. *Выбирайте для классов и методов осмысленные имена, ясно указывающие на их значение.*

Классы, как и переменные, следует называть именами, отражающими их назначение. (В стандартной библиотеке имеются примеры, где это правило нарушается, скажем, класс `Date` описывает время, а не дату.)

Удобно принять следующие условные обозначения: имя класса должно быть именем существительным (`Order`) или именем существительным, которому предшествует имя прилагательное (`RushOrder`) или деепричастие (`BillingAddress`). Как правило, методы доступа должны начинаться словом `get`, представленным строчными буквами (`getSalary`), а модифицирующие методы — словом `set`, также представленным строчными буквами (`setSalary`).

В этой главе были рассмотрены основы объектов и классов, которые делают Java объектным языком. Но для того чтобы быть действительно объектно-ориентированным, язык программирования должен также поддерживать наследование и полиморфизм. О реализации этих принципов ООП в Java речь пойдет в следующей главе.

Наследование

В этой главе...

- ▶ Классы, суперклассы и подклассы
- ▶ Глобальный суперкласс `Object`
- ▶ Обобщенные списочные массивы
- ▶ Объектные оболочки и автоупаковка
- ▶ Методы с переменным числом параметров
- ▶ Классы перечислений
- ▶ Рефлексия
- ▶ Рекомендации по применению наследования

В главе 4 были рассмотрены классы и объекты. А эта глава посвящена наследованию — фундаментальному принципу объектно-ориентированного программирования (ООП). Принцип наследования состоит в том, что новые классы можно создавать из уже существующих. При наследовании методы и поля существующего класса используются повторно (наследуются) вновь создаваемым классом, причем для адаптации нового класса к новым ситуациям в него добавляют дополнительные поля и методы. Этот прием играет в Java весьма важную роль.

В этой главе будет также рассмотрен механизм рефлексии, позволяющий исследовать свойства классов в ходе выполнения программы. *Рефлексия* — эффективный, но очень сложный механизм. А поскольку рефлексия больше интересует разработчиков инструментальных средств, чем прикладных программ, то при первом чтении можно ограничиться лишь беглым просмотром той части главы, которая посвящена данному механизму.

Классы, суперклассы и подклассы

Вернемся к примеру класса `Employee`, рассмотренному в предыдущей главе. Допустим, вы работаете в организации, где работа руководящего состава учитывается иначе, чем работа остальных сотрудников. Разумеется, руководители во многих отношениях являются обычными наемными сотрудниками. Руководящим и обычным сотрудникам выплачивается заработная плата, но первые за свои достижения получают еще и премии. В таком случае для расчета зарплаты следует применять наследование. Почему? А потому, что нужно определить новый класс `Manager`, в который придется ввести новые функциональные возможности. Но в этот класс можно перенести кое-что из того, что уже запрограммировано в классе `Employee`, сохранив все поля, определенные в исходном классе. Говоря более абстрактно, между классами `Manager` и `Employee` существует вполне очевидное отношение “является”: каждый руководитель является сотрудником. Именно это отношение и служит явным признаком наследования.

Ниже показано, как определяется класс `Manager`, производный от класса `Employee`. Для обозначения наследования в Java служит ключевое слово `extends`.

```
class Manager extends Employee
{
    Дополнительные методы и поля
}
```



НА ЗАМЕТКУ C++! Механизмы наследования в Java и C++ сходны. В Java вместо знака `:` для обозначения наследования служит ключевое слово `extends`. Любое наследование в Java является открытым, т.е. в этом языке нет аналога закрытому и защищенному наследованию, допускаемому в C++.

Ключевое слово `extends` означает, что на основе существующего класса создается новый. Существующий класс называется *суперклассом*, *базовым* или *родительским*, а вновь создаваемый — *подклассом*, *производным* или *порожденным*. В среде программирующих на Java наиболее широко распространены термины *суперкласс* и *подкласс*, хотя некоторые из них предпочитают пользоваться терминами *родительский* и *порожденный*, более тесно связанными с понятием наследования.

Класс `Employee` является суперклассом. Это не означает, что он имеет превосходство над своим подклассом или обладает более широкими функциональными возможностями. На самом деле все *наоборот*: функциональные возможности подкласса шире, чем у суперкласса. Как станет ясно в дальнейшем, класс `Manager` инкапсулирует больше данных и содержит больше методов, чем его суперкласс `Employee`.



НА ЗАМЕТКУ! Префиксы *супер-* и *под-* пришли в программирование из теории множеств. Множество всех сотрудников содержит в себе множество всех руководителей. В этом случае говорят, что множество сотрудников является *супермножеством* по отношению к множеству руководителей. Иначе говоря, множество всех руководителей является *подмножеством* для множества всех сотрудников.

Класс `Manager` содержит новое поле, в котором хранится величина премии, а также новый метод, позволяющий задавать эту величину, как показано ниже.

```
class Manager extends Employee
{
```

```
private double bonus;
. . .
public void setBonus(double b)
{
    bonus = b;
}
}
```

В этих методах и полях нет ничего особенного. Имея объект типа `Manager`, можно просто вызывать для него метод `setBonus()` следующим образом:

```
Manager boss = ...;
boss.setBonus(5000);
```

Разумеется, для объекта типа `Employee` вызвать метод `setBonus()` *нельзя*, поскольку его нет среди методов, определенных в классе `Employee`. Но в то же время методы `getName()` и `getHireDay()` можно вызывать для объектов типа `Manager`, поскольку они наследуются от суперкласса `Employee`, хотя и не определены в классе `Manager`. Поля `name`, `salary` и `hireDay` также наследуются от суперкласса. Таким образом, у каждого объекта типа `Manager` имеются четыре поля: `name`, `salary`, `hireDay` и `bonus`.

Определяя подкласс посредством расширения суперкласса, достаточно указать лишь отличия между подклассом и суперклассом. Разрабатывая классы, следует размещать общие методы в суперклассе, а специальные — в подклассе. Такое выделение общих функциональных возможностей в отдельном суперклассе широко распространено в ООП.

Но некоторые методы суперкласса не подходят для подкласса `Manager`. В частности, метод `getSalary()` должен возвращать сумму основной зарплаты и премии. Следовательно, нужно *переопределить* метод, т.е. реализовать новый метод, замещающий соответствующий метод из суперкласса:

```
class Manager extends Employee
{
    . . .
    public double getSalary()
    {
        . . .
    }
    . . .
}
```

Как же реализовать такой метод? На первый взгляд, сделать это очень просто — нужно лишь вернуть сумму полей `salary` и `bonus` следующим образом:

```
public double getSalary()
{
    return salary + bonus; // не работает!
}
```

Но оказывается, что такой способ не годится. Метод `getSalary()` из класса `Manager` не имеет доступа к закрытым полям суперкласса. Иными словами, метод `getSalary()` из класса `Manager` не может непосредственно обратиться к полю `salary`, несмотря на то, что у каждого объекта типа `Manager` имеется поле с таким же именем. Только методы из класса `Employee` имеют доступ к закрытым полям суперкласса. Если же методам из класса `Manager` требуется доступ к закрытым полям, они должны сделать то же, что и любой другой метод: использовать открытый интерфейс (в данном случае метод `getSalary()` из класса `Employee`).

Итак, сделаем еще одну попытку. Вместо непосредственного обращения к полю `salary` попробуем вызвать метод `getSalary()`, как показано ниже.

```
public double getSalary()
{
    double baseSalary = getSalary(); // по-прежнему не работает!
    return baseSalary + bonus;
}
```

Дело в том, что метод `getSalary()` вызывает сам себя, поскольку в классе `Manager` имеется одноименный метод (именно его мы и пытаемся реализовать). В итоге возникает бесконечная цепочка вызовов одного и того же метода, что приводит к аварийному завершению программы.

Нужно найти способ явно указать, что требуется вызвать метод `getSalary()` из суперкласса `Employee`, а не из текущего класса. Для этой цели служит специальное ключевое слово `super`. В приведенной ниже строке кода вызывается метод `getSalary()` именно из класса `Employee`.

```
super.getSalary()
```

Вот как выглядит правильный вариант метода `getSalary()` для класса `Manager`:

```
public double getSalary()
{
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```



НА ЗАМЕТКУ! Некоторые считают ключевое слово `super` аналогом ссылки `this`. Но эта аналогия не совсем точна — ключевое слово `super` не означает ссылку на объект. Например, по нему нельзя присвоить значение другой объектной переменной. Это слово лишь сообщает компилятору, что нужно вызвать метод из суперкласса.

Как видите, в подкласс можно *вводить* поля, а также *вводить* и *переопределять* методы из суперкласса. И в результате наследования ни одно поле или метод из класса не удаляется.



НА ЗАМЕТКУ C++! Для вызова метода из суперкласса в Java служит ключевое слово `super`. А в C++ для этого можно указать имя суперкласса вместе с операцией `::`. Например, метод `getSalary()` из класса `Manager` в C++ можно было бы вызвать следующим образом: `Employee::getSalary()`, вместо `super.getSalary()`.

И наконец, снабдим класс `Manager` конструктором, как показано ниже.

```
public Manager(String n, double s, int year, int month, int day)
{
    super(n, s, year, month, day);
    bonus = 0;
}
```

Здесь ключевое слово `super` имеет уже другой смысл. Приведенное ниже выражение означает вызов конструктора суперкласса `Employee` с параметрами `n`, `s`, `year`, `month` и `day`.

```
super(n, s, year, month, day);
```

Конструктор класса `Manager` не имеет доступа к закрытым полям класса `Employee`, и поэтому он должен инициализировать их, вызывая другой конструктор с помощью ключевого слова `super`. Вызов, содержащий обращение `super`, должен быть первым оператором в конструкторе подкласса.

Если конструктор подкласса не вызывает явно ни одного из конструкторов суперкласса, то из этого суперкласса автоматически вызывается конструктор без аргументов. Если же в суперклассе отсутствует конструктор без аргументов, а конструктор подкласса не вызывает явно другой конструктор из суперкласса, то компилятор Java выдаст сообщение об ошибке.



НА ЗАМЕТКУ! Напомним, что ключевое слово `this` применяется в следующих двух случаях: для указания ссылки на неявный параметр и для вызова другого конструктора того же класса. Аналогично ключевое слово `super` используется в следующих двух случаях: для вызова метода и конструктора из суперкласса. При вызове конструкторов ключевые слова `this` и `super` имеют почти одинаковый смысл. Вызов конструктора должен быть первым оператором в вызывающем конструкторе. Параметры такого конструктора передаются вызывающему конструктору того же класса (`this`) или конструктору суперкласса (`super`).



НА ЗАМЕТКУ C++! Вызов конструктора `super` в C++ не применяется. Вместо этого для создания суперкласса служит список инициализации. В C++ конструктор класса `Manager` выглядел бы следующим образом:

```
Manager::Manager(String n, double s, int year, int month, int day) // C++
: Employee(n, s, year, month, day)
{
    bonus = 0;
}
```

После переопределения метода `getSalary()` для объектов типа `Manager` всем руководящим сотрудникам дополнительно к зарплате будет начислена премия. Обратимся к конкретному примеру, создав объект типа `Manager` и установив величину премии, как показано ниже.

```
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
```

Затем образуем массив, в котором должны храниться три объекта типа `Employee`:

```
Employee[] staff = new Employee[3];
```

Далее заполним его объектами типа `Manager` и `Employee`:

```
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

И наконец, выведем зарплату каждого сотрудника:

```
for (Employee e : staff)
    System.out.println(e.getName() + " " + e.getSalary());
```

В результате выполнения цикла выводятся следующие строки:

```
Carl Cracker 85000.0
Harry Hacker 50000.0
Tommy Tester 40000.0
```

Из элементов массива `staff[1]` и `staff[2]` выводятся основные зарплаты обычных сотрудников, поскольку они представлены объектами типа `Employee`. Но объект из элемента массива `staff[0]` относится к классу `Manager`, и поэтому в методе `getSalary()` из этого класса к основной зарплате прибавляется премия. Обратите особое внимание на следующий вызов:

```
e.getSalary()
```

Здесь вызывается именно тот метод `getSalary()`, который необходим в данном случае. Обратите внимание на то, что объявленным типом переменной `e` является `Employee`, но фактическим типом объекта, на который может ссылаться переменная `e`, может быть как `Employee`, так и `Manager`.

Когда переменная `e` ссылается на объект типа `Employee`, при обращении `e.getSalary()` вызывается метод `getSalary()` из класса `Employee`. Но если переменная `e` ссылается на объект типа `Manager`, то вызывается метод `getSalary()` из класса `Manager`. Виртуальной машине известен фактический тип объекта, на который ссылается переменная `e`, поэтому с вызовом метода никаких затруднений не возникает.

Способность переменной (например, `e`) ссылаться на объекты, имеющие разные фактические типы, называется *полиморфизмом*. Автоматический выбор нужного метода во время выполнения программы называется *динамическим связыванием*. Оба эти понятия будут подробнее обсуждаться далее в главе.



НА ЗАМЕТКУ C++! В Java нет необходимости объявлять метод как виртуальный. Динамическое связывание выполняется по умолчанию. Если же не требуется, чтобы метод был виртуальным, его достаточно объявить с ключевым словом `final` (оно будет рассматриваться далее в этой главе).

В листинге 5.1 представлен пример кода, демонстрирующий отличия в расчете заработной платы для объектов типа `Employee` (листинг 5.2) и `Manager` (листинг 5.3).

Листинг 5.1. Исходный код из файла `inheritance/ManagerTest.java`

```
1 package inheritance;
2
3 /**
4  * В этой программе демонстрируется наследование
5  * @version 1.21 2004-02-21
6  * @author Cay Horstmann
7  */
8 public class ManagerTest
9 {
10     public static void main(String[] args)
11     {
12         // построить объект типа Manager
13         Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
14         boss.setBonus(5000);
15
16         Employee[] staff = new Employee[3];
17
18         // заполнить массив staff объектами типа Manager и Employee
19         staff[0] = boss;
20         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
21         staff[2] = new Employee("Tommy Tester", 40000, 1990, 3, 15);
22     }
23 }
```

```
23 // вывести данные обо всех объектах типа Employee
24 for (Employee e : staff)
25     System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
26 }
27 }
```

Листинг 5.2. Исходный код из файла inheritance/Employee.java

```
1 package inheritance;
2
3 import java.util.Date;
4 import java.util.GregorianCalendar;
5
6 public class Employee
7 {
8     private String name;
9     private double salary;
10    private Date hireDay;
11
12    public Employee(String n, double s, int year, int month, int day)
13    {
14        name = n;
15        salary = s;
16        GregorianCalendar calendar = new GregorianCalendar(year, month-1, day);
17        hireDay = calendar.getTime();
18    }
19
20    public String getName()
21    {
22        return name;
23    }
24    public double getSalary()
25    {
26        return salary;
27    }
28
29    public Date getHireDay()
30    {
31        return hireDay;
32    }
33
34    public void raiseSalary(double byPercent)
35    {
36        double raise = salary * byPercent / 100;
37        salary += raise;
38    }
39 }
```

Листинг 5.3. Исходный код из файла inheritance/Manager.java

```
1 package inheritance;
2
3 public class Manager extends Employee
4 {
5     private double bonus;
6     /**
```

```
7 * @param n Имя сотрудника
8 * @param s Зарплата
9 * @param year Год приема на работу
10 * @param month Месяц приема на работу
11 * @param day День приема на работу
12 */
13 public Manager(String n, double s, int year, int month, int day)
14 {
15     super(n, s, year, month, day);
16     bonus = 0;
17 }
18
19 public double getSalary()
20 {
21     double baseSalary = super.getSalary();
22     return baseSalary + bonus;
23 }
24
25 public void setBonus(double b)
26 {
27     bonus = b;
28 }
29 }
```

Иерархии наследования

Наследование не обязательно ограничивается одним уровнем классов. Например, на основе класса `Manager` можно создать подкласс `Executive`. Совокупность всех классов, производных от общего суперкласса, называется *иерархией наследования*. Условно иерархия наследования показана на рис. 5.1. Путь от конкретного класса к его потомкам в иерархии называется *цепочкой наследования*.

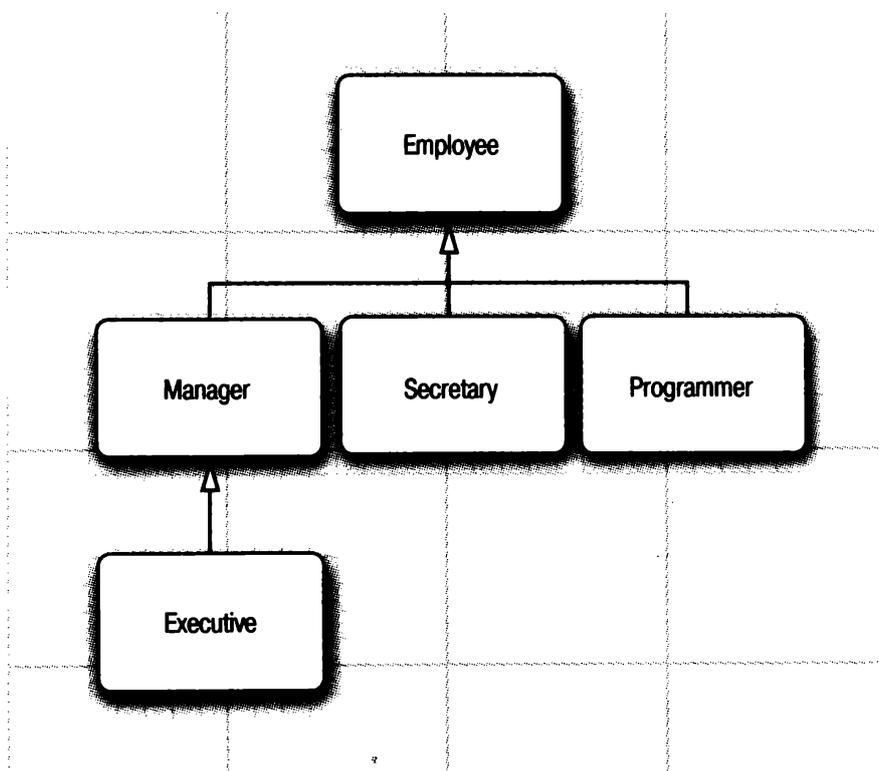


Рис. 5.1. Иерархия наследования для класса `Employee`

Обычно для класса существует несколько цепочек наследования. На основе класса `Employee` для сотрудников можно, например, создать подкласс `Programmer` для программистов или класс `Secretary` для секретарей, причем они будут совершенно независимыми как от класса `Manager` для руководителей, так и друг от друга. Процесс формирования подклассов можно продолжать как угодно долго.



НА ЗАМЕТКУ C++! В Java множественное наследование не поддерживается. Задачи, для которых в других языках применяется множественное наследование, в Java решаются с помощью механизма интерфейсов (интерфейсы будут рассматриваться в начале следующей главы).

Полиморфизм

Существует простое правило, позволяющее определить, стоит ли в конкретной ситуации применять наследование или нет. Если между объектами существует отношение “является”, то каждый объект подкласса *является* объектом суперкласса. Например, каждый руководитель является сотрудником. Следовательно, имеет смысл сделать класс `Manager` подклассом, производным от класса `Employee`. Естественно, обратное утверждение неверно — не каждый сотрудник является руководителем.

Отношение “является” можно выявить и по-другому, используя *принцип подстановки*. Этот принцип состоит в том, что объект подкласса можно использовать вместо любого объекта суперкласса. Например, объект подкласса можно присвоить переменной суперкласса следующим образом:

```
Employee e;  
e = new Employee(...); // предполагается объект класса Employee.  
e = new Manager(...); // допускается использовать и объект класса Manager
```

В Java объектные переменные являются *полиморфными*. Переменная типа `Employee` может ссылаться как на объект класса `Employee`, так и на объект любого подкласса, производного от класса `Employee` (например, `Manager`, `Executive`, `Secretary` и т.п.). Применение принципа *полиморфизма* было продемонстрировано в листинге 5.1 следующим образом:

```
Manager boss = new manager(...);  
Employee[] staff = new Employee[3];  
staff[0] = boss;
```

Здесь переменные `staff[0]` и `boss` ссылаются на один и тот же объект. Но переменная `staff[0]` рассматривается компилятором только как объект типа `Employee`. Это означает, что допускается следующий вызов:

```
boss.setBonus(5000); // Допустимо!
```

В то же время приведенное ниже выражение некорректно.

```
staff[0].setBonus(5000); // ОШИБКА!
```

Дело в том, что переменная `staff[0]` объявлена как объект типа `Employee`, а метод `setBonus()` в этом классе отсутствует. Но присвоить ссылку на объект суперкласса переменной подкласса нельзя. Например, следующий оператор считается недопустимым:

```
Manager m = staff[i]; // ОШИБКА!
```

Причина очевидна: не все сотрудники являются руководителями. Если бы это присваивание произошло и переменная `m` могла бы ссылаться на объект типа `Employee`, который не представляет руководителей, то впоследствии оказался бы возможным вызов метода `m.setBonus(...)`, что привело бы к ошибке при выполнении программы.



ВНИМАНИЕ! В Java массив ссылок на объекты подкласса можно преобразовать в массив ссылок на объекты суперкласса. Для этого приводить типы явным образом не требуется. В качестве примера рассмотрим следующий массив ссылок на объекты типа `Manager`:

```
Manager[] managers = new Manager[10];
```

Вполне допустимо преобразовать его в массив `Employee[]`, как показано ниже.

```
Employee[] staff = managers; // Допустимо!
```

В самом деле, каждый руководитель является сотрудником, а объект типа `Manager` содержит все поля и методы, присутствующие в объекте типа `Employee`. Но тот факт, что переменные `managers` и `staff` ссылаются на один и тот же массив, может привести к неприятным последствиям. Рассмотрим следующее выражение:

```
staff[0] = new Employee("Harry Hacker", ...);
```

Компилятор вполне допускает подобное присваивание. Но `staff[0]` и `manager[0]` — это одна и та же ссылка, поэтому ситуация выглядит так, как будто мы пытаемся присвоить ссылку на объект, описывающий рядового сотрудника, переменной, соответствующей объекту руководителя. В результате формально становится возможным вызов `managers[0].setBonus(1000)`, который повлечет за собой обращение к несуществующему методу и разрушение содержимого памяти.

Для того чтобы предотвратить подобную ситуацию, в созданном массиве запоминается тип элементов и отслеживается допустимость хранящихся ссылок. Так, если массив создан с помощью выражения `new Manager[10]`, то он рассматривается как массив объектов типа `Manager`, и попытка записать в него ссылку на объект типа `Employee` приведет к исключению `ArrayStoreException`, возникающему при нарушении порядка сохранения данных в массиве.

Динамическое связывание

Рассмотрим подробнее, что же происходит при вызове метода, принадлежащего некоторому объекту. При этом выполняются следующие действия.

1. Компилятор проверяет объявленный тип объекта, а также имя метода. Допустим, происходит вызов метода `x.f(param)`, причем неявный параметр `x` объявлен как экземпляр класса `C`. Следует иметь в виду, что может существовать несколько методов под именем `f`, имеющих разные типы параметров (например, методы `f(int)` и `f(String)`). Компилятор нумерует все методы под именем `f` в классе `C` и все доступные методы под именем `f` в суперклассах, производных от класса `C`. (Закрытые методы в суперклассе недоступны.) Итак, компилятору известны все возможные кандидаты на вызов метода под именем `f`.
2. Затем компилятор определяет типы параметров, указанных при вызове метода. Если среди всех методов под именем `f` имеется только один метод, типы параметров которого совпадают с указанными, происходит его вызов. Этот процесс называется *разрешением перегрузки*. Например, при вызове `x.f("Hello")` компилятор выберет метод `f(String)`, а не метод `f(int)`. Но ситуация может осложниться вследствие преобразования типов (`int` — в `double`,

Manager — в Employee и т.д.). Если компилятор не находит ни одного метода с подходящим набором параметров или в результате преобразования типов оказывается несколько методов, соответствующих данному вызову, выдается сообщение об ошибке. Итак, компилятору известно имя и типы параметров метода, который должен быть вызван.



НА ЗАМЕТКУ! Напомним, что имя метода и список типов его параметров образуют *сигнатуру* метода. Например, методы `f(int)` и `f(String)` имеют одинаковые имена, но разные сигнатуры. Если в подклассе определен метод, сигнатура которого совпадает с сигнатурой некоторого метода из суперкласса, то метод подкласса переопределяет его, замещая собой.

Возвращаемый тип не относится к сигнатуре метода. Но при переопределении метода необходимо сохранить совместимость возвращаемых типов. В подклассе возвращаемый тип может быть заменен на подтип исходного типа. Допустим, в классе `Employee` определен метод `getBuddy()` следующим образом:

```
public Employee getBuddy() { ... }
```

Как известно, у начальников не принято заводить приятельские отношения с подчиненными. Для того чтобы отразить этот факт, в подклассе `Manager` метод `getBuddy()` переопределяется следующим образом:

```
public Manager getBuddy() { ... } // Изменение возвращаемого типа допустимо!
```

В этом случае говорят, что для методов `getBuddy()` определены *ковариантные* возвращаемые типы.

3. Если метод является закрытым (`private`), статическим (`static`), конечным (`final`) или конструктором, компилятору точно известно, как его вызвать. (Модификатор доступа `final` описывается в следующем разделе.) Такой процесс называется *статическим связыванием*. В противном случае вызываемый метод определяется по фактическому типу неявного параметра, а во время выполнения программы происходит *динамическое связывание*. В данном примере компилятор сформировал бы вызов метода `f(String)` с помощью динамического связывания.
4. Если при выполнении программы для вызова метода используется динамическое связывание, виртуальная машина должна вызвать версию метода, соответствующую фактическому типу объекта, на который ссылается переменная `x`. Допустим, объект имеет *фактический* тип `D` подкласса, производного от класса `C`. Если в классе `D` определен метод `f(String)`, то вызывается именно он. В противном случае поиск вызываемого метода `f(String)` осуществляется в суперклассе и т.д.

На поиск вызываемого метода уходит слишком много времени, поэтому виртуальная машина заранее создает для каждого класса *таблицу методов*, в которой перечисляются сигнатуры всех методов и фактически вызываемые методы. При вызове метода виртуальная машина просто просматривает таблицу методов. В данном примере виртуальная машина проверяет таблицу методов класса `D` и обнаруживает вызываемый метод `f(String)`. Такими методами могут быть `D.f(String)` или `X.f(String)`, если `X` — некоторый суперкласс для класса `D`. С этим связана одна любопытная особенность. Если вызывается метод `super.f(param)`, то компилятор просматривает таблицу методов суперкласса, на который указывает неявный параметр `super`.

Рассмотрим подробнее вызов метода `e.getSalary()` из листинга 5.1. Переменная `e` имеет тип `Employee`. В этом классе имеется только один метод `getSalary()`, у которого нет параметров. Следовательно, в данном случае можно не беспокоиться о разрешении перегрузки.

При объявлении метода `getSalary()` не указывались ключевые слова `private`, `static` или `final`, поэтому он связывается динамически. Виртуальная машина создает таблицу методов из классов `Employee` и `Manager`. Из таблицы для класса `Employee` следует, что все методы определены в самом классе, как показано ниже.

```
Employee:
  getName() -> Employee.getName()
  getSalary() -> Employee.getSalary()
  getHireDay() -> Employee.getHireDay()
  raiseSalary(double) -> Employee.raiseSalary(double)
```

На самом деле это не совсем так. Как станет ясно в дальнейшем, у класса `Employee` имеется суперкласс `Object`, от которого он наследует большое количество методов. Но мы не будем пока что принимать их во внимание.

Таблица методов из класса `Manager` имеет несколько иной вид. В этом классе три метода наследуются, один метод — переопределяется и еще один — добавляется, как показано ниже.

```
Manager:
  getName() -> Employee.getName()
  getSalary() -> Manager.getSalary()
  getHireDay() -> Employee.getHireDay()
  raiseSalary(double) -> Employee.raiseSalary(double)
  setBonus(double) -> Manager.setBonus(double)
```

Во время выполнения программы вызов метода `e.getSalary()` разрешается следующим образом.

1. Сначала виртуальная машина загружает таблицу методов, соответствующую фактическому типу переменной `e`. Это может быть таблица методов из класса `Employee`, `Manager` или другого подкласса, производного от класса `Employee`.
2. Затем виртуальная машина определяет класс, в котором определен метод `getSalary()` с соответствующей сигнатурой. В итоге вызываемый метод становится известным.
3. И наконец, виртуальная машина вызывает этот метод.

Динамическое связывание обладает одной важной особенностью: оно позволяет видоизменять программы без перекомпиляции их исходного кода. Это делает программы динамически *расширяемыми*. Допустим, в программу добавлен новый класс `Executive`, и переменная `e` может ссылаться на объект этого класса. Код, содержащий вызов метода `e.getSalary()`, заново компилировать не нужно. Если переменная `e` ссылается на объект типа `Executive`, то автоматически вызывается метод `Executive.getSalary()`.



ВНИМАНИЕ! При переопределении область действия метода из подкласса должна быть не меньше области действия метода из суперкласса. Так, если метод из суперкласса был объявлен как `public`, то и метод из подкласса должен быть объявлен как `public`. Программисты часто ошибаются, забывая указать модификатор доступа `public` при объявлении метода в подклассе. В подобных случаях компилятор сообщает, что привилегии доступа к данным ограничены.

Предотвращение наследования: конечные классы и методы

Иногда наследование оказывается нежелательным. Классы, которые нельзя расширить, называются *конечными*. Для указания на это в определении класса используется модификатор доступа `final`. Допустим, требуется предотвратить создание подклассов, производных от класса `Executive`. В таком случае класс `Executive` определяется следующим образом:

```
final class Executive extends Manager
{
    ...
}
```

Отдельный метод класса также может быть конечным. Такой метод не может быть переопределен в подклассах. (Все методы конечного класса автоматически являются конечными.) Ниже приведен пример объявления конечного метода.

```
class Employee
{
    ...
    public final String getName()
    {
        return name;
    }
    ...
}
```



НА ЗАМЕТКУ! Напомним, что с помощью модификатора доступа `final` могут быть также описаны поля, являющиеся константами. После создания объекта значение такого поля нельзя изменить. Но если класс объявлен как `final`, то конечными автоматически становятся только его методы, но не поля.

Существует единственный аргумент в пользу указания ключевого слова `final` при объявлении метода или класса: гарантия неизменности семантики в подклассе. Так, методы `getTime()` и `setTime()` являются конечными в классе `Calendar`. Поступая подобным образом, разработчики данного класса берут на себя ответственность за корректность преобразования содержимого объекта типа `Date` в состояние календаря. В подклассах невозможно изменить принцип преобразования. В качестве другого примера можно привести конечный класс `String`. Создавать подклассы, производные от этого класса, запрещено. Следовательно, если имеется переменная типа `String`, то можно не сомневаться, что она ссылается именно на символьную строку, а не на что-нибудь другое.

Некоторые программисты считают, что ключевое слово `final` следует применять при объявлении всех методов. Исключением из этого правила являются только те случаи, когда имеются веские основания для применения принципа полиморфизма. Действительно, в C++ и C# для применения принципа полиморфизма в методах необходимо принимать специальные меры. Возможно, данное правило и слишком жесткое, но несомненно одно: при составлении иерархии классов следует серьезно подумать о целесообразности применения конечных классов и методов.

На заре развития Java некоторые программисты пытались использовать ключевое слово `final` для того, чтобы исключить издержки, обусловленные динамическим связыванием. Если метод не переопределяется и невелик, компилятор применяет процедуру оптимизации, которая состоит в непосредственном *встраивании*

кода. Например, вызов метода `e.getName()` заменяется доступом к полю `e.name`. Такое усовершенствование вполне целесообразно, поскольку ветвление программы несовместимо с упреждающей загрузкой команд, применяемой в процессорах. Но, если метод `getName()` будет переопределен, компилятор не сможет непосредственно встроить его код. Ведь ему неизвестно, какой из методов должен быть вызван при выполнении программы.

Правда, компонент виртуальной машины, называемый *динамическим компилятором*, может выполнять оптимизацию более эффективно, чем обыкновенный компилятор. Ему точно известно, какие именно классы расширяют данный класс, и он может проверить, действительно ли метод переопределяется. Если же метод невелик, часто вызывается и не переопределен, то динамический компилятор выполнит непосредственное встраивание его кода. Но что произойдет, если виртуальная машина загрузит другой подкласс, который переопределяет встраиваемый метод? Тогда оптимизатор должен отменить встраивание кода. В подобных случаях выполнение программы замедляется, хотя это происходит редко.

Приведение типов

В главе 3 был рассмотрен процесс принудительного преобразования одного типа в другой, называемый *приведением типов*. Для этой цели в Java предусмотрена специальная запись. Например, при выполнении следующего фрагмента кода значение переменной `x` преобразуется в целочисленное отбрасыванием дробной части:

```
double x = 3.405;
int nx = (int) x;
```

И как иногда возникает потребность в преобразовании значения с плавающей точкой в целочисленное, ссылке на объект требуется порой привести к типу другого класса. Для такого приведения типов служат те же самые синтаксические конструкции, что и для числовых выражений. С этой целью имя нужного класса следует заключить в скобки и поставить перед той ссылкой на объект, которую требуется привести к искомому типу. Ниже приведен соответствующий тому пример.

```
Manager boss = (Manager) staff[0];
```

Для такого приведения типов существует только одна причина: необходимость использовать все функциональные возможности объекта после того, как его фактический тип был на время забыт. Например, в классе `ManagerTest` массив `staff` содержит объекты типа `Employee`. Этот тип выбран потому, что в некоторых элементах данного массива хранятся данные о рядовых сотрудниках. А для того чтобы получить доступ ко всем новым полям из класса `Manager`, скорее всего, придется привести некоторые элементы массива `staff` к типу `Manager`. (В примере кода, рассмотренном в начале этой главы, были приняты специальные меры, чтобы избежать приведения типов. В частности, переменная `boss` была инициализирована объектом типа `Manager`, перед тем как разместить ее в массиве. Для того чтобы задать величину премии руководящего сотрудника, нужно знать правильный тип соответствующего объекта.)

Как известно, у каждой объектной переменной в Java имеется свой тип. Тип объектной переменной определяет разновидность объекта, на который ссылается эта переменная, а также ее функциональные возможности. Например, переменная `staff[1]` ссылается на объект типа `Employee`, поэтому она может ссылаться и на объект типа `Manager`.

В процессе своей работы компилятор проверяет, не обещаете ли вы слишком много, сохраняя значение в переменной. Так, если вы присваиваете переменной суперкласса ссылку на объект подкласса, то обещаете *меньше* положенного, и компилятор просто разрешает вам сделать это. А если вы присваиваете объект суперкласса переменной подкласса, то обещаете *больше* положенного, и поэтому вы должны подтвердить свои обещания, указав в скобках имя класса для приведения типов. Таким образом, виртуальная машина получает возможность контролировать ваши действия при выполнении программы.

А что произойдет, если попытаться осуществить приведение типов вниз по цепочке наследования и попробовать обмануть компилятор в отношении содержимого объекта, как в приведенной ниже строке кода?

```
Manager boss = (Manager) staff[1]; /* ОШИБКА!
```

При выполнении программы система обнаружит несоответствие и сгенерирует исключение типа `ClassCastException`. Если его не перехватить, нормальное выполнение программы будет прервано. Таким образом, перед приведением типов следует непременно проверить его корректность. Для этой цели служит операция `instanceof`, как показано ниже.

```
if (staff[1] instanceof Manager)
{
    boss = (Manager) staff[1];
    . . .
}
```

И наконец, компилятор не позволит выполнить некорректное приведение типов, если для этого нет никаких оснований. Например, наличие приведенной ниже строки в исходном коде программы приведет к ошибке во время компиляции, поскольку класс `Date` не является подклассом, производным от класса `Employee`.

```
Date c = (Date) staff[1];
```

Таким образом, можно сформулировать следующие основные правила приведения типов при наследовании.

- Приведение типов можно выполнять только в иерархии наследования.
- Для того чтобы проверить корректность приведения суперкласса к подклассу, следует выполнить операцию `instanceof`.



НА ЗАМЕТКУ! Если в приведенном ниже выражении `x` содержит пустое значение `null`, исключение не будет сгенерировано, но лишь возвратится логическое значение `false`.

```
x instanceof C
```

И это вполне логично. Ведь пустая ссылка типа `null` не указывает ни на один из объектов, а следовательно, она не указывает ни на один из объектов типа `C`.

На самом деле приведение типов при наследовании — не самое лучшее решение. В данном примере выполнять преобразование объекта типа `Employee` в объект типа `Manager` совсем не обязательно. Метод `getSalary()` вполне способен оперировать объектами обоих типов, поскольку при динамическом связывании правильный метод автоматически определяется благодаря принципу полиморфизма.

Приведение типов целесообразно лишь в том случае, когда для объектов, представляющих руководителей, требуется вызвать особый метод, имеющийся только в классе `Manager`, например метод `setBonus()`. Если же по какой-нибудь причине потребуется вызвать метод `setBonus()` для объекта типа `Employee`, следует задать себе вопрос: не свидетельствует ли это о недостатках суперкласса? Возможно, имеет смысл пересмотреть структуру суперкласса и добавить в него метод `setBonus()`. Не забывайте, что для преждевременного завершения программы достаточно единственного неперехваченного исключения типа `ClassCastException`. А в целом при наследовании лучше свести к минимуму приведение типов и выполнение операции `instanceof`.



НА ЗАМЕТКУ C++! В Java для приведения типов служит устаревший синтаксис языка C, но он действует подобно безопасной операции `dynamic_cast` динамического приведения типов в C++. Например, приведенные ниже строки кода, написанные на разных языках, почти равнозначны.

```
Manager boss = (Manager)staff[1]; // Java
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
```

Но у них имеется одно важное отличие. Если приведение типов завершается неудачно, то вместо пустого объекта генерируется исключение. В этом смысле приведение типов в Java напоминает приведение ссылок в C++. И это весьма существенный недостаток языка Java. Ведь в C++ контроль и преобразование типов можно выполнить в одной операции следующим образом:

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
if (boss != NULL) ...
```

А в Java для этой цели приходится сочетать операцию `instanceof` с приведением типов, как показано ниже.

```
if (staff[1] instanceof Manager) // Java
{
    Manager boss = (Manager)staff[1];
    ...
}
```

Абстрактные классы

Чем дальше вверх по иерархии наследования, тем более универсальными и абстрактными становятся классы. В некотором смысле родительские классы, находящиеся на верхней ступени иерархии, становятся настолько абстрактными, что их рассматривают как основу для разработки других классов, а не как классы, позволяющие создавать конкретные объекты. Например, сотрудник — это человек, а человек может быть и студентом. Поэтому расширим иерархию, в которую входит класс `Employee`, добавив в нее классы `Person` и `Student`. Отношения наследования между всеми этими классами показаны на рис. 5.2.

А зачем вообще столь высокий уровень абстракции? Существуют определенные свойства, характерные для каждого человека, например имя. Ведь у сотрудников вообще и у студентов в частности имеются свои имена, и поэтому при внедрении общего суперкласса придется перенести метод `getName()` на более высокий уровень в иерархии наследования.

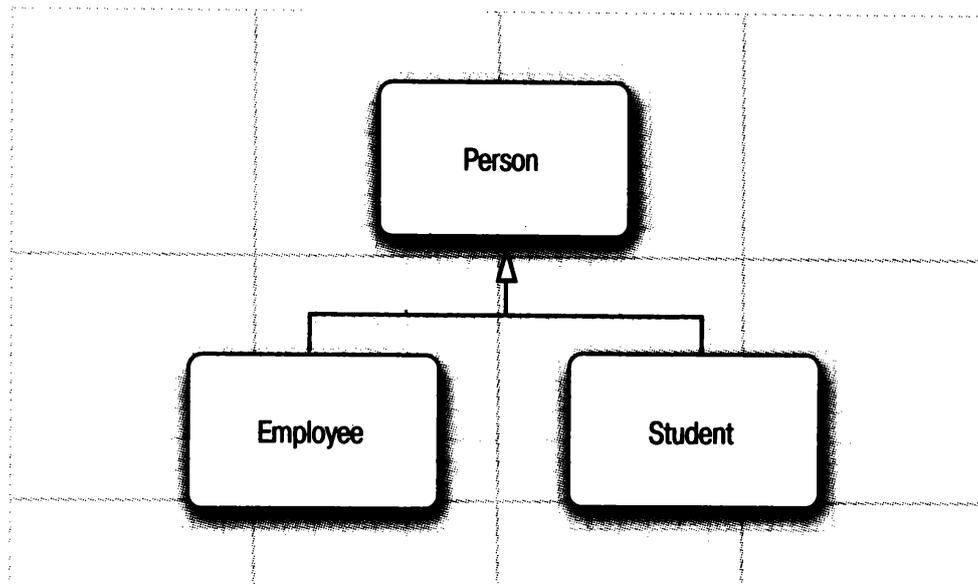


Рис. 5.2. Блок-схема, демонстрирующая иерархию наследования для класса `Person` и его подклассов

А теперь введем новый метод `getDescription()`, предназначенный для составления краткой характеристики человека, например:

```
an employee with a salary of $50,000.00
(сотрудник с зарплатой 50 тыс. долларов США)
a student majoring in computer science
(студент, изучающий вычислительную технику)
```

Для классов `Employee` и `Student` такой метод реализуется довольно просто. Но какие сведения о человеке следует разместить в класс `Person`? Ведь в нем ничего нет, кроме имени. Разумеется, можно было бы реализовать метод `Person.getDescription()`, возвращающий пустую строку. Но есть способ получше. От реализации этого метода в классе `Person` можно вообще отказаться, если воспользоваться ключевым словом `abstract`, как показано ниже.

```
public abstract String getDescription();
    // реализация не требуется
```

Для большей ясности класс, содержащий один или несколько абстрактных методов, можно объявить абстрактным следующим образом:

```
abstract class Person
{
    ...
    public abstract String getDescription();
}
```

Помимо абстрактных методов, абстрактные классы могут содержать конкретные поля и методы. Например, в классе `Person` хранится имя человека и содержится конкретный метод, возвращающий это имя, как показано ниже.

```
abstract class Person
{
    private String name;
    public Person(String n)
    {
        name = n;
    }
}
```

```
public abstract String getDescription();

public String getName()
{
    return name;
}
}
```



СОВЕТ. Некоторые программисты не осознают, что абстрактные классы могут содержать конкретные методы. Общие поля и методы (будь то абстрактные или конкретные) следует всегда перемещать в суперкласс, каким бы он ни был: абстрактным или конкретным.

Абстрактные методы представляют собой прототипы методов, реализованных в подклассах. Расширяя абстрактный класс, можно оставить некоторые или все абстрактные методы неопределенными. При этом подкласс также станет абстрактным. Если даже определить все методы, то и тогда подкласс не перестанет быть абстрактным.

Определим, например, класс `Student`, расширяющий абстрактный класс `Person` и реализующий метод `getDescription()`. Ни один из методов в классе `Student` не является абстрактным, поэтому нет никакой необходимости объявлять сам класс абстрактным. Впрочем, класс может быть объявлен абстрактным, даже если он и не содержит ни одного абстрактного метода.

Создать экземпляры абстрактного класса нельзя. Например, приведенное ниже выражение ошибочно. Но можно создать объекты конкретного подкласса.

```
new Person("Vince Vu");
```

Следует иметь в виду, что для абстрактных классов *можно* создавать объектные переменные, но такие переменные должны ссылаться на объект неабстрактного класса. Рассмотрим следующую строку кода:

```
Person p = new Student("Vince Vu", "Economics");
```

где `p` — переменная абстрактного типа `Person`, ссылающаяся на экземпляр неабстрактного подкласса `Student`.



НА ЗАМЕТКУ C++! В C++ абстрактный метод называется чистой виртуальной функцией. Его обозначение оканчивается символами `=0`, как показано ниже.

```
class Person // C++
{
    public:
        virtual string getDescription() = 0;
        ...
};
```

Класс в C++ является абстрактным, если он содержит хотя бы одну чистую виртуальную функцию. В C++ отсутствует специальное ключевое слово для обозначения абстрактных классов.

Определим конкретный подкласс `Student`, расширяющий абстрактный класс `Person`, следующим образом:

```
class Student extends Person
{
    private String major;
```

```
public Student(String n, String m)
{
    super(n);
    major = m;
}

public String getDescription()
{
    return "a student majoring in " + major;
}
}
```

В этом подклассе определяется метод `getDescription()`. Все методы из класса `Student` являются конкретными, а следовательно, класс больше не является абстрактным.

В примере программы из листинга 5.4 определяются один абстрактный суперкласс `Person` (из листинга 5.5) и два конкретных подкласса `Employee` (из листинга 5.6) и `Student` (из листинга 5.7). Сначала в этой программе массив типа `Person` заполняется ссылками на экземпляры классов `Employee` и `Student`:

```
Person[] people = new Person[2];
people[0] = new Employee(. . .);
people[1] = new Student(. . .);
```

Затем имена и описания сотрудников, представленных этими объектами, выводятся следующим образом:

```
for (Person p : people)
    System.out.println(p.getName() + ", " + p.getDescription());
```

Некоторых читателей присутствие вызова `p.getDescription()` может озадачить. Не относится ли он к неопределенному методу? Следует иметь в виду, что переменная `p` вообще не ссылается ни на один из объектов абстрактного класса `Person`, поскольку создать такой объект просто невозможно. Переменная `p` ссылается на объект конкретного подкласса, например `Employee` или `Student`. А для этих объектов метод `getDescription()` определен.

Можно ли пропустить в классе `Person` все абстрактные методы и определить метод `getDescription()` в подклассах `Employee` и `Student`? Это не будет ошибкой, но тогда метод `getDescription()` нельзя будет вызвать с помощью переменной `p`. Компилятор гарантирует, что вызываются только те методы, которые определены в классе.

Абстрактные методы являются важным понятием языка Java. Они в основном применяются при создании *интерфейсов*, которые будут подробно рассмотрены в главе 6.

Листинг 5.4. Исходный код из файла `abstractClasses/PersonTest.java`

```
1 package abstractClasses;
2 /**
3  * В этой программе демонстрируется применение абстрактных классов
4  * @version 1.01 2004-02-21
5  * @author Cay Horstmann
6  */
7 public class PersonTest
8 {
9     public static void main(String[] args)
```

```
10 {
11     Person[] people = new Person[2];
12
13     // заполнить массив people объектами типа Student и Employee
14     people[0] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
15     people[1] = new Student("Maria Morris", "computer science");
16
17     // вывести имена и описания всех лиц,
18     // представленных объектами типа Person
19     for (Person p : people)
20         System.out.println(p.getName() + ", " + p.getDescription());
21 }
22 }
```

Листинг 5.5. Исходный код из файла `abstractClasses/Person.java`

```
1 package abstractClasses;
2
3 public abstract class Person
4 {
5     public abstract String getDescription();
6     private String name;
7
8     public Person(String n)
9     {
10         name = n;
11     }
12
13     public String getName()
14     {
15         return name;
16     }
17 }
```

Листинг 5.6. Исходный код из файла `abstractClasses/Employee.java`

```
1 package abstractClasses;
2
3 import java.util.Date;
4 import java.util.GregorianCalendar;
5
6 public class Employee extends Person
7 {
8     private double salary;
9     private Date hireDay;
10
11     public Employee(String n, double s, int year, int month, int day)
12     {
13         super(n);
14         salary = s;
15         GregorianCalendar calendar = new GregorianCalendar(year, month-1, day);
16         hireDay = calendar.getTime();
17     }
18     public double getSalary()
19     {
```

```
20     return salary;
21 }
22
23 public Date getHireDay()
24 {
25     return hireDay;
26 }
27
28 public String getDescription()
29 {
30     return String.format("an employee with a salary of $%.2f", salary);
31 }
32
33 public void raiseSalary(double byPercent)
34 {
35     double raise = salary * byPercent / 100;
36     salary += raise;
37 }
38 }
```

Листинг 5.7. Исходный код из файла abstractClasses/Student.java

```
1 package abstractClasses;
2
3 public class Student extends Person
4 {
5     private String major;
6
7     /**
8      * @param n Имя студента
9      * @param m Специализация студента
10    */
11    public Student(String n, String m)
12    {
13        // передать строку n конструктору суперкласса в качестве параметра
14        super(n);
15        major = m;
16    }
17
18    public String getDescription()
19    {
20        return "a student majoring in " + major;
21    }
22 }
```

Защищенный доступ

Как известно, поля в классе желательно объявлять как `private`, а некоторые методы — как `public`. Любые закрытые (т.е. `private`) компоненты программы невидимы из других классов. Это утверждение справедливо и для подклассов: у подкласса отсутствует доступ к закрытым полям суперкласса.

Но иногда приходится ограничивать доступ к некоторому методу и открывать его лишь для подклассов. Реже возникает потребность предоставлять методам из подкласса доступ к полям в суперклассе. В таком случае компонент класса объявляется

защищенным с помощью ключевого слова `protected`. Так, если поле `hireDay` объявлено в суперклассе `Employee` защищенным, а не закрытым, методы из подкласса `Manager` смогут обращаться к нему непосредственно.

Но методам из класса `Manager` доступны лишь поля `hireDay`, принадлежащие объектам самого класса `Manager`, но не класса `Employee`. Это ограничение введено в качестве предупредительной меры против злоупотреблений механизмом защищенного доступа, не позволяющим создавать подклассы лишь для того, чтобы получить доступ к защищенным полям.

На практике пользоваться защищенными полями следует очень аккуратно. Допустим, созданный вами класс, в котором имеются защищенные поля, используется другими разработчиками. Без вашего ведома другие могут создавать подклассы, производные от вашего класса, тем самым получая доступ к защищенным полям. В таком случае вы уже не сможете изменить реализацию своего класса, не уведомив об этом других заинтересованных лиц. Но это противоречит самому духу ООП, поощряющему инкапсуляцию данных.

Применение защищенных методов более оправданно. Метод можно объявить в классе защищенным, чтобы ограничить его применение. Это означает, что в методах подклассов, предшественники которых известны изначально, можно вызвать защищенный метод, а методы других классов — нельзя. В качестве примера можно привести защищенный метод `clone()` из класса `Object`, более подробно рассматриваемый в главе 6.



НА ЗАМЕТКУ C++! В Java защищенные компоненты программ доступны из всех подклассов, а также из других классов того же самого пакета. Этим Java отличается от C++, где ключевое слово `protected` имеет несколько иной смысл. Таким образом, в Java ограничения на доступ к защищенным элементам менее строги, чем в C++.

Итак, в Java предоставляются следующие четыре модификатора доступа, определяющие границы области действия компонентов программы.

1. Модификатор доступа `private` — ограничивает область действия классом.
2. Модификатор доступа `public` — не ограничивает область действия.
3. Модификатор доступа `protected` — ограничивает область действия пакетом и всеми подклассами.
4. Модификатор доступа *отсутствует* — область действия ограничивается пакетом (к сожалению) по умолчанию.

Глобальный суперкласс `Object`

Класс `Object` является исходным предшественником всех остальных классов, поэтому каждый класс в Java расширяет класс `Object`. Но явно отражать этот факт, как в приведенной ниже строке кода, совсем не обязательно.

```
class Employee extends Object
```

Если суперкласс явно не указан, им считается класс `Object`. А поскольку каждый класс в Java расширяет класс `Object`, то очень важно знать, какими средствами обладает сам класс `Object`. В этой главе мы дадим беглый обзор самых основных из

них, прочие средства класса Object будут рассмотрены в других главах, а остальные сведения о них вы можете найти в соответствующей документации. (Некоторые методы из класса Object целесообразно рассматривать лишь вместе с потоками, которые обсуждаются в главе 14.)

Переменную типа Object можно использовать в качестве ссылки на объект любого типа следующим образом:

```
Object obj = new Employee("Harry Hacker", 35000);
```

Разумеется, переменная этого класса полезна лишь как средство для хранения значений произвольного типа. Чтобы сделать с этим значением что-то конкретное, нужно знать его исходный тип, а затем выполнить приведение типов, как показано ниже. В Java объектами *не* являются только *примитивные* типы: числа, символы и логические значения.

```
Employee e = (Employee) obj;
```

Все массивы относятся к типам объектов тех классов, которые расширяют класс Object, независимо от того, содержатся ли в элементах массива объекты или простые типы.

```
Employee[] staff = new Employee[10];  
obj = staff; // Допустимо!  
obj = new int[10]; // Допустимо!
```



НА ЗАМЕТКУ C++! В C++ аналогичного глобального базового класса нет, хотя любой указатель можно, конечно, преобразовать в указатель типа void*.

Метод equals ()

В методе equals () из класса Object проверяется, равнозначны ли два объекта. А поскольку метод equals () реализован в классе Object, то в нем определяется только следующее: ссылаются ли переменные на один и тот же объект. В качестве проверки по умолчанию эти действия вполне оправданы: всякий объект равнозначен самому себе. Для некоторых классов большего и не требуется. Например, вряд ли кому-то потребуется анализировать два объекта типа PrintStream и выяснять, отличаются ли они чем-нибудь. Но в ряде случаев равнозначными должны считаться объекты одного типа, находящиеся в одном и том же состоянии.

Рассмотрим в качестве примера объекты, описывающие сотрудников. Очевидно, что они одинаковы, если совпадают имена, размеры заработной платы и даты их приема на работу. (Строго говоря, в настоящих системах учета данных о сотрудниках более оправдано сравнение идентификационных номеров. А здесь лишь демонстрируются принципы реализации метода equals () в коде, как показано ниже.)

```
class Employee  
{  
    public boolean equals(Object otherObject)  
    {  
        // быстро проверить объекты на идентичность  
        if (this == otherObject) return true;  
  
        // вернуть логическое значение false,  
        // если явный параметр имеет пустое значение null  
        if (otherObject == null) return false;
```

```

// если классы не совпадают, они не равнозначны
if (getClass() != otherObject.getClass())
    return false;

// Теперь известно, что объект otherObject
// относится к типу Employee и не является пустым
Employee other = (Employee) otherObject;

// проверить, хранятся ли в полях объектов одинаковые значения
return name.equals(other.name)
    && salary == other.salary
    && hireDay.equals(other.hireDay);
}
}

```

Метод `getClass()` возвращает класс объекта. Этот метод будет подробно обсуждаться далее в главе. Для того чтобы объекты можно было считать равнозначными, они как минимум должны быть объектами одного и того же класса.



СОВЕТ. Для того чтобы в поле `name` или `hireDay` не оказалось пустого значения `null`, воспользуйтесь методом `Objects.equals()`. В результате вызова метода `Objects.equals(a, b)` возвращается логическое значение `true`, если оба его аргумента имеют пустое значение `null`; логическое значение `false`, если только один из аргументов имеет пустое значение `null`; а иначе делается вызов `a.equals(b)`. С учетом этого последний оператор в теле приведенного выше метода `Employee.equals()` приобретает следующий вид:

```

return Objects.equals(name, other.name)
    && salary == other.salary
    && Object.equals(hireDay, other.hireDay);

```

Определяя метод `equals()` для подкласса, сначала следует вызывать одноименный метод из суперкласса. Если проверка даст отрицательный результат, объекты нельзя считать равнозначными. Если же поля суперкласса совпадают, можно приступить к сравнению полей подкласса, как показано ниже.

```

class Manager extends Employee
{
    . . .
    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) return false;
        // При вызове метода super.equals() проверяется,
        // принадлежит ли объект otherObject тому же самому классу
        Manager other = (Manager) otherObject;
        return bonus == other.bonus;
    }
}

```

Проверка равнозначности объектов и наследование

Каким образом должен действовать метод `equals()`, если неявные и явные его параметры не принадлежат одному и тому же классу? Это спорный вопрос. В предыдущем примере из метода `equals()` возвращалось логическое значение `false`, если

классы не совпадали полностью. Но многие программисты пользуются следующей проверкой:

```
if (!(otherObject instanceof Employee)) return false;
```

При этом остается вероятность, что объект `otherObject` принадлежит подклассу. Поэтому такой подход может стать причиной непредвиденных осложнений. Спецификация Java требует, чтобы метод `equals()` обладал следующими характеристиками.

1. *Рефлексивность*. При вызове `x.equals(x)` по любой ненулевой ссылке `x` должно возвращаться логическое значение `true`.
2. *Симметричность*. При вызове `x.equals(y)` по любым ссылкам `x` и `y` должно возвращаться логическое значение `true` тогда и только тогда, когда при вызове `y.equals(x)` возвращается логическое значение `true`.
3. *Транзитивность*. Если при вызовах `x.equals(y)` и `y.equals(z)` по любым ссылкам `x`, `y` и `z` возвращается логическое значение `true`, то и при вызове `x.equals(z)` возвращается логическое значение `true`.
4. *Согласованность*. Если объекты, на которые делаются ссылки `x` и `y`, не изменяются, то при повторном вызове `x.equals(y)` должно возвращаться то же самое значение.
5. При вызове `x.equals(null)` по любой непустой ссылке `x` должно возвращаться логическое значение `false`.

Обоснованность приведенных выше правил не вызывает сомнений. Так, совершенно очевидно, что результаты проверки не должны зависеть от того, делается ли в программе вызов `x.equals(y)` или `y.equals(x)`. Но применение правила симметрии имеет свои особенности, если явный и неявный параметры принадлежат разным классам. Рассмотрим следующий вызов:

```
e.equals(m)
```

где объект `e` принадлежит классу `Employee`, а объект `m` — классу `Manager`, причем каждый из них содержит одинаковые имена, зарплату и дату приема на работу. Если при вызове `e.equals(m)` выполняется проверка с помощью операции `instanceof`, то возвращается логическое значение `true`. Но это означает, что и при обратном вызове `m.equals(e)` также должно возвращаться логическое значение `true`, поскольку правило симметричности не позволяет возвращать логическое значение `false` или генерировать исключение.

В итоге класс `Manager` попадает в затруднительное положение. Его метод `equals()` должен сравнивать объект данного класса с любым объектом типа `Employee` без учета данных, позволяющих отличить руководящего сотрудника от рядового! И в этом случае операция `instanceof` выглядит менее привлекательно.

Некоторые специалисты считают, что проверка с помощью метода `getClass()` некорректна, поскольку в этом случае нарушается принцип подстановки. В подтверждение они приводят метод `equals()` из класса `AbstractSet`, который проверяет, содержат ли два множества одинаковые элементы и расположены ли они в одинаковом порядке. Класс `AbstractSet` выступает в роли суперкласса для классов `TreeSet` и `HashSet`, которые не являются абстрактными. В этих классах применяются различные алгоритмы для обращения к элементам множества. Но на практике

необходимо иметь возможность сравнивать любые два множества, независимо от того, как они реализованы.

Следует, однако, признать, что рассматриваемый здесь пример слишком специфичен. В данном случае имело бы смысл объявить метод `AbstractSet.equals()` конечным, чтобы нельзя было изменить семантику проверки множеств на равнозначность. (На самом деле при объявлении метода ключевое слово `final` не указано. Это дает возможность подклассам реализовать более эффективный алгоритм проверки на равнозначность.)

Таким образом, вырисовываются два разных сценария.

- Если проверка на равнозначность реализована в подклассе, правило симметричности требует использовать метод `getClass()`.
- Если проверка производится средствами суперкласса, можно выполнить операцию `instanceof`. В этом случае возможна ситуация, когда два объекта разных классов будут признаны равнозначными.

В примере с рядовыми и руководящими сотрудниками два объекта считаются равнозначными, если их поля совпадают. Так, если имеются два объекта типа `Manager` с одинаковыми именами, заработной платой и датой приема на работу, но с отличающимися величинами премии, такие объекты следует признать разными. А для этого требуется проверка с помощью метода `getClass()`. Но допустим, что для проверки на равнозначность используется идентификационный номер сотрудника. Такая проверка имеет смысл для всех подклассов. В этом случае можно выполнить операцию `instanceof` и объявить метод `Employee.equals()` как `final`.



НА ЗАМЕТКУ! В стандартной библиотеке Java содержится более 150 реализаций метода `equals()`. В одних из них применяются в различных сочетаниях операции `instanceof`, вызовы метода `getClass()` и фрагменты кода, предназначенные для обработки исключения `ClassCastException`, а в других не выполняется практически никаких действий. Обратитесь за справкой к документации на класс `java.sql.Timestamp`, где указывается на некоторые непреодолимые трудности реализации. В частности, класс `Timestamp` наследует от класса `java.util.Date`, где в методе `equals()` организуется проверка с помощью операции `instanceof`, но переопределить этот метод симметрично и точно не представляется возможным.

Ниже приведены рекомендации для создания приближающегося к идеалу метода `equals()`.

1. Присвойте явному параметру имя `otherObject`. Впоследствии его тип нужно будет привести к типу другой переменной под названием `other`.
2. Проверьте, одинаковы ли ссылки `this` и `otherObject`, следующим образом:

```
if (this == otherObject) return true;
```

Это выражение составлено лишь в целях оптимизации проверки. Ведь намного быстрее проверить одинаковость ссылок, чем сравнивать поля объектов.

3. Выясните, является ли ссылка `otherObject` пустой (`null`), как показано ниже. Если она является таковой, следует вернуть логическое значение `false`. Эту проверку нужно сделать обязательно.

```
if (otherObject == null) return false;
```

4. Сравните классы `this` и `otherObject`. Если семантика проверки может измениться в подклассе, воспользуйтесь методом `getClass()` следующим образом:

```
if (getClass() != otherObject.getClass()) return false;
```

Если одна и та же семантика остается справедливой для *всех* подклассов, произведите проверку с помощью операции `instanceof` следующим образом:

```
if (!(otherObject instanceof ClassName)) return false;
```

5. Приведите тип объекта `otherObject` к типу переменной требуемого класса, как показано ниже.

```
ИмяКласса other = (ИмяКласса)otherObject;
```

6. Сравните все поля, как показано ниже. Для полей примитивных типов служит операция `==`, а для объектных полей — метод `Objects.equals()`. Если все поля двух объектов совпадают, возвращается логическое значение `true`, а иначе — логическое значение `false`.

```
return поле1 == other.поле1
    && поле2.equals(other.поле2)
    && ...;
```

Если вы переопределяете в подклассе метод `equals()`, в него следует включить вызов `super.equals(other)`.



СОВЕТ. Если имеются поля типа массива, для проверки равнозначности соответствующих элементов массива можно воспользоваться статическим методом `Arrays.equals()`.



ВНИМАНИЕ! Реализуя метод `equals()`, многие программисты допускают типичную ошибку. Сможете ли вы сами выяснить, какая ошибка возникнет при выполнении следующего фрагмента кода?

```
public class Employee
{
    public boolean equals(Employee other)
    {
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
    ...
}
```

В этом методе тип явного параметра определяется как `Employee`. В итоге переопределяется не метод `equals()` из класса `Object`, а совершенно посторонний метод. Застраховаться от возникновения подобной ошибки можно, специально обозначив метод, который переопределяет соответствующий метод из суперкласса, дескриптором `@Override`, как показано ниже.

```
@Override public boolean equals(Object other)
```

Если при этом будет случайно определен новый метод, компилятор возвратит сообщение об ошибке. Допустим, в классе `Employee` присутствует такая строка кода:

```
@Override public boolean equals(Employee other)
```

Этот метод не переопределяет ни одного из методов суперкласса `Object`, и поэтому будет обнаружена ошибка.

java.util.Arrays 1.2

- `static boolean equals(type[] a, type[] b) 5.0`

Возвращает логическое значение `true`, если массивы имеют одинаковую длину и одинаковые элементы на соответствующих позициях. Массивы могут содержать компоненты типа `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`.

java.util.Objects 7

- `static boolean equals(Object a, Object b)`

Возвращает логическое значение `true`, если оба параметра, `a` и `b`, имеют пустое значение `null`; логическое значение `false`, если один из них имеет пустое значение `null`; а иначе — результат вызова `a.equals(b)`.

Метод hashCode ()

Хеш-код — это целое число, генерируемое на основе конкретного объекта. Хеш-код можно рассматривать как некоторый шифр: если `x` и `y` — разные объекты, то с большой степенью вероятности должны различаться результаты вызовов `x.hashCode()` и `y.hashCode()`. В табл. 5.1 приведено несколько примеров хеш-кодов, полученных в результате вызова метода `hashCode()` из класса `String`.

Таблица 5.1. Хеш-коды, получаемые с помощью метода `hashCode()`

Символьная строка	Хеш-код
Hello	69609650
Harry	69496448
Hacker	-2141031506

Для вычисления хеш-кода в классе `String` применяется следующий алгоритм:

```
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```

Метод `hashCode()` определен в классе `Object`. Поэтому у каждого объекта имеется хеш-код, определяемый по умолчанию. Этот хеш-код вычисляется по адресу памяти, занимаемой объектом. Рассмотрим следующий пример кода:

```
String s = "Ok";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = new String("Ok");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Результаты выполнения этого фрагмента кода приведены в табл. 5.2.

Таблица 5.2. Хеш-коды для объектов типа String и StringBuffer

Объект	Хеш-код
s	2556
sb	20526976
t	2556
tb	20527144

Обратите внимание на то, что символьным строкам `s` и `t` соответствуют одинаковые хеш-коды, поскольку они вычисляются на основе содержимого строкового объекта. А у строителей строк `sb` и `tb` хеш-коды отличаются. Дело в том, что в классе `StringBuilder` метод `hashCode()` не определен, и поэтому из класса `Object` вызывается исходный метод `hashCode()`, где хеш-код определяется по адресу памяти, занимаемой объектом.

Если вы переопределяете метод `equals()`, вам следует переопределить и метод `hashCode()` для объектов, которые пользователи могут вставлять в хеш-таблицу. (Подробнее хеш-таблицы будут обсуждаться в главе 13.)

Метод `hashCode()` должен возвращать целочисленное значение, которое может быть отрицательным. Для того чтобы хеш-коды разных объектов отличались, достаточно объединить хеш-коды полей экземпляра. Ниже приведен пример реализации метода `hashCode` в классе `Employee`.

```
class Employee
{
    public int hashCode()
    {
        return 7 * name.hashCode()
            + 11 * new Double(salary).hashCode()
            + 13 * hireDay.hashCode();
    }
    . . .
}
```

Но в версии Java 7 были внесены два усовершенствования. Во-первых, можно воспользоваться методом `Objects.hashCode()`, безопасно обрабатывающим пустые значения. В частности, он возвращает нуль, если его аргумент имеет пустое значение `null`, а иначе — результат вызова метода `hashCode()` для заданного аргумента. Ниже приведен пример применения данного метода в коде.

```
public int hashCode()
{
    return 7 * Objects.hashCode(name)
        + 11 * new Double(salary).hashCode()
        + 13 * Objects.hashCode(hireDay);
}
```

И во-вторых, можно вызвать метод `Objects.hash()`, если требуется объединить несколько хеш-значений, что еще лучше. В этом случае метод `Objects.hashCode()` будет вызван для каждого аргумента с целью объединить получаемые в итоге хеш-значения. В таком случае метод `Employee.hashCode()` реализуется очень просто, как показано ниже.

```
public int hashCode()
{
    return Objects.hash(name, salary, hireDay);
}
```

Методы `equals()` и `hashCode()` должны быть совместимы: если при вызове `x.equals(y)` возвращается логическое значение `true`, то и результаты вызовов `x.hashCode()` и `y.hashCode()` также должны совпадать. Так, если в методе `Employee.equals()` сравниваются идентификационные номера сотрудников, то при вычислении хеш-кода методу `hashCode()` также потребуются идентификационные номера, но не имя сотрудника и не адрес памяти, занимаемой соответствующим объектом.



СОВЕТ. Если имеются поля типа массива, для вычисления хеш-кода, состоящего из хеш-кодов элементов массива, можно воспользоваться методом `Arrays.hashCode()`.

java.lang.Object 1.0

- `int hashCode()`
Возвращает хеш-код объекта. Хеш-код представляет собой положительное или отрицательное целое число. Для равнозначных объектов должны возвращаться одинаковые хеш-коды.

java.lang.Objects 7

- `int hash(Object... объекты)`
Возвращает хеш-код, состоящий из хеш-кодов всех предоставляемых объектов.
- `static int hashCode(Object a)`
Возвращает нуль, если параметр `a` имеет пустое значение `null`, а иначе — делает вызов `a.hashCode()`.

java.util.Arrays 1.2

- `static int hashCode(type[] a) 5.0`
Вычисляет хеш-код массива `a`, который может содержать компоненты типа `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`.

Метод toString()

Еще одним важным в классе `Object` является метод `toString()`, возвращающий значение объекта в виде символьной строки. В качестве примера можно привести метод `toString()` из класса `Point`, который возвращает символьную строку, подобную приведенной ниже.

```
java.awt.Point[x=10,y=20]
```

Большинство (но не все) методов `toString()` возвращают символьную строку, состоящую из имени класса, за которым следуют значения его полей в квадратных скобках. Ниже приведен пример реализации метода `toString()` в классе `Employee`.

```
public String toString()
{
    return "Employee[name=" + name
        + ",salary=" + salary
        + ",hireDay=" + hireDay
        + "];"
}
```

На самом деле этот метод можно усовершенствовать. Вместо жесткого кодирования имени класса в методе `toString()` достаточно вызвать метод `getClass().getName()` и получить символьную строку, содержащую имя класса, как показано ниже.

```
public String toString()
{
    return getClass().getName()
        + "[name=" + name
        + ",salary=" + salary
        + ",hireDay=" + hireDay
        + "];"
}
```

Такой метод `toString()` пригоден для подклассов. Разумеется, при создании подкласса следует определить собственный метод `toString()` и добавить поля подкласса. Так, если в суперклассе делается вызов `getClass().getName()`, то в подклассе просто вызывается метод `super.toString()`. Ниже приведен пример реализации метода `toString()` в классе `Manager`.

```
class Manager extends Employee
{
    . . .
    public String toString()
    {
        return super.toString()
            + "[bonus=" + bonus
            + "];"
    }
}
```

Теперь состояние объекта типа `Manager` выводится следующим образом:

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

Метод `toString()` универсален. Имеется следующее веское основание для его реализации в каждом классе: если объект объединяется с символьной строкой с помощью операции `+`, компилятор Java автоматически вызывает метод `toString()`, чтобы получить строковое представление этого объекта:

```
Point p = new Point(10, 20);
String message = "The current position is " + p;
// метод p.toString() вызывается автоматически
```



СОВЕТ. Вместо вызова `x.toString()` можно использовать выражение `"" + x`. Сцепление пустой символьной строки со строковым представлением объекта `x` равнозначно вызову метода `x.toString()`. Такое выражение будет корректным, даже если переменная относится к одному из примитивных типов.

Если `x` — произвольный объект и в программе имеется следующая строка кода:

```
System.out.println(x);
```

то из метода `println()` будет вызван метод `x.toString()` и выведена символьная строка результата. Метод `toString()`, определенный в классе `Object`, выводит имя класса и адрес объекта. Рассмотрим следующий вызов:

```
System.out.println(System.out);
```

После выполнения метода `println()` отображается такая строка:

```
java.io.PrintStream@2f6684
```

Как видите, разработчики класса `PrintStream` не позаботились о переопределении метода `toString()`.



ВНИМАНИЕ! Как это ни досадно, но массивы наследуют метод `toString()` от класса `Object`, в результате чего тип массива выводится в архаичном формате. Например, при выполнении приведенного ниже фрагмента кода получается символьная строка `"[I@1a46e30"`, где префикс `[I` означает массив целых чисел.

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
String s = "" + luckyNumbers;
```

В качестве выхода из этого неприятного положения может стать вызов статического метода `Arrays.toString()`. Так, при выполнении следующего кода будет получена символьная строка `"[2, 3, 5, 7, 11, 13]"`:

```
String s = Arrays.toString(luckyNumbers);
```

А для корректного вывода многомерных массивов следует вызывать метод `Arrays.deepToString()`.

Метод `toString()` отлично подходит для регистрации и протоколирования. Этот метод определен во многих классах из стандартной библиотеки, что позволяет получать полезные сведения о состоянии объекта. Так, например, для регистрации сообщений можно применить следующее выражение:

```
System.out.println("Current position = " + position);
```

Из главы 11 вы узнаете, что существует решение и получше: воспользоваться объектом класса `Logger` и сделать следующий вызов:

```
Logger.global.info("Current position = " + position);
```



СОВЕТ. Настоятельно рекомендуется переопределять метод `toString()` в каждом создаваемом вами классе. Это будет полезно как вам, так и тем, кто пользуется плодами ваших трудов.

В примере программы из листинга 5.8 демонстрируется применение методов `equals()`, `hashCode()` и `toString()`, реализованных в классах `Employee` (из листинга 5.9) и `Manager` (из листинга 5.10).

Листинг 5.8. Исходный код из файла `equals/EqualsTest.java`

```
1 package equals;
2
```

```
3 /**
4  * В этой программе демонстрируется применение метода equals()
5  * @version 1.12 2012-01-26
6  * @author Cay Horstmann
7  */
8 public class EqualsTest
9 {
10     public static void main(String[] args)
11     {
12         Employee alicel = new Employee("Alice Adams", 75000, 1987, 12, 15);
13         Employee alicel2 = alicel;
14         Employee alicel3 = new Employee("Alice Adams", 75000, 1987, 12, 15);
15         Employee bob = new Employee("Bob Brandson", 50000, 1989, 10, 1);
16
17         System.out.println("alicel == alicel2: " + (alicel == alicel2));
18
19         System.out.println("alicel == alicel3: " + (alicel == alicel3));
20
21         System.out.println("alicel.equals(alicel3): " + alicel.equals(alicel3));
22
23         System.out.println("alicel.equals(bob): " + alicel.equals(bob));
24
25         System.out.println("bob.toString(): " + bob);
26
27         Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
28         Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
29         boss.setBonus(5000);
30         System.out.println("boss.toString(): " + boss);
31         System.out.println("carl.equals(boss): " + carl.equals(boss));
32         System.out.println("alicel.hashCode(): " + alicel.hashCode());
33         System.out.println("alicel3.hashCode(): " + alicel3.hashCode());
34         System.out.println("bob.hashCode(): " + bob.hashCode());
35         System.out.println("carl.hashCode(): " + carl.hashCode());
36     }
37 }
```

Листинг 5.9. Исходный код из файла equals/Employee.java

```
1 package equals;
2
3 import java.util.Date;
4 import java.util.GregorianCalendar;
5 import java.util.Objects;
6
7 public class Employee
8 {
9     private String name;
10    private double salary;
11    private Date hireDay;
12
13    public Employee(String n, double s, int year, int month, int day)
14    {
15        name = n;
16        salary = s;
17        GregorianCalendar calendar = new GregorianCalendar(year, month-1, day);
18        hireDay = calendar.getTime();
```

```
19  }
20
21  public String getName()
22  {
23      return name;
24  }
25
26  public double getSalary()
27  {
28      return salary;
29  }
30
31  public Date getHireDay()
32  {
33      return hireDay;
34  }
35
36  public void raiseSalary(double byPercent)
37  {
38      double raise = salary * byPercent / 100;
39      salary += raise;
40  }
41
42  public boolean equals(Object otherObject)
43  {
44      // быстро проверить объекты на идентичность
45      if (this == otherObject) return true;
46
47      // если явный параметр имеет пустое значение null,
48      // должно быть возвращено логическое значение false
49      if (otherObject == null) return false;
50      // если классы не совпадают, они не равнозначны
51      if (getClass() != otherObject.getClass()) return false;
52
53      // теперь известно, что otherObject – непустой объект типа Employee
54      Employee other = (Employee) otherObject;
55
56      // проверить, содержат ли поля одинаковые значения
57      return Objects.equals(name, other.name) && salary == other.salary &&
58         Objects.equals(hireDay, other.hireDay);
59  }
60
61  public int hashCode()
62  {
63      return Objects.hash(name, salary, hireDay);
64  }
65
66  public String toString()
67  {
68      return getClass().getName() + "[name=" + name +
69         ",salary=" + salary + ",hireDay=" + hireDay + "];"
70  }
71 }
```

Листинг 5.10. Исходный код из файла equals/Manager.java

```
1 package equals;
2
3 public class Manager extends Employee
4 {
5     private double bonus;
6
7     public Manager(String n, double s, int year, int month, int day)
8     {
9         super(n, s, year, month, day);
10        bonus = 0;
11    }
12
13    public double getSalary()
14    {
15        double baseSalary = super.getSalary();
16        return baseSalary + bonus;
17    }
18
19    public void setBonus(double b)
20    {
21        bonus = b;
22    }
23    public boolean equals(Object otherObject)
24    {
25        if (!super.equals(otherObject)) return false;
26        Manager other = (Manager) otherObject;
27        // В методе super.equals() проверяется, принадлежат ли объекты,
28        // доступные по ссылкам this и other, одному и тому же классу
29        return bonus == other.bonus;
30    }
31
32    public int hashCode()
33    {
34        return super.hashCode() + 17 * new Double(bonus).hashCode();
35    }
36
37    public String toString()
38    {
39        return super.toString() + "[bonus=" + bonus + "]";
40    }
41 }
```

java.lang.Object 1.0

- Class getClass()
Возвращает класс объекта, содержащий сведения об объекте. Как будет показано далее в этой главе, в Java поддерживается динамическое представление классов, инкапсулированное в классе Class.

- `boolean equals(Object otherObject)`

Сравнивает два объекта и возвращает логическое значение `true`, если объекты занимают одну и ту же область памяти, а иначе — логическое значение `false`. Этот метод следует переопределить при создании собственных классов.

- `String toString()`

Возвращает символьную строку, представляющую значение объекта. Этот метод следует переопределить при создании собственных классов.

`java.lang.Class 1.0`

- `String getName()`

Возвращает имя класса.

- `Class getSuperclass()`

Возвращает имя суперкласса данного класса в виде объекта типа `Class`.

Обобщенные списочные массивы

Во многих языках программирования и, в частности, в `C`, размер всех массивов должен задаваться еще на стадии компиляции программы. Это ограничение существенно затрудняет работу программиста. Сколько сотрудников будет работать в отделе? Однозначно — не больше 100 человек. А что, если где-то есть отдел, насчитывающий 150 сотрудников, или же если отдел невелик и в нем работают только 10 человек? Ведь в этом случае 90% массива не используется!

В `Java` ситуация намного лучше — размер массива можно задавать уже во время выполнения программы, как показано ниже.

```
int actualSize = ...;
Employee[] staff = new Employee[actualSize];
```

Разумеется, этот код не решает проблему динамического видоизменения массивов во время выполнения программы. Задав размер массива, его затем нелегко изменить. Это затруднение проще всего разрешить, используя списочные массивы. Для их создания применяются экземпляры класса, который называется `ArrayList`. Этот класс действует подобно обычному массиву, но может динамически изменять его размеры по мере добавления новых элементов или удаления существующих. При этом программисту не приходится писать дополнительный код.

Класс `ArrayList` является обобщенным с параметром типа. Тип элемента массива заключается в угловые скобки и добавляется к имени класса: `ArrayList<Employee>`. Подробнее о создании собственных обобщенных классов речь пойдет в главе 13, но пользоваться объектами типа `ArrayList` можно, и не зная все эти технические подробности.

Ниже приведена строка кода, в которой создается списочный массив, предназначенный для хранения объектов типа `Employee`.

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

Указывать параметр типа в угловых скобках с обеих сторон выражения не очень удобно. Поэтому, начиная с версии Java 7, появилась возможность опускать параметр типа с правой стороны выражения:

```
ArrayList<Employee> staff = new ArrayList<>();
```

Это так называемый *ромбовидный* оператор, потому что угловые скобки, `<>`, напоминают ромб. Ромбовидный оператор следует использовать вместе с оператором `new`. Компилятор проверяет, что именно происходит с новым значением. Если оно присваивается переменной, передается методу или возвращается из метода, то компилятор сначала проверяет обобщенный тип переменной, параметра или метода, а затем заключает этот тип в угловые скобки. В данном примере новое значение `new ArrayList<>()` присваивается переменной типа `ArrayList<Employee>`. Следовательно, `Employee` становится обобщенным типом.



НА ЗАМЕТКУ! До версии Java SE 5.0 обобщенные классы отсутствовали. Вместо них применялся единственный класс `ArrayList`, который позволял хранить объекты типа `Object`. Если вы работаете со старыми версиями Java, не добавляйте к имени `ArrayList` суффикс в угловых скобках `<...>`. Вы можете и дальше пользоваться именем `ArrayList` без суффикса `<...>`. Класс `ArrayList` можно рассматривать как базовый, или так называемый “сырой” тип.



НА ЗАМЕТКУ! В еще более старых версиях языка Java для создания массивов, размеры которых динамически изменялись, программисты пользовались классом `Vector`. Но класс `ArrayList` эффективнее. С его появлением отпала необходимость пользоваться классом `Vector`.

Для добавления новых элементов в списочный массив служит метод `add()`. Ниже приведен фрагмент кода, используемый для заполнения такого массива объектами типа `Employee`.

```
staff.add(new Employee("Harry Hacker", . . .));  
staff.add(new Employee("Tony Tester", . . .));
```

Списочный массив управляет внутренним массивом ссылок на объекты. В конечном итоге элементы массива могут оказаться исчерпаны. И здесь на помощь приходят специальные средства списочного массива. Так, если метод `add()` вызывается при заполненном внутреннем массиве, из списочного массива автоматически создается массив большего размера, куда копируются все объекты.

Если заранее известно, сколько элементов следует хранить, то перед заполнением списочного массива нужно вызвать метод `ensureCapacity()` следующим образом:

```
staff.ensureCapacity(100);
```

При вызове этого метода выделяется память для внутреннего массива, состоящего из 100 объектов. Затем можно вызвать метод `add()`, который уже не будет испытывать затруднений при перераспределении памяти. Первоначальную емкость списочного массива можно передать конструктору класса `ArrayList` в качестве параметра:

```
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```



ВНИМАНИЕ! Выделение памяти для списочного и обычного массивов происходит по-разному. Так, приведенные ниже выражения не равнозначны.

```
new ArrayList<Employee>(100) // емкость списочного массива равна 100  
new Employee[100] // размер обычного массива равен 100
```

Между емкостью списочного массива и размером обычного массива имеется существенное отличие. Выделяя память для массива из 100 элементов, вы резервируете их для дальнейшего использования. В то же время емкость списочного массива — это всего лишь возможная величина. В ходе работы она может быть увеличена (за счет выделения дополнительной памяти), но сразу после создания списочный массив не содержит ни одного элемента.

Метод `size()` возвращает фактическое количество элементов в списочном массиве. Например, в результате приведенного ниже вызова возвращается текущее количество элементов в массиве `staff`.

```
staff.size()
```

Равнозначное выражение для определения размера обычного массива `a` выглядит так:

```
a.length
```

Если вы уверены, что списочный массив будет иметь постоянный размер, то можете вызвать метод `trimToSize()`. Этот метод устанавливает размер блока памяти таким образом, чтобы он точно соответствовал количеству хранимых элементов. Система “сборки мусора” предотвращает неэффективное использование памяти, освобождая ее излишки.

Если после усечения размера списочного массива методом `trimToSize()` добавить в этот массив новые элементы, блок памяти будет перемещен, на что потребуется дополнительное время. Поэтому вызывать данный метод следует лишь в том случае, когда точно известно, что дополнительные элементы в списочный массив вводиться не будут.



НА ЗАМЕТКУ C++! Класс `ArrayList` ведет себя так же, как и шаблон `vector` в C++. В частности, класс `ArrayList` и шаблон `vector` относятся к универсальным типам. Но в шаблоне `vector` происходит перегрузка операции `[]`, что упрощает доступ к элементам массива. В Java перегрузка операций не предусмотрена, поэтому методы следует вызывать явным образом. Кроме того, шаблон `vector` в C++ передается по значению. Если `a` и `b` — два вектора, то выражение `a = b` приведет к созданию нового вектора, длина которого равна `b`. Все элементы вектора `b` будут скопированы в вектор `a`. В результате выполнения того же самого выражения в Java переменные `a` и `b` будут ссылаться на один и тот же списочный массив.

`java.util.ArrayList<T>` 1.2

- `ArrayList<T>()`
Конструирует пустой списочный массив.
- `ArrayList<T>(int initialCapacity)`
Конструирует пустой списочный массив заданной емкости.
Параметры: `initialCapacity` Первоначальная емкость списочного массива

- `boolean add(T obj)`

Добавляет элемент в конец массива. Всегда возвращает логическое значение `true`.

Параметры: `obj` Добавляемый элемент

- `int size()`

Возвращает количество элементов, хранящихся в списочном массиве. (Количество элементов отличается от емкости массива и не превосходит ее.)

- `void ensureCapacity(int capacity)`

Обеспечивает емкость списочного массива, достаточную для хранения заданного количества элементов без изменения внутреннего массива, предназначенного для хранения данных в памяти.

Параметры: `capacity` Требуемая емкость списочного массива

- `void trimToSize()`

Сокращает емкость списочного массива до его текущего размера.

Доступ к элементам списочных массивов

К сожалению, ничто не дается бесплатно. За удобство, предоставляемое автоматическим регулированием размера списочного массива, приходится расплачиваться более сложным синтаксисом, который требуется для доступа к его элементам. Дело в том, что класс `ArrayList` не входит в состав Java, а является лишь служебным классом, специально введенным в стандартную библиотеку.

Вместо удобных квадратных скобок для доступа к элементам списочного массива приходится вызывать методы `get()` и `set()`. Например, для установки i -го элемента списочного массива служит следующее выражение:

```
staff.set(i, harry);
```

Это равнозначно приведенному ниже выражению для установки i -го элемента обычного массива `a`. Как в обычных, так и в списочных массивах индексы отсчитываются от нуля.

```
a[i] = harry;
```



ВНИМАНИЕ! Не вызывайте метод `list.set(i, x)` до тех пор, пока размер списочного массива больше i . Например, следующий код написан неверно:

```
ArrayList<Employee> list = new ArrayList<Employee>(100); //емкость 100, размер 0  
list.set(0, x); // элемента 0 в списочном массиве пока еще нет
```

Для заполнения списочного массива вызывайте метод `add()` вместо метода `set()`, а последний применяйте только для замены ранее введенного элемента.

Получить элемент списочного массива можно с помощью метода `get()`, как показано ниже.

```
Employee e = staff.get(i);
```

Это равнозначно следующему выражению для массива `a`:

```
Employee e = a[i];
```



НА ЗАМЕТКУ! Когда обобщенные классы отсутствовали, единственным вариантом возврата значения из метода `get()` базового типа `ArrayList` была ссылка на объект класса `Object`. Очевидно, что в вызывающем методе приходилось выполнять приведение типов следующим образом:

```
Employee e = (Employee) staff.get(i);
```

При использовании класса базового типа `ArrayList` возможны неприятные ситуации, связанные с тем, что его методы `add()` и `set()` допускают передачу параметра любого типа. Так, приведенный ниже вызов воспринимается компилятором как правильный.

```
staff.set(i, new Date());
```

Серьезные осложнения могут возникнуть лишь после того, когда вы извлечете объект и попытаетесь привести его к типу `Employee`. А если вы воспользуетесь обобщением `ArrayList<Employee>`, то ошибка будет выявлена уже на стадии компиляции.

Иногда гибкость и удобство доступа к элементам удается сочетать, пользуясь следующим приемом. Сначала создайте список массивов и добавьте в него все нужные элементы:

```
ArrayList<X> list = new ArrayList<X>();  
while (...)  
{  
    x = ...;  
    list.add(x);  
}
```

Затем, используя метод `toArray()`, скопируйте все элементы в массив следующим образом:

```
X[] a = new X[list.size()];  
list.toArray(a);
```

Элементы можно добавлять не только в конец списочного массива, но и в его середину:

```
int n = staff.size() / 2;  
staff.add(n, e);
```

Элемент по индексу `n` и следующие за ним элементы сдвигаются, чтобы освободить место для нового элемента. Если после вставки элемента новый размер списочного массива превышает его емкость, происходит копирование массива.

Аналогично можно удалить элемент из середины списочного массива следующим образом:

```
Employee e = staff.remove(n);
```

Элементы, следующие после удаленного элемента, сдвигаются влево, а размер списочного массива уменьшается на единицу. Вставка и удаление элементов списочного массива не особенно эффективна. Для массивов, размеры которых невелики, это не имеет особого значения. Но если при обработке больших объемов данных приходится часто вставлять и удалять элементы, попробуйте вместо списочного массива воспользоваться связным списком. Особенности программирования связных списков рассматриваются в главе 13.

Для перебора содержимого списочного массива можно также организовать цикл в стиле `for each` следующим образом:

```
for (Employee e : staff)
    сделать что-нибудь с переменной e
```

Выполнение этого цикла дает такой же результат, как и приведенного ниже цикла:

```
for (int i = 0; i < staff.size(); i++)
{
    Employee e = staff.get(i);
    сделать что-нибудь с переменной e
}
```

В листинге 5.11 приведен исходный код видоизмененной версии программы `EmployeeTest` из главы 4. Обычный массив `Employee[]` в ней заменен обобщенным списочным массивом `ArrayList<Employee>`. Обратите внимание на следующие особенности данной версии программы.

- Не нужно задавать размер массива.
- С помощью метода `add()` можно добавлять сколько угодно элементов.
- Вместо свойства `length` для подсчета количества элементов служит метод `size()`.
- Вместо выражения `a[i]` для доступа к элементу массива вызывается метод `a.get(i)`.

Листинг 5.11. Исходный код из файла `arrayList/ArrayListTest.java`

```
1 package arrayList;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируется применение класса ArrayList
7  * @version 1.11 2012-01-26
8  * @author Cay Horstmann
9  */
10 public class ArrayListTest
11 {
12     public static void main(String[] args)
13     {
14         // заполнить списочный массив staff тремя объектами типа Employee
15         ArrayList<Employee> staff = new ArrayList<>();
16         staff.add(new Employee("Carl Cracker", 75000, 1987, 12, 15));
17         staff.add(new Employee("Harry Hacker", 50000, 1989, 10, 1));
18         staff.add(new Employee("Tony Tester", 40000, 1990, 3, 15));
19
20         // поднять всем сотрудникам зарплату на 5%
21         for (Employee e : staff)
22             e.raiseSalary(5);
23
24         // вывести данные обо всех объектах типа Employee
25         for (Employee e : staff)
26             System.out.println("name=" + e.getName() + ", salary=" +
27                 e.getSalary() + ", hireDay=" + e.getHireDay());
28     }
29 }
```

которых отличаются от `Employee`. Это настораживает, но если хорошенько подумать, то именно такое поведение было характерно для кода до внедрения обобщений в Java. И хотя целостность виртуальной машины Java при этом не нарушается, а безопасность обеспечивается, в то же время теряются преимущества контроля типов на стадии компиляции.

С другой стороны, если попытаться присвоить списочный массив базового типа `ArrayList` типизированному массиву, как показано ниже, то компилятор выдаст соответствующее предупреждение.

```
ArrayList<Employee> result = employeeDB.find(query);  
    // выдается предупреждение
```



НА ЗАМЕТКУ! Чтобы увидеть текст предупреждающего сообщения, при вызове компилятора следует указать параметр `-Xlint:unchecked`.

Попытка выполнить приведение типов не исправит ситуацию, как показано ниже. Изменится лишь само предупреждающее сообщение. На этот раз оно уведомит о неверном приведении типов.

```
ArrayList<Employee> result = (ArrayList<Employee>) employeeDB.find(query);  
    // на этот раз появится другое предупреждение
```

Это происходит из-за некоторых неудачных ограничений, накладываемых на обобщенные типы в Java. Ради совместимости компилятор преобразует типизированные списочные массивы в объекты базового типа `ArrayList` после того, как будет проверено, соблюдены ли правила контроля типов. В процессе выполнения программы все списочные массивы одинаковы: виртуальная машина не получает никаких данных о параметрах типа. Поэтому приведение типов (`ArrayList`) и (`ArrayList<Employee>`) проходит одинаковую проверку во время работы программы.

В подобных случаях от вас мало что зависит. При видоизменении унаследованного кода вам остается только следить за сообщениями компилятора, довольствуясь тем, что они не уведомляют о серьезных ошибках.

Удовлетворившись результатами компиляции унаследованного кода, вы можете пометить переменную, принимающую результат приведения типов, аннотацией `@SuppressWarnings("unchecked")`, как показано ниже.

```
@SuppressWarnings("unchecked") ArrayList<Employee> result =  
    (ArrayList<Employee>) employeeDB.find(query);  
    // выдается еще одно предупреждение
```

Объектные оболочки и автоупаковка

Иногда переменные примитивных типов вроде `int` приходится преобразовывать в объекты. У всех примитивных типов имеются аналоги в виде классов. Например, существует класс `Integer`, соответствующий типу `int`. Такие классы принято называть *объектными оболочками*. Они имеют вполне очевидные имена: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, `Void` и `Boolean`. (У первых шести классов имеется общий суперкласс `Number`.) Классы объектных оболочек являются неизменяемыми. Это означает, что изменить значение, хранящееся в объектной оболочке после ее создания, нельзя.

Допустим, в списочном массиве требуется хранить целые числа. К сожалению, с помощью параметра типа в угловых скобках нельзя задать примитивный тип, например, выражение `ArrayList<int>` недопустимо. И здесь приходит на помощь класс объектной оболочки `Integer`. В частности, списочный массив, предназначенный для хранения объектов типа `Integer`, можно объявить следующим образом:

```
ArrayList<Integer> list = new ArrayList<Integer>();
```



ВНИМАНИЕ! Применение объектной оболочки типа `ArrayList<Integer>` оказывается менее эффективным, чем массив `int[]`. Причина очевидна: каждое целочисленное значение инкапсулировано внутри объекта, и для его записи или извлечения необходимо предпринимать дополнительные действия. Таким образом, применение объектных оболочек оправдано лишь для больших коллекций, когда удобство работы программиста важнее эффективности работы самой программы.

Начиная с версии Java SE 5.0 стало проще добавлять и извлекать элементы из массива. Рассмотрим следующую строку кода:

```
list.add(3);
```

Она автоматически преобразуется в приведенную ниже строку кода. Подобное автоматическое преобразование называется *автоупаковкой*.

```
list.add(new Integer(3));
```



НА ЗАМЕТКУ! Для такого преобразования больше подходит обозначение *автоматическое заключение в оболочку*, но понятие *упаковки* было заимствовано из C#.

С другой стороны, если присвоить объект типа `Integer` переменной типа `int`, целочисленное значение будет автоматически извлечено из объекта, т.е. *распаковано*. Иными словами, компилятор преобразует следующую строку кода:

```
int n = list.get(i);
```

в приведенную ниже строку кода.

```
int n = list.get(i).intValue();
```

Автоматическая упаковка и распаковка примитивных типов может выполняться и при вычислении арифметических выражений. Например, операцию инкремента можно применить к переменной, содержащей ссылку на объект типа `Integer`, как показано ниже.

```
Integer n = 3;  
n++;
```

Компилятор автоматически распакует целочисленное значение из объекта, увеличит его на единицу и снова упакует в объект.

На первый взгляд может показаться, что примитивные типы и их объектные оболочки — одно и то же. Их отличие становится очевидным при выполнении операции проверки на равенство. Как вам должно быть уже известно, при выполнении операции `==` над объектом проверяется, ссылаются ли переменные на один и тот же адрес памяти, где находится объект. Ниже приведен пример, в котором, несмотря на равенство целочисленных значений, проверка на равенство, вероятнее всего, даст отрицательный результат.

```
Integer a = 1000;
Integer b = 1000;
if (a == b) . . .
```

Но реализация Java *может*, если пожелает, заключить часто встречающиеся значения в оболочки одинаковых объектов, и тогда сравнение даст положительный результат. Хотя такая неоднозначность результатов мало кому нужна. В качестве выхода из этого положения можно воспользоваться методом `equals()` при сравнении объектов-оболочек.



НА ЗАМЕТКУ! Спецификация автоупаковки требует, чтобы значения типа `boolean`, `byte`, `char` меньше 127, а также `short` и `int` в пределах от -128 до 127 упаковывались в фиксированные объекты. Так, если переменные `a` и `b` из предыдущего примера инициализировать значением 100, их сравнение должно дать положительный результат.

И наконец, следует заметить, что за упаковку и распаковку отвечает не виртуальная машина, а компилятор. Он включает в программу необходимые вызовы, а виртуальная машина лишь выполняет байт-код.

Объектные оболочки числовых значений находят широкое распространение еще и по другой причине. Создатели Java решили, что в составе классов объектных оболочек удобно было бы реализовать методы для преобразования символьных строк в числа. Чтобы преобразовать символьную строку в целое число, можно воспользоваться выражением, подобным приведенному ниже.

```
int x = Integer.parseInt(s);
```

Обратите внимание на то, что создавать объект типа `Integer` в этом случае совсем не обязательно, так как метод `parseInt()` является статическим. И тем не менее класс `Integer` — подходящее для этого место.

Ниже описаны наиболее употребительные методы из класса `Integer`. Аналогичные методы имеются и в других классах объектных оболочек для значений примитивных типов.



ВНИМАНИЕ! Некоторые считают, что с помощью классов объектных оболочек можно реализовать методы, видоизменяющие свои числовые параметры. Но это неверно. Как пояснялось в главе 4, на Java нельзя написать метод, увеличивающий целое число, передаваемое ему в качестве параметра, поскольку все параметры в этом языке передаются *только по значению*.

```
public static void triple(int x) // не работает!
{
    x++; // видоизменить локальную переменную
}
```

Но, может быть, это ограничение можно обойти, используя вместо типа `int` класс `Integer`?

```
public static void triple(Integer x) // все равно не работает!
{
    ...
}
```

Дело в том, что объект типа `Integer` не позволяет изменять содержащиеся в нем данные. Следовательно, изменять числовые параметры, передаваемые методам, с помощью классов объектных оболочек нельзя.

Если все-таки требуется создать метод, изменяющий свои числовые параметры, для этого можно воспользоваться одним из *контейнерных* типов, определенных в пакете `org.omg.CORBA`. К таким типам относятся `IntHolder`, `BooleanHolder` и др. Каждый такой тип содержит открытое (!) поле `value`, через которое можно обращаться к хранящемуся в нем числу, как показано ниже.

```
public static void triple(IntHolder x)
{
    x.value++;
}
```

java.lang.Integer 1.0

- `int intValue()`

Возвращает значение из данного объекта типа `Integer` в виде числового значения типа `int` (этот метод переопределяет метод `intValue()` из класса `Number`).

- `static String toString(int i)`

Возвращает новый объект типа `String`, представляющий числовое значение в десятичной форме.

- `static String toString(int i, int radix)`

Возвращает новый объект типа `String`, представляющий число в системе счисления, определяемой параметром `radix`.

- `static int parseInt(String s)`

- `static int parseInt(String s, int radix)`

Возвращают целое значение. Предполагается, что объект типа `String` содержит символьную строку, представляющую целое число в десятичной системе счисления (в первом варианте метода) или же в системе счисления, которая задается параметром `radix` (во втором варианте метода).

- `static Integer valueOf(String s)`

- `static Integer valueOf(String s, int radix)`

Возвращают новый объект типа `Integer`, инициализированный целым значением, которое задается с помощью первого параметра. Предполагается, что объект типа `String` содержит символьную строку, представляющую целое число в десятичной системе счисления (в первом варианте метода) или же в системе счисления, которая задается параметром `radix` (во втором варианте метода).

java.text.NumberFormat 1.1

- `Number parse(String s)`

Возвращает числовое значение, полученное в результате синтаксического анализа параметра. Предполагается, что объект типа `String` содержит символьную строку, представляющую числовое значение.

Методы с переменным числом параметров

До версии Java SE 5.0 количество параметров любого метода, написанного на Java, было фиксировано. Теперь же есть возможность создавать методы, позволяющие при разных вызовах задавать различное количество параметров. С одним из таких методов, `printf()`, вы уже знакомы. Ниже представлены два примера обращения к нему.

```
System.out.printf("%d", n);
System.out.printf("%d %s", n, "widgets");
```

В обоих приведенных выше строках кода вызывается один и тот же метод, но в первом случае этому методу передаются два параметра, а во втором — три параметра. Метод `printf()` определяется в общей форме следующим образом:

```
public class PrintStream
{
    public PrintStream printf(String fmt, Object... args)
        { return format(fmt, args); }
}
```

Здесь многоточие (...) обозначается часть кода Java. Оно указывает на то, что в дополнение к параметру `fmt` можно указывать любое количество объектов. По существу, метод `printf()` получает два параметра: форматировавшую строку и массив типа `Object[]`, в котором хранятся все остальные параметры. (Если этому методу передаются целочисленные значения или же значения одного из примитивных типов, то они преобразуются в объекты путем автоупаковки.) После этого метод решает непростую задачу разбора форматировавшей строки `fmt` и связывания спецификаторов формата со значениями параметров `args[i]`. Иными словами, в методе `printf()` тип параметра `Object...` означает то же самое, что и `Object[]`.

Компилятор при обработке исходного кода выявляет каждое обращение к методу `printf()`, размещает параметры в массиве и, если требуется, выполняет автоупаковку:

```
System.out.printf("%d %s", new Object[] { new Integer(n), "widgets" } );
```

В случае необходимости можно создать собственные методы с переменным числом параметров, указав любые типы параметров, в том числе и примитивные типы. Ниже приведен пример простого метода, в котором вычисляется и возвращается максимальное из произвольного количества значений.

```
public static double max(double... values)
{
    double largest = Double.MIN_VALUE;
    for (double v : values) if (v > largest) largest = v;
    return largest;
}
```

Вызов этого метода может выглядеть следующим образом:

```
double m = max(3.1, 40.4, -5);
```

В этом случае компилятор передает методу `max()` параметры в виде следующего выражения: `new double[] { 3.1, 40.4, -5 }`.



НА ЗАМЕТКУ! В качестве последнего параметра метода, число параметров которого может быть переменным, допускается задавать массив следующим образом:

```
System.out.printf("%d %s", new Object[] { new Integer(1), "widgets" } );
```

Таким образом, имеющийся метод, последний параметр которого является массивом, можно переопределить как метод с переменным числом параметров, не изменяя существующего кода. Подобным образом в Java SE 5.0 был расширен метод `MessageFormat.format()`. По желанию метод `main()` можно даже объявить следующим образом:

```
public static void main(String... args)
```

Классы перечислений

Как упоминалось в главе 3, начиная с версии Java SE 5.0 в Java поддерживаются перечислимые типы. Ниже приведен характерный пример применения перечислимых типов в коде.

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

Тип, объявленный подобным образом, на самом деле представляет собой класс. Допускается существование всего четырех экземпляров указанного класса перечисления. Другие объекты в таком классе создать нельзя.

Таким образом, для проверки перечислимых значений на равнозначность совсем не обязательно использовать метод `equals()`. Для этой цели вполне подойдет операция `==`. По желанию к перечислимым типам можно добавить конструкторы, методы и поля. Очевидно, что конструкторы могут вызываться только при создании перечислимых констант. Ниже приведен соответствующий тому пример.

```
public enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private String abbreviation;

    private Size(String abbreviation) { this.abbreviation = abbreviation; }
    public String getAbbreviation() { return abbreviation; }
}
```

Все перечислимые типы реализуются с помощью подклассов, производных от класса `Enum`. Они наследуют от этого класса ряд методов. Наиболее часто применяется метод `toString()`, возвращающий имя перечислимой константы. Например, при вызове метода `Size.SMALL.toString()` возвращается символьная строка "SMALL". Статический метод `valueOf()` выполняет действия, противоположные методу `toString()`. Например, в результате выполнения приведенного ниже выражения переменной `s` будет присвоено значение `Size.SMALL`.

```
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

Каждый перечислимый тип содержит статический метод `values()`, который возвращает массив всех перечислимых значений:

```
Size[] values = Size.values();
```

Метод `ordinal()` возвращает позицию перечислимой константы в объявлении `enum`, начиная с нуля. Например, при вызове `Size.MEDIUM.ordinal()` возвратится значение 1. Обращение с перечислимыми типами демонстрирует короткая программа, приведенная в листинге 5.12.



НА ЗАМЕТКУ! У класса Enum имеется также параметр типа, который был ранее опущен ради простоты. Например, перечислимый тип Size на самом деле преобразуется в Enum<Size>, но в этом разделе такая возможность не рассматривалась. В частности, параметр типа используется в методе compareTo(). (Метод compareTo() будет обсуждаться в главе 6, а параметры типа — в главе 12.)

Листинг 5.12. Исходный код из файла enums/EnumTest.java

```
1 package enums;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируются перечислимые типы
7  * @version 1.0 2004-05-24
8  * @author Cay Horstmann
9  */
10 public class EnumTest
11 {
12     public static void main(String[] args)
13     {
14         Scanner in = new Scanner(System.in);
15         System.out.print("Enter a size: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
16         String input = in.next().toUpperCase();
17         Size size = Enum.valueOf(Size.class, input);
18         System.out.println("size=" + size);
19         System.out.println("abbreviation=" + size.getAbbreviation());
20         if (size == Size.EXTRA_LARGE)
21             System.out.println("Good job--you paid attention to the _.");
22     }
23 }
24
25 enum Size
26 {
27     SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
28
29     private Size(String abbreviation) { this.abbreviation = abbreviation; }
30     public String getAbbreviation() { return abbreviation; }
31
32     private String abbreviation;
33 }
```

java.lang.Enum<E> 5.0

- static Enum valueOf(Class enumClass, String name)
Возвращает перечислимую константу указанного класса с заданным именем.
- String toString()
Возвращает имя перечислимой константы.

- `int ordinal()`
Возвращает позицию данной перечислимой константы в объявлении `enum`, начиная с нуля.
- `int compareTo(E other)`
Возвращает отрицательное целое значение, если перечислимая константа следует перед параметром `other`, 0 — если `this==other`, а иначе — положительное целое значение. Порядок следования констант задается объявлением `enum`.

Рефлексия

Библиотека рефлексии предоставляет богатый набор инструментальных средств для манипулирования кодом Java в динамическом режиме. Эта возможность широко используется в JavaBeans при создании компонентов. (Средства JavaBeans будут рассматриваться во втором томе данной книги.) Благодаря рефлексии появляется возможность поддерживать инструментальные средства, подобные тем, которыми пользуются программирующие на Visual Basic. Так, если в процессе разработки или выполнения программы добавляется новый класс, можно организовать опрос с целью выяснить возможности нового класса.

Программа, которая способна анализировать возможности классов, называется *рефлексивной*. Рефлексия — очень мощный механизм, который можно применять для решения перечисленных ниже задач. А в последующих разделах поясняется, как пользоваться этим механизмом.

- Анализ возможностей классов в процессе выполнения программы.
- Проверка объектов при выполнении программы, например, с помощью рефлексии можно реализовать метод `toString()`, совместимый со всеми классами.
- Реализация обобщенного кода для работы с массивами.
- Применение объектов типа `Method`, которые работают аналогично указателям на функции в языках, подобных C++.

Рефлексия — не только эффективный, но и сложный механизм. Ею интересуются в основном разработчики инструментальных средств, тогда как программисты, пишущие обычные прикладные программы, зачастую ею не пользуются. Если вы занимаетесь только написанием прикладных программ и еще не готовы к разработке инструментальных средств, можете пропустить оставшуюся часть этой главы, а в дальнейшем, если потребуется, вернуться к ее заключительным разделам.

Класс Class

Во время выполнения программы исполняющая система Java всегда осуществляет *динамическую идентификацию типов* объектов любых классов. Получаемые в итоге сведения используются виртуальной машиной для выбора подходящего вызываемого метода.

Но получить доступ к этой информации можно иначе, используя специальный класс, который так и называется: `Class`. Метод `getClass()` из класса `Object` возвращает экземпляр типа `Class`, как показано ниже.

```
Employee e;  
.  
.  
.  
Class cl = e.getClass();
```

Подобно тому, как объект типа `Employee` описывает свойства конкретного сотрудника, объект типа `Class` описывает свойства конкретного класса. Вероятно, наиболее употребительным в классе `Class` является метод `getName()`, возвращающий имя класса. Например, при выполнении следующей строки кода:

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

выводится символьная строка

```
Employee Harry Hacker
```

если объект `e` относится к классу `Employee`, или же символьная строка

```
Manager Harry Hacker
```

если объект `e` относится к классу `Manager`.

Если же класс находится в пакете, то имя пакета включается в имя класса следующим образом:

```
Date d = new Date();  
Class cl = d.getClass();  
String name = cl.getName(); // в переменной name устанавливается  
                           // строковое значение "java.util.Date"
```

Вызывая статический метод `forName()`, можно также получить объект типа `Class`, соответствующий имени класса в строковом представлении, как показано ниже.

```
String className = "java.util.Date";  
Class cl = Class.forName(className);
```

Этот метод можно применять, если имя класса хранится в символьной строке, содержимое которой во время выполнения программы изменяется. Он действует правильно, если переменная `className` содержит имя класса или интерфейса. В противном случае метод `forName()` *генерирует проверяемое исключение*. Порядок вызова обработчика исключений при обращении с этим методом описан ниже, в разделе "Основы обработки исключений".



СОВЕТ. При запуске программы на выполнение сначала загружается класс, содержащий метод `main()`. Он загружает все необходимые классы. Каждый из классов, в свою очередь, загружает необходимые ему классы и т.д. Если приложение достаточно крупное, этот процесс может отнять немало времени, и пользователю придется ожидать окончания загрузки. Для того чтобы вызвать у пользователя иллюзию быстрого запуска, можно воспользоваться следующим приемом. Убедитесь в том, что класс, содержащий метод `main()`, не обращается явно к другим классам. Образите в этом классе начальный экран приложения. Затем приступайте к загрузке остальных классов, вызывая метод `Class.forName()`.

Третий способ получения объекта типа `Class` прост и удобен. Если `T` — это некоторый тип, то `T.class` — объект соответствующего класса. Например:

```
Class cl1 = Date.class; // если импортирован пакет java.util.*;  
Class cl2 = int.class;  
Class cl3 = Double[].class;
```

Следует иметь в виду, что объект типа `Class` фактически описывает *тип*, который не обязательно является классом. Например, тип `int` — это не класс, но, несмотря на это, `int.class` — это объект типа `Class`.



НА ЗАМЕТКУ! Начиная с версии Java SE 5.0, класс `Class` является параметризованным. Например, ссылка `Employee.class` соответствует типу `Class<Employee>`. Мы не будем пока что обсуждать этот вопрос, чтобы не усложнять и без того довольно абстрактные понятия. На практике вы можете вообще игнорировать параметр типа и пользоваться обычным вариантом класса `Class`. Более подробно данный вопрос обсуждается в главе 13.



ВНИМАНИЕ! Исторически сложилось так, что метод `getName()` возвращает для массивов довольно странные имена.

- При вызове `Double[].class.getName()` возвращается символьная строка `"[Ljava.lang.Double;"`.
- При вызове `int[].class.getName()` возвращается символьная строка `"[I"`.

Виртуальная машина Java поддерживает однозначный объект типа `Class` для каждого типа. Следовательно, для сравнения объектов можно использовать операцию `==`, как показано ниже.

```
if (e.getClass() == Employee.class) . . .
```

Еще один полезный метод позволяет создавать при выполнении программы новый экземпляр класса. Имя метода, как и следовало ожидать, — `newInstance()`. Этот метод вызывается следующим образом:

```
e.getClass().newInstance();
```

В результате выполнения приведенной выше строки кода создается новый экземпляр того же класса, что и `e`. Для инициализации вновь созданного объекта в методе `newInstance()` используется конструктор без аргументов. Если в классе отсутствует конструктор без аргументов, генерируется исключение.

Используя методы `forName()` и `newInstance()`, можно создавать экземпляры классов, имена которых хранятся в символьных строках:

```
String s = "java.util.Date";
Object m = Class.forName(s).newInstance();
```



НА ЗАМЕТКУ! Если при создании объекта по имени класса требуется передать конструктору какие-нибудь параметры, то использовать приведенные выше операторы нельзя. Вместо этого следует вызвать метод `newInstance()` из класса `Constructor`.



НА ЗАМЕТКУ C++! Метод `newInstance()` является аналогом виртуального конструктора в C++. И хотя виртуальные конструкторы отсутствуют в C++ как языковое средство, такое понятие реализуется с помощью специализированных библиотек. Класс `Class` соответствует классу `typeid` в C++, а метод `getClass()` — операции `typeid`. Класс `Class` в Java более универсален, чем его аналог в C++. Класс `typeid` позволяет только извлекать символьную строку с именем типа, но не способен создавать объекты данного типа.

Основы обработки исключений

Обработка исключений подробно рассматривается в главе 11, но в предшествующих ей главах иногда будут встречаться методы, при выполнении которых могут возникать исключения. Если во время выполнения программы возникает ошибка, программа генерирует исключение. Это более гибкий процесс, чем простое прекращение выполнения программы, поскольку программист может создавать обработчики исключений, которые перехватывают исключения и каким-то образом обрабатывают их.

Если обработчик не предусмотрен, система преждевременно завершает выполнение программы и выводит на экран сообщение о типе исключения. Возможно, вы уже сталкивались с подобными сообщениями об исключениях. Так, если вы пытались использовать пустую ссылку или обращались за границы массива, то возникала исключительная ситуация и появлялось соответствующее сообщение.

Существуют две разновидности исключений: *непроверяемые* и *проверяемые*. При возникновении проверяемого исключения компилятор проверяет, предусмотрен ли для него обработчик. Но большинство исключений являются непроверяемыми. К ним относится, например, исключительная ситуация, возникающая при обращении по пустой ссылке. Компилятор не проверяет, предусмотрен ли в программе обработчик исключений. Вообще говоря, при написании программ следует стараться избегать подобных ошибок, а не предусматривать их обработку ради перестраховки.

Но не всех ошибок можно избежать. Если, несмотря на все ваши усилия, остается фрагмент кода, при выполнении которого может быть сгенерировано исключение, и вы не предусмотрели его обработку, компилятор будет настаивать на том, чтобы вы предоставили соответствующий обработчик. Например, метод `Class.forName()` может сгенерировать проверяемое исключение. В главе 11 мы рассмотрим ряд методик обработки исключений, а до тех пор покажем, каким образом реализуются простейшие обработчики исключений.

Итак, заключим один или несколько операторов, при выполнении которых могут возникнуть проверяемые исключения, в блок `try`, а код обработчика — в блок `catch` следующим образом:

```
try
{
    Операторы, способные генерировать исключения
}
catch (Exception e)
{
    Код обработчика исключений
}
```

Рассмотрим следующий пример кода:

```
try
{
    String name = . . . ; // получить имя класса
    Class cl = Class.forName(name); // может сгенерировать исключение
    сделать что-нибудь с переменной cl
}
catch (Exception e)
{
    e.printStackTrace();
}
```


три класса содержат метод `getModifiers()`, возвращающий целое значение, которое соответствует используемым модификаторам доступа, например `public` или `static`. Для анализа этого числа применяются статические методы класса `Modifiers` из пакета `java.lang.reflect`. В этом классе имеются, в частности, методы `isPublic()`, `isPrivate()` и `isFinal()`, определяющие, является ли анализируемый метод или конструктор открытым, закрытым или конечным. Все, что нужно для этого сделать, — применить соответствующий метод из класса `Modifier` к целому значению, которое возвращается методом `getModifiers()`. Для вывода самих модификаторов служит метод `Modifier.toString()`.

Методы `getFields()`, `getMethods()` и `getConstructors()` из класса `Class` возвращают массивы открытых полей, методов и конструкторов, принадлежащих анализируемому классу. К ним относятся и открытые поля суперклассов. Методы `getDeclaredFields()`, `getDeclaredMethods()` и `getDeclaredConstructors()` из класса `Class` возвращают массивы, состоящие из всех полей, методов и конструкторов, объявленных в классе. К ним относятся закрытые и защищенные компоненты, но не компоненты суперклассов.

В примере программы из листинга 5.13 показано, как отобразить все сведения о классе. Эта программа предлагает пользователю ввести имя класса, а затем выводит сигнатуры всех методов и конструкторов вместе с именами всех полей класса. Допустим, пользователь ввел в режиме командной строки следующее:

```
java.lang.Double
```

В результате выполнения данной программы на экране появится следующий текст:

```
public class java.lang.Double extends java.lang.Number
{
    public java.lang.Double(java.lang.String);
    public java.lang.Double(double);

    public int hashCode();
    public int compareTo(java.lang.Object);
    public int compareTo(java.lang.Double);
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public static java.lang.String toString(double);
    public static java.lang.Double valueOf(java.lang.String);
    public static boolean isNaN(double);
    public boolean isNaN();
    public static boolean isInfinite(double);
    public boolean isInfinite();
    public byte byteValue();
    public short shortValue();
    public int intValue();
    public long longValue();
    public float floatValue();
    public double doubleValue();
    public static double parseDouble(java.lang.String);
    public static native long doubleToLongBits(double);
    public static native long doubleToRawLongBits(double);
    public static native double longBitsToDouble(long);

    public static final double POSITIVE_INFINITY;
```

```

public static final double NEGATIVE_INFINITY;
public static final double NaN;
public static final double MAX_VALUE;
public static final double MIN_VALUE;
public static final java.lang.Class TYPE;
private double value;
private static final long serialVersionUID;
}

```

Листинг 5.13. Исходный код из файла reflection/ReflectionTest.java

```

1  package reflection;
2
3  import java.util.*;
4  import java.lang.reflect.*;
5  /**
6   * В этой программе рефлексия применяется для вывода
7   * всех компонентов класса \
8   * @version 1.1 2004-02-21
9   * @author Cay Horstmann
10 */
11
12 public class ReflectionTest
13 {
14     public static void main(String[] args)
15     {
16         // извлечь имя класса из аргументов командной строки или
17         // введенных пользователем данных
18         String name;
19         if (args.length > 0) name = args[0];
20         else
21         {
22             Scanner in = new Scanner(System.in);
23             System.out.println("Enter class name (e.g. java.util.Date): ");
24             name = in.next();
25         }
26
27         try
28         {
29             // вывести имя класса и суперкласса (if != Object)
30             Class cl = Class.forName(name);
31             Class supercl = cl.getSuperclass();
32             String modifiers = Modifier.toString(cl.getModifiers());
33             if (modifiers.length() > 0) System.out.print(modifiers + " ");
34             System.out.print("class " + name);
35             if (supercl != null && supercl != Object.class)
36                 System.out.print(" extends " + supercl.getName());
37
38             System.out.print("\n\n");
39             printConstructors(cl);
40             System.out.println();
41             printMethods(cl);
42             System.out.println();
43             printFields(cl);
44             System.out.println("}");
45         }
46         catch (ClassNotFoundException e)

```

```
47     {
48         e.printStackTrace();
49     }
50     System.exit(0);
51 }
52
53 /**
54  * Выводит все конструкторы класса
55  * @param cl a class
56  */
57 public static void printConstructors(Class cl)
58 {
59     Constructor[] constructors = cl.getDeclaredConstructors();
60
61     for (Constructor c : constructors)
62     {
63         String name = c.getName();
64         System.out.print(" ");
65         String modifiers = Modifier.toString(c.getModifiers());
66         if (modifiers.length() > 0) System.out.print(modifiers + " ");
67         System.out.print(name + "(");
68
69         // вывести типы параметров
70         Class[] paramTypes = c.getParameterTypes();
71         for (int j = 0; j < paramTypes.length; j++)
72         {
73             if (j > 0) System.out.print(", ");
74             System.out.print(paramTypes[j].getName());
75         }
76         System.out.println(");");
77     }
78 }
79
80 /**
81  * Выводит все методы класса
82  * @param cl a class
83  */
84 public static void printMethods(Class cl)
85 {
86     Method[] methods = cl.getDeclaredMethods();
87     for (Method m : methods)
88     {
89         Class retType = m.getReturnType();
90         String name = m.getName();
91
92         System.out.print(" ");
93         // вывести модификаторы, возвращаемый тип и имя метода
94         String modifiers = Modifier.toString(m.getModifiers());
95         if (modifiers.length() > 0) System.out.print(modifiers + " ");
96         System.out.print(retType.getName() + " " + name + "(");
97
98         // вывести типы параметров
99         Class[] paramTypes = m.getParameterTypes();
100        for (int j = 0; j < paramTypes.length; j++)
101        {
102            if (j > 0) System.out.print(", ");
103            System.out.print(paramTypes[j].getName());
104        }
105        System.out.println(");");
```

```
106 }
107 }
108
109 /**
110  * Выводит все поля класса
111  * @param cl a class
112  */
113 public static void printFields(Class cl)
114 {
115     Field[] fields = cl.getDeclaredFields();
116
117     for (Field f : fields)
118     {
119         Class type = f.getType();
120         String name = f.getName();
121         System.out.print(" ");
122         String modifiers = Modifier.toString(f.getModifiers());
123         if (modifiers.length() > 0) System.out.print(modifiers + " ");
124         System.out.println(type.getName() + " " + name + ";");
125     }
126 }
127 }
```

Эта программа примечательна тем, что она способна анализировать любой класс, загружаемый интерпретатором Java, а не только классы, доступные во время компиляции. В следующей главе мы применим ее для анализа внутренних классов, автоматически генерируемых компилятором Java.

java.lang.Class 1.0

- `Field[] getFields()` 1.1
- `Field[] getDeclaredFields()` 1.1

Метод `getFields()` возвращает массив, который содержит объекты типа `Field`, соответствующие открытым полям анализируемого класса или его суперкласса. А метод `getDeclaredFields()` возвращает массив, содержащий объекты типа `Field`, соответствующие всем полям анализируемого класса. Оба метода возвращают массив нулевой длины, если такие поля отсутствуют или же если объект типа `Class` представляет собой простой тип или массив.

- `Method[] getMethods()` 1.1
- `Method[] getDeclaredMethods()` 1.1

Возвращают массив, который содержит объекты типа `Method`, соответствующие только открытым методам, включая унаследованные (метод `getMethods()`), или же всем методам анализируемого класса или интерфейса, за исключением унаследованных (метод `getDeclaredMethods()`).

- `Constructor[] getConstructors()` 1.1
- `Constructor[] getDeclaredConstructors()` 1.1

Возвращают массив, который содержит объекты типа `Constructor`, соответствующие только открытым конструкторам (метод `getConstructors()`) или же всем конструкторам класса, представленного объектом типа `Class` (метод `getDeclaredMethods()`).

java.lang.reflect.Field 1.1**java.lang.reflect.Method 1.1****java.lang.reflect.Constructor 1.1**

- `Class getDeclaringClass()`
Возвращает объект типа `Class`, соответствующий классу, в котором определен заданный конструктор, метод или поле.
- `Class[] getExceptionTypes()` (в классах `Constructor` и `Method`)
Возвращает массив объектов типа `Class`, представляющих типы исключений, генерируемых заданным методом.
- `int getModifiers()`
Возвращает целое значение, соответствующее модификатору заданного конструктора, метода или поля. Для анализа возвращаемого значения следует использовать методы из класса `Modifier`.
- `String getName()`
Возвращает символьную строку, в которой содержится имя конструктора, метода или поля.
- `Class[] getParameterTypes()` (в классах `Constructor` и `Method`)
Возвращает массив объектов типа `Class`, представляющих типы параметров.
- `Class getReturnType()` (в классе `Method`)
Возвращает объект тип `Class`, соответствующий возвращаемому типу.

java.lang.reflect.Modifier 1.1

- `static String toString(int modifiers)`
Возвращает символьную строку с модификаторами, соответствующими битам, установленным в целочисленном значении параметра `modifiers`.
 - `static boolean isAbstract(int modifiers)`
 - `static boolean isFinal(int modifiers)`
 - `static boolean isInterface(int modifiers)`
 - `static boolean isNative(int modifiers)`
 - `static boolean isPrivate(int modifiers)`
 - `static boolean isProtected(int modifiers)`
 - `static boolean isPublic(int modifiers)`
 - `static boolean isStatic(int modifiers)`
 - `static boolean isStrict(int modifiers)`
 - `static boolean isSynchronized(int modifiers)`
 - `static boolean isVolatile(int modifiers)`
- Проверяют разряды целого значения параметра `modifiers`, которые соответствуют модификаторам доступа, указываемым при объявлении методов.

Анализ объектов во время выполнения с помощью рефлексии

Как следует из предыдущего раздела, для определения имен и типов полей любого объекта достаточно выполнить два действия.

- Получить соответствующий объект типа `Class`.
- Вызвать метод `getDeclaredFields()` для этого объекта.

А в этом разделе мы пойдем дальше и попробуем определить содержимое полей данных. Разумеется, если имя и тип объекта известны при написании программы, это не составляет никакого труда. Но рефлексия позволяет сделать это и для объектов, которые неизвестны на стадии компиляции.

Ключевым в этом процессе является метод `get()` из класса `Field`. Если объект `f` относится к типу `Field` (например, получен в результате вызова метода `getDeclaredFields()`), а объект `obj` относится к тому же самому классу, что и объект `f`, то при вызове `f.get(obj)` возвратится объект, значение которого будет текущим значением поля в объекте `obj`. Покажем действие этого механизма на следующем конкретном примере:

```
Employee harry = new Employee("Harry Hacker", 35000, 10, 1, 1989);
Class cl = harry.getClass();
// объект типа Class, представляющий класс Employee
Field f = cl.getDeclaredField("name");
// поле name из класса Employee
Object v = f.get(harry);
// значение в поле name объекта harry, т.е. объект типа String,
// содержащий символьную строку "Harry Hacker"
```

На первый взгляд, в приведенном выше фрагменте кода нет ничего каверзного, но его выполнение приводит к ошибке. В частности, поле `name` объявлено как `private`, а следовательно, метод `get()` сгенерирует исключение `IllegalAccessException`. Ведь метод `get()` можно применять только к открытым полям. Механизм безопасности Java позволяет определить, какие именно поля содержит объект, но не дает возможности прочитать их содержимое, если эти поля недоступны.

По умолчанию в механизме рефлексии соблюдаются правила доступа, установленные в Java. Но если программа не контролируется диспетчером защиты, то эти правила можно обойти. Чтобы сделать это, достаточно вызвать метод `setAccessible()` для объектов типа `Field`, `Method` или `Constructor`, например, следующим образом:

```
f.setAccessible(true); // теперь можно сделать вызов f.get(harry);
```

Метод `setAccessible()` относится к классу `AccessibleObject`, являющемуся общим суперклассом для классов `Field`, `Method` и `Constructor`. Он нужен для обеспечения нормальной работы отладчиков, поддержки постоянных хранилищ и выполнения других подобных функций. Мы применим его далее в этой главе для создания обобщенного метода `toString()`.

С методом `get()` связано еще одно затруднение. Поле `name` относится к типу `String`, а следовательно, его значение можно вернуть в виде объекта типа `Object`. Но допустим, что требуется определить значение поля `salary`. Оно относится к типу `double`, а в Java числовые типы не являются объектами. Чтобы разрешить это затруднение, можно воспользоваться методом `getDouble()` из класса `Field` или же методом `get()`. Используя механизм рефлексии, любой из этих методов автоматически заключит поле в оболочку соответствующего класса, в данном случае — `Double`.

Разумеется, значение поля можно не только определять, но и задавать. Так, при вызове `f.set(obj, value)` в объекте `obj` задается новое значение `value` поля, представленного переменной `f`.

В примере кода из листинга 5.14 показано, каким образом создается обобщенный метод `toString()`, пригодный для любого класса. Сначала в нем вызывается метод `getDeclaredFields()` для получения всех полей, а затем метод `setAccessible()`, делающий все эти поля доступными. Далее определяется имя и значение каждого поля. Кроме того, в листинге 5.14 демонстрируется способ преобразования значений в символьные строки путем рекурсивного вызова метода `toString()`, как показано ниже.

```
class ObjectAnalyzer
{
    public String toString(Object obj)
    {
        Class cl = obj.getClass();
        . . .
        String r = cl.getName();
        // проверить поля этого класса и всех его суперклассов
        do
        {
            r += "[";
            Field[] fields = cl.getDeclaredFields();
            AccessibleObject.setAccessible(fields, true);
            // получить имена и значения всех полей
            for (Field f : fields)
            {
                if (!Modifier.isStatic(f.getModifiers()))
                {
                    if (!r.endsWith("[") r += ",";
                    r += f.getName() + "=";
                    try
                    {
                        Object val = f.get(obj);
                        r += toString(val);
                    }
                    catch (Exception e) { e.printStackTrace(); }
                }
            }
            r += "]";
            cl = cl.getSuperclass();
        }
        while (cl != null);
        return r;
    }
    . . .
}
```

Код из листинга 5.14 достаточно сложен для понимания. Циклические ссылки могут привести к бесконечной рекурсии, а следовательно, объект `ObjectAnalyzer` (из листинга 5.15) должен отслеживать объекты, которые уже были проанализированы. Кроме того, для просмотра массивов необходим другой подход, который более подробно будет рассмотрен в следующем разделе.

Итак, для просмотра содержимого любого объекта можно воспользоваться методом `toString()`. Рассмотрим следующий вызов:

```
ArrayList<Integer> squares = new ArrayList<>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
System.out.println(new ObjectAnalyzer().toString(squares));
```

В результате выполнения этого фрагмента кода будет выведена следующая информация:

```
java.util.ArrayList[elementData=class java.lang.Object[] {
  java.lang.Integer[value=1] [] [],
  java.lang.Integer[value=4] [] [],
  java.lang.Integer[value=9] [] [],
  java.lang.Integer[value=16] [] [],
  java.lang.Integer[value=25] [] [], null, null, null, null, null},
size=5] [modCount=5] [] []
```

Используя такой обобщенный метод `toString()`, можно реализовать конкретные методы `toString()` в собственных классах. Это можно сделать, например, следующим образом:

```
public String toString()
{
    return new ObjectAnalyzer().toString(this);
}
```

Такое применение метода `toString()` оказывается полезным при отладке программ.

Листинг 5.14. Исходный код из файла `objectAnalyzer/ObjectAnalyzerTest.java`

```
1 package objectAnalyzer;
2
3 import java.util.ArrayList;
4
5 /**
6  * В этой программе рефлексия применяется для слежения за объектами
7  * @version 1.12 2012-01-26
8  * @author Cay Horstmann
9  */
10 public class ObjectAnalyzerTest
11 {
12     public static void main(String[] args)
13     {
14         ArrayList<Integer> squares = new ArrayList<>();
15         for (int i = 1; i <= 5; i++)
16             squares.add(i * i);
17         System.out.println(new ObjectAnalyzer().toString(squares));
18     }
19 }
```

Листинг 5.15. Исходный код из файла `objectAnalyzer/ObjectAnalyzer.java`

```
1 package objectAnalyzer;
2
3 import java.lang.reflect.AccessibleObject;
4 import java.lang.reflect.Array;
5 import java.lang.reflect.Field;
6 import java.lang.reflect.Modifier;
7 import java.util.ArrayList;
8
9 public class ObjectAnalyzer
10 {
```

```
11 private ArrayList<Object> visited = new ArrayList<>();
12
13 /**
14  * Преобразует объект в строковое представление всех перечисляемых полей
15  * @param obj Объект
16  * @return Возвращает строку с именем класса и всеми полями объекта,
17  *         а также их значениями
18  */
19 public String toString(Object obj)
20 {
21     if (obj == null) return "null";
22     if (visited.contains(obj)) return "...";
23     visited.add(obj);
24     Class cl = obj.getClass();
25     if (cl == String.class) return (String) obj;
26     if (cl.isArray())
27     {
28         String r = cl.getComponentType() + "[]{";
29         for (int i = 0; i < Array.getLength(obj); i++)
30         {
31             if (i > 0) r += ",";
32             Object val = Array.get(obj, i);
33             if (cl.getComponentType().isPrimitive()) r += val;
34             else r += toString(val);
35         }
36         return r + "}";
37     }
38
39     String r = cl.getName();
40     // проверить поля этого класса и всех его суперклассов
41     do
42     {
43         r += "[";
44         Field[] fields = cl.getDeclaredFields();
45         AccessibleObject.setAccessible(fields, true);
46         // получить имена и значения всех полей
47         for (Field f : fields)
48         {
49             if (!Modifier.isStatic(f.getModifiers()))
50             {
51                 if (!r.endsWith("[") r += ",";
52                 r += f.getName() + "=";
53                 try
54                 {
55                     Class t = f.getType();
56                     Object val = f.get(obj);
57                     if (t.isPrimitive()) r += val;
58                     else r += toString(val);
59                 }
60                 catch (Exception e)
61                 {
62                     e.printStackTrace();
63                 }
64             }
65         }
66         r += "]";
67         cl = cl.getSuperclass();
```

```
68     }
69     while (cl != null);
70
71     return r;
72 }
73 }
```

java.lang.reflect.AccessibleObject 1.2

- `void setAccessible(boolean flag)`

Устанавливает признак доступности заданного объекта рефлексии. Логическое значение `true` параметра `flag` обозначает, что проверка доступа к компонентам языка Java отменена и закрытые свойства объекта теперь доступны для выборки и установки.

- `boolean isAccessible()`

Получает значение признака доступности заданного объекта рефлексии.

- `static void setAccessible(AccessibleObject[] array, boolean flag)`

Удобен для установки признака доступности массива объектов.

java.lang.Class 1.1

- `Field getField(String name)`

- `Field[] getFields()`

Возвращают общедоступное поле с указанным именем или массив, содержащий все поля.

- `Field getDeclaredField(String name)`

- `Field[] getDeclaredFields()`

Возвращают объявленное в классе поле с указанным именем или же массив, содержащий все поля.

java.lang.reflect.Field 1.1

- `Object get(Object obj)`

Возвращает значение поля объекта `obj`, описываемого данным объектом типа `Field`.

- `void set(Object obj, Object newValue)`

Устанавливает новое значение в поле объекта `obj`, описываемого данным объектом типа `Field`.

Написание кода обобщенного массива с помощью рефлексии

Класс `Array` из пакета `java.lang.reflect` позволяет создавать массивы динамически. Этим можно, например, воспользоваться для реализации метода `copyOf()` в классе `Arrays`. Напомним, что этот метод служит для наращивания уже

заполненного массива. Ниже показано, каким образом такое наращивание реализуется в коде.

```
Employee[] a = new Employee[100];  
.  
.  
.  
// массив заполнен  
a = Arrays.copyOf(a, 2 * a.length);
```

Как написать такой обобщенный метод? В этом нам поможет тот факт, что массив `Employee[]` можно преобразовать в массив `Object[]`. Звучит многообещающе. Итак, предпримем первую попытку создать обобщенный метод следующим образом:

```
public static Object[] badCopyOf(Object[] a, int newLength) // бесполезно  
{  
    Object[] newArray = new Object[newLength];  
    System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));  
    return newArray;  
}
```

Но в этом коде возникает затруднение, связанное с фактическим использованием получаемого в итоге массива. Этот массив содержит объекты и относится к типу `Object[]`, поскольку он создан с помощью следующего выражения:

```
new Object[newLength]
```

Массив объектов типа `Object[]` не может быть преобразован в массив типа `Employee[]`. При попытке сделать это возникнет исключение типа `ClassCastException`. Как пояснялось ранее, в ходе выполнения программы исполняющая система Java запоминает первоначальный тип элементов массива, т.е. тип, указанный в операции `new`. Массив типа `Employee[]` можно временно преобразовать в массив типа `Object[]` и обратно, но массив, изначально созданный как относящийся к типу `Object[]`, преобразовать в массив типа `Employee[]` нельзя. Чтобы написать подходящий обобщенный метод, нужно каким-то образом создать новый массив, тип которого совпадал бы с типом исходного массива. Для этого потребуются методы класса `Array` из пакета `java.lang.reflect` и особенно метод `newInstance()`, создающий новый массив. Тип элементов массива и требуемая его длина должны передаваться обобщенному методу в качестве параметров следующим образом:

```
Object newArray = Array.newInstance(componentType, newLength);
```

Чтобы сделать это, нужно определить длину и тип элементов нового массива. Длину можно получить с помощью метода `Array.getLength()`. Статический метод `getLength()` из класса `Array` возвращает длину любого массива. А для того чтобы определить тип элементов нового массива, необходимо выполнить следующие действия.

1. Определить, какому именно классу принадлежит объект `a`.
2. Убедиться в том, что он действительно является массивом.
3. Воспользоваться методом `getComponentType()` из класса `Class` (определен лишь для объектов типа `Class`, представляющих массивы), чтобы получить требуемый тип массива.

Почему же метод `getLength()` принадлежит классу `Array`, а метод `getComponentType()` — классу `Class`? Вряд ли это известно кому-либо, кроме разработчиков этих классов. Существующее распределение методов по классам приходится иногда принимать таким, каким оно есть.

Ниже приведен исходный код рассматриваемого здесь обобщенного метода.

```
public static Object goodCopyOf(Object a, int newLength)
{
    Class cl = a.getClass();
    if (!cl.isArray()) return null;
    Class componentType = cl.getComponentType();
    int length = Array.getLength(a);
    Object newArray = Array.newInstance(componentType, newLength);
    System.arraycopy(a, 0, newArray, 0, Math.min(length, newLength));
    return newArray;
}
```

Следует иметь в виду, что метод `goodCopyOf()` можно применять для наращивания массива любого типа, а не только массива объектов, как показано ниже.

```
int[] a = { 1, 2, 3, 4, 5 };
a = (int[]) goodCopyOf(a, 10);
```

Для этого параметр метода `goodCopyOf()` объявляется как относящийся к типу `Object`, а не как массив объектов (т.е. типа `Object[]`). Массив типа `int[]` можно преобразовать в объект типа `Object`, но не в массив объектов!

В листинге 5.16 демонстрируется применение обоих вариантов метода `CopyOf()`: `badCopyOf()` и `goodCopyOf()`. Следует, однако, иметь в виду, что приведение типа значения, возвращаемого методом `badCopyOf()`, приведет к генерированию исключения.

Листинг 5.16. Исходный код из файла `arrays/CopyOfTest.java`

```
1 package arrays;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5
6 /**
7  * В этой программе демонстрируется применение рефлексии
8  * для манипулирования массивами
9  * @version 1.2 2012-05-04
10 * @author Cay Horstmann
11 */
12 public class CopyOfTest
13 {
14     public static void main(String[] args)
15     {
16         int[] a = { 1, 2, 3 };
17         a = (int[]) goodCopyOf(a, 10);
18         System.out.println(Arrays.toString(a));
19
20         String[] b = { "Tom", "Dick", "Harry" };
21         b = (String[]) goodCopyOf(b, 10);
22         System.out.println(Arrays.toString(b));
23
24         System.out.println("The following call will generate an exception.");
25         b = (String[]) badCopyOf(b, 10);
26     }
27
28     /**
29      * В этом методе предпринимается попытка нарастить массив путем
30      * выделения нового массива и копирования в него всех прежних элементов
```

```
31 * @param Нарастиваемый массив
32 * @param newLength Новая длина массива
33 * @return Возвращаемый наращенный массив, содержащий все элементы
34 * массива a, но он относится к типу Object[], а не к типу массива a
35 */
36 public static Object[] badCopyOf(Object[] a, int newLength) // бесполезно
37 {
38     Object[] newArray = new Object[newLength];
39     System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));
40     return newArray;
41 }
42
43 /**
44 * Этот метод наращивает массив, выделяя новый массив того же типа
45 * и копируя в него все прежние элементы
46 * @param Нарастиваемый массив. Может быть массивом объектов или же
47 * массивом примитивных типов
48 * @return Возвращаемый наращенный массив, содержащий все
49 * элементы массива a
50 */
51 public static Object goodCopyOf(Object a, int newLength)
52 {
53     Class cl = a.getClass();
54     if (!cl.isArray()) return null;
55     Class componentType = cl.getComponentType();
56     int length = Array.getLength(a);
57     Object newArray = Array.newInstance(componentType, newLength);
58     System.arraycopy(a, 0, newArray, 0, Math.min(length, newLength));
59     return newArray;
60 }
61 }
```

java.lang.reflect.Array 1.1

- static Object get(Object array, int index)
- static xxx getXxx(Object array, int index)
Возвращают значение элемента указанного массива по заданному индексу. (Символами xxx обозначаются примитивные типы boolean, byte, char, double, float, int, long, short.)
- static void set(Object array, int index, Object newValue)
- static setXxx(Object array, int index, xxx newValue)
Устанавливают новое значение в элементе указанного массива по заданному индексу. (Символами xxx обозначаются примитивные типы boolean, byte, char, double, float, int, long, short.)
- static int getLength(Object array)
- Возвращает длину указанного массива.
- static Object newInstance(Class componentType, int length)
- static Object newInstance(Class componentType, int[] lengths)
- Возвращают новый массив, состоящий из компонентов указанного типа и имеющий заданную размерность.

Вызов произвольных методов

В языках C и C++ можно выполнить произвольную функцию по указателю на нее. На первый взгляд, в Java не предусмотрены указатели на методы, т.е. в этом языке отсутствует возможность передавать одному методу адрес другого метода, чтобы последний мог затем вызвать первый. Создатели Java заявили, что указатели на методы небезопасны и часто порождают ошибки, а действия, для которых часто применяются такие указатели, удобнее выполнять с помощью *интерфейсов*, рассматриваемых в следующей главе. Тем не менее механизм рефлексии позволяет вызывать произвольные методы.



НА ЗАМЕТКУ! Среди нестандартных расширений Java, которые корпорация Microsoft внесла в свой язык J++ (и в появившийся затем язык C#), имеется тип указателей на методы, отличающийся от класса `Method`, обсуждаемого в этом разделе. Такие указатели называются *делегатами*. Но внутренние классы, рассматриваемые в следующей главе, оказываются более удобной и универсальной языковой конструкцией, чем делегаты.

Напомним, что поле объекта можно проверить с помощью метода `get()` из класса `Field`. Аналогично класс `Method` содержит метод `invoke()`, позволяющий вызвать метод, заключенный в оболочку текущего объекта этого класса, следующим образом:

```
Object invoke(Object obj, Object... args)
```

Первый параметр этого метода является неявным, а остальные объекты представляют собой явные параметры. Если метод статический, то первый параметр игнорируется, а вместо него можно указать пустое значение `null`. Так, если объект `m1` представляет метод `getName()` из класса `Employee`, то можно сделать следующий вызов:

```
String n = (String) m1.invoke(harry);
```

Если возвращаемый тип оказывается примитивным, метод `invoke()` возвратит вместо него тип объекта-оболочки. Допустим, объект `m2` представляет метод `getSalary()` из класса `Employee`. В таком случае возвращаемый объект-оболочка фактически относится к типу `Double`, и поэтому его необходимо привести к примитивному типу `double`. Это нетрудно сделать с помощью автоматической распаковки следующим образом:

```
double s = (Double) m2.invoke(harry);
```

Как получить объект типа `Method`? Можно, конечно, вызвать метод `getDeclaredMethods()` и найти искомый объект среди возвращаемого массива объектов типа `Method`. Кроме того, можно вызвать метод `getMethod()` из класса `Class`. Его действие можно сравнить с методом `getField()`, получающим символьную строку с именем поля и возвращающим объект типа `Field`. Но методов с одним и тем же именем может быть несколько, и среди них приходится тщательно выбирать нужный метод. Именно по этой причине необходимо также предусмотреть массив, содержащий типы параметров искомого метода. Сигнатура метода `getMethod()` выглядит следующим образом:

```
Method getMethod(String name, Class... parameterTypes)
```

В качестве примера ниже показано, каким образом получают указатели на методы `getName()` и `raiseSalary()` из класса `Employee`.

```
Method m1 = Employee.class.getMethod("getName");
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

Итак, выяснив правила применения объектов типа Method, продемонстрируем их употребление непосредственно в коде. В листинге 5.17 приведена программа, выводящая таблицу значений математической функции вроде `Math.sqrt` или `Math.sin`. Результат выполнения этой программы выглядит следующим образом:

```
public static native double java.lang.Math.sqrt(double)
1.0000 | 1.0000
2.0000 | 1.4142
3.0000 | 1.7321
4.0000 | 2.0000
5.0000 | 2.2361
6.0000 | 2.4495
7.0000 | 2.6458
8.0000 | 2.8284
9.0000 | 3.0000
10.0000 | 3.1623
```

Разумеется, код, осуществляющий вывод таблицы на экран, не зависит от конкретной функции, как показано ниже.

```
double dx = (to - from) / (n - 1);
for (double x = from; x <= to; x += dx)
{
    double y = (Double) f.invoke(null, x);
    System.out.printf("%10.4f | %10.4f\n", x, y);
}
```

Здесь `f` — это объект типа Method. А поскольку вызывается статический метод, то в качестве первого параметра методу `invoke()` передается пустое значение `null`. Для вывода таблицы со значениями математической функции `Math.sqrt` служит следующая строка кода:

```
Math.class.getMethod("sqrt", double.class)
```

При вызове метода `getMethod()` ему передается имя метода `sqrt()` из класса `Math` и параметр типа `double`. В листинге 5.17 приведен весь исходный код обобщенного варианта программы табличного вывода значений функции.

Листинг 5.17. Исходный код из файла `methods/MethodTableTest.java`

```
1 package methods;
2
3 import java.lang.reflect.*;
4
5 /**
6  * В этой программе демонстрируется применение рефлексии для вызова методов
7  * @version 1.2 2012-05-04
8  * @author Cay Horstmann
9  */
10 public class MethodTableTest
11 {
12     public static void main(String[] args) throws Exception
13     {
14         // получить указатели на методы square() и sqrt()
15         Method square = MethodTableTest.class.getMethod("square", double.class);
```

```
16     Method sqrt = Math.class.getMethod("sqrt", double.class);
17
18     // вывести значения x и y в табличном виде
19
20     printTable(1, 10, 10, square);
21     printTable(1, 10, 10, sqrt);
22 }
23
24 /**
25  * Возвращает квадрат числа
26  * @param x Число
27  * @return x Квадрат числа
28  */
29 public static double square(double x)
30 {
31     return x * x;
32 }
33 /**
34  * Выводит в табличном виде значения x и y указанного метода
35  * @param Нижняя граница значений x
36  * @param Верхняя граница значений x
37  * @param n Количество строк в таблице
38  * @param f Метод, получающий и возвращающий значение типа double
39  */
40 public static void printTable(double from, double to, int n, Method f)
41 {
42     // вывести сигнатуру метода в заголовке таблицы
43     System.out.println(f);
44
45     double dx = (to - from) / (n - 1);
46
47     for (double x = from; x <= to; x += dx)
48     {
49         try
50         {
51             double y = (Double) f.invoke(null, x);
52             System.out.printf("%10.4f | %10.4f%n", x, y);
53         }
54         catch (Exception e)
55         {
56             e.printStackTrace();
57         }
58     }
59 }
60 }
```

Таким образом, с помощью объектов типа `Method` можно делать то же, что и с помощью указателей на функции в С (или делегатов в С#). Как и в С, такой стиль программирования обычно неудобен и часто приводит к ошибкам. Что, если, например, вызвать метод `invoke()` с неверно заданными параметрами? В этом случае метод `invoke()` сгенерирует исключение.

Кроме того, параметры метода `invoke()` и возвращаемое им значение обязательно должны быть типа `Object`. А это влечет за собой приведение типов в соответствующих местах кода. В итоге компилятор будет лишен возможности тщательно проверить исходный код программы. Следовательно, ошибки в ней проявятся только на стадии тестирования, когда исправить их будет намного труднее. Более того,

программа, использующая механизм рефлексии для получения указателей на методы, работает заметно медленнее, чем программа, непосредственно вызывающая эти методы.

По этим причинам пользоваться объектами типа `Method` рекомендуется только в самом крайнем случае. Намного лучше применять интерфейсы и внутренние классы, рассматриваемые в следующей главе. В частности, следуя указаниям разработчиков Java, объекты типа `Method` не рекомендуется применять для организации функций обратного вызова, поскольку для этой цели вполне подходят интерфейсы, позволяющие создавать программы, которые работают намного быстрее и надежнее.

`java.lang.reflect.Method 1.1`

- `public Object invoke(Object implicitParameter, Object[] explicitParameters)`
Вызывает метод, описанный в объекте, передавая ему заданные параметры и возвращая значение, вычисленное этим методом. Для статических методов в качестве неявного параметра передается пустое значение `null`. Прimitives типы следует передавать только в виде объектных оболочек классов. Возвращаемые значения примитивных типов должны извлекаться из объектных оболочек путем автоматической распаковки.

Рекомендации по применению наследования

В завершение этой главы приведем некоторые рекомендации относительно надлежащего к применению очень полезного механизма наследования.

1. Размещайте общие операции и поля в суперклассе.

Поле `name` было размещено в класс `Person` именно потому, чтобы не повторять его в классах `Employee` и `Student`.

2. Старайтесь не пользоваться защищенными полями.

Некоторые разработчики полагают, что следует “на всякий случай” объявлять большинство полей защищенными, чтобы подклассы могли обращаться к ним по мере надобности. Но имеются две веские причины, по которым такой механизм не гарантирует достаточной защиты. Во-первых, множество подклассов неограниченно. Всякий может создать подкласс, производный от данного класса, и написать программу, получающую непосредственный доступ к защищенным полям его экземпляра, нарушая инкапсуляцию. И во-вторых, в Java к защищенным полям имеют доступ все классы, находящиеся в том же самом пакете, независимо от того, являются ли они подклассами данного класса или нет. В то же время полезно объявлять защищенными методы, которые не предназначены для общего употребления и должны быть переопределены в подклассах.

3. Используйте наследование для моделирования отношений “является”.

Наследование позволяет экономить время и труд при разработке программ, но иногда им злоупотребляют. Допустим, требуется создать класс `Contractor`. У сотрудника, нанимаемого по контракту, имеется свое имя и дата заключения договора, но у него нет оклада. У него почасовая оплата, причем он работает не так давно, чтобы повышать оплату его труда. Ниже показано, как можно

сделать класс Contractor подклассом, производным от класса Employee, добавив поле hourlyWage.

```
class Contractor extends Employee
{
    private double hourlyWage;
    . . .
}
```

Но это не совсем удачная идея. Ведь в этом случае получается, что каждый сотрудник, нанятый по контракту, имеет и оклад, и почасовую оплату. Если вы попытаетесь реализовать методы для распечатки платежных и налоговых ведомостей, то сразу же проявится недостаток такого подхода. Программа, которую вам придется написать, будет гораздо длиннее той, которую вы могли бы создать, не прибегая к неоправданному наследованию.

Отношение “контрактный сотрудник–постоянный сотрудник” не удовлетворяет критерию “является”. Сотрудники, нанятые по контракту, не являются постоянными и относятся к особой категории сотрудников.

4. Не пользуйтесь наследованием, если не все методы имеет смысл сделать наследуемыми.

Допустим, требуется создать класс Holiday. Разумеется, праздники — это разновидность календарных дней, а дни можно представить в виде объектов типа GregorianCalendar, поэтому наследование можно применить следующим образом:

```
class Holiday extends GregorianCalendar(...)
```

К сожалению, множество праздников оказывается *незамкнутым* при наследовании. Среди открытых методов из класса GregorianCalendar имеется метод add(), который может превратить праздники в будни следующим образом:

```
Holiday christmas;
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

Следовательно, наследование в данном случае не подходит.

5. Переопределяя метод, не изменяйте предполагаемое его поведение.

Принцип подстановки распространяется не только на синтаксис, но и на поведение, что важнее. При переопределении метода не следует без особых на то причин изменять его поведение. В этом компилятор вам не поможет. Ведь он не в состоянии проверить, оправдано ли переопределение метода. Допустим, требуется устранить упомянутый выше недостаток метода add() из класса Holiday, переопределив этот метод таким образом, чтобы он, например, не выполнял никаких действий или же возвращал следующий праздничный день. Но такое переопределение нарушает принцип подстановки. При выполнении приведенной ниже последовательности операторов пользователь вправе ожидать вполне *определенного* поведения и соответствующего результата, независимо от того, является ли объект x экземпляром класса GregorianCalendar или Holiday.

```
int d1 = x.get(Calendar.DAY_OF_MONTH);
x.add(Calendar.DAY_OF_MONTH, 1);
int d2 = x.get(Calendar.DAY_OF_MONTH);
System.out.println(d2 - d1);
```

Очевидно, что приведенный выше пример несколько надуман. Разные пользователи посчитают естественным различное поведение программы. Так, существует мнение, что принцип подстановки требует, чтобы в методе `Manager.equals()` не учитывалась премия в поле `bonus`, поскольку она игнорируется в методе `Employee.equals()`. Подобные споры могут длиться бесконечно и не дать никакого результата. Поэтому, принимая конкретное решение, следует руководствоваться теми целями, для которых создается программа.

6. *Пользуйтесь принципом полиморфизма, а не данными о типе.*

Вспомните о принципе полиморфизма, как только увидите код, имеющий следующий вид:

```
if (x типа 1)
    действие1(x);
else if (x типа 2)
    действие2(x);
```

Имеют ли `действие_1` и `действие_2` общий характер? Если имеют, то поместите соответствующие методы в общий суперкласс или интерфейс обоих типов. Тогда можно просто сделать приведенный ниже вызов и выполнить правильное действие с помощью механизма динамического связывания, присущего полиморфизму.

```
x.действие();
```

Код, в котором применяется принцип полиморфизма или реализован интерфейс, намного легче сопровождать и расширять, чем код, изобилующий проверками типов.

7. *Не злоупотребляйте механизмом рефлексии.*

Механизм рефлексии позволяет создавать программы с высоким уровнем абстракции, где поля и методы определяются во время выполнения. Такая возможность чрезвычайно полезна для системного программирования, но для прикладного — практически не нужна. Рефлексия — очень хрупкий механизм, поскольку компилятор не может помочь в обнаружении ошибок. Все ошибки проявляются во время выполнения программы и приводят к возникновению исключений.

В этой главе было показано, каким образом в Java поддерживаются основные понятия и принципы ООП: классы, наследование и полиморфизм. А в следующей главе мы затронем две более сложные темы, которые очень важны для эффективного программирования на Java: интерфейсы и внутренние классы.

Интерфейсы и внутренние классы

В этой главе...

- ▶ Интерфейсы
- ▶ Клонирование объектов
- ▶ Интерфейсы и обратные вызовы
- ▶ Внутренние классы
- ▶ Прокси-классы

Итак, вы ознакомились со всеми основными инструментами для объектно-ориентированного программирования на Java. В этой главе будет представлен ряд усовершенствованных и широко применяемых методик программирования. Несмотря на то что эти методики не совсем очевидны, владеть ими должен всякий, стремящийся профессионально программировать на Java.

Первая из этих методик называется *интерфейсами* и позволяет указывать, что именно должны делать классы, не уточняя, как именно они должны это делать. В классах может быть реализован один или несколько интерфейсов. Если возникает потребность в интерфейсе, применяются объекты этих классов. Рассмотрев интерфейсы, мы перейдем к клонированию объектов, которое иногда называют *полным копированием*. Клоном объекта является новый объект, имеющий точно такое же состояние. В частности, клон можно видоизменить, вообще не затрагивая оригинал.

Далее мы обсудим механизм *внутренних классов*. С технической точки зрения внутренние классы довольно сложны — они определяются внутри других классов, а их методы имеют доступ к полям окружающего класса. Внутренние классы оказываются полезными при разработке коллекций взаимосвязанных классов. Они, в частности, позволяют создавать компактный, высокопрофессиональный код для реализации событийно-ориентированного ГПИ.

И в завершение главы будут представлены *прокси-классы*, реализующие произвольные интерфейсы. Прокси-классы представляют собой весьма специфические конструкции, полезные для создания инструментальных средств системного программирования. При первом чтении книги вы можете благополучно пропустить эту заключительную часть главы.

Интерфейсы

Интерфейс в Java не является классом. Он представляет собой множество *требований*, предъявляемых к классу, который должен соответствовать интерфейсу. Как правило, один разработчик, собирающийся воспользоваться трудами другого разработчика для решения конкретной задачи, заявляет: “Если ваш класс будет соответствовать определенному интерфейсу, я смогу решить свою задачу”. Обратимся к конкретному примеру. Метод `sort()` из класса `Array` позволяет упорядочить массив объектов при одном условии: объекты должны принадлежать классам, реализующим интерфейс `Comparable`.

Вот как выглядит этот интерфейс:

```
public interface Comparable
{
    int compareTo(Object other);
}
```

Это означает, что любой класс, реализующий интерфейс `Comparable`, должен иметь метод `compareTo()`, получающий параметр типа `Object` и возвращающий целое значение.



НА ЗАМЕТКУ! Начиная с версии Java SE 5.0, интерфейс `Comparable` стал обобщенным, как показано ниже.

```
public interface Comparable<T>
{
    int compareTo(T other); // этот параметр относится к типу T
}
```

Так, если класс реализует интерфейс `Comparable<Employee>`, он должен содержать следующий метод:

```
int compareTo(Employee other);
```

По желанию можно по-прежнему пользоваться базовым (иначе называемым “сырым”) типом `Comparable`, но в этом случае придется явным образом приводить параметр метода `compareTo()` к требуемому типу.

Все методы интерфейса автоматически считаются открытыми, поэтому, объявляя метод в интерфейсе, указывать модификатор доступа `public` необязательно. Разумеется, существует и неявное требование: метод `compareTo()` должен на самом деле быть способным сравнивать два объекта и возвращать признак того, что один из них больше другого. В таком случае этот метод должен возвращать отрицательное числовое значение, если объект `x` меньше объекта `y`, нулевое значение — если они равны, в противном случае — положительное числовое значение.

Данный конкретный интерфейс имеет единственный метод, а у некоторых интерфейсов может быть больше одного метода. Как станет ясно в дальнейшем, с помощью

интерфейсов можно также объявлять константы. Но важнее другое: интерфейсы не могут реализовываться в виде объектов. Ведь у интерфейсов нет ни полей, ни тела методов — все это содержится в классах, реализующих соответствующие интерфейсы. Таким образом, интерфейс можно воспринимать как абстрактный класс, лишенный всяких полей экземпляра. Но у этих двух понятий имеется существенное отличие, которое будет подробнее проанализировано далее в этой главе.

Допустим теперь, что требуется воспользоваться методом `sort()` из класса `Array` для сортировки объектов типа `Employee`. В этом случае класс `Employee` должен реализовать интерфейс `Comparable`.

Для того чтобы класс реализовал интерфейс, нужно выполнить два действия.

1. Объявить, что класс реализует интерфейс.
2. Определить в классе все методы, указанные в интерфейсе.

Для того чтобы объявить, что класс реализует интерфейс, служит ключевое слово `implements`:

```
class Employee implements Comparable
```

Разумеется, теперь нужно реализовать метод `compareTo()`, например, для сравнения зарплаты сотрудников. Ниже приведен код, реализующий метод `compareTo()` в классе `Employee`.

```
public int compareTo(Object otherObject)
{
    Employee other = (Employee) otherObject;
    return Double.compare(salary, other.salary);
}
```

В приведенном выше фрагменте кода применяется статический метод `Double.compare()`, возвращающий отрицательное числовое значение, если первый его аргумент меньше второго; нулевое значение, если аргументы равны, иначе — положительное числовое значение.



ВНИМАНИЕ! В интерфейсе при объявлении метода `compareTo()` модификатор доступа `public` не указывается, поскольку все методы интерфейса автоматически являются открытыми. Но при реализации интерфейса этот модификатор доступа должен быть указан. В противном случае компилятор предположит, что область действия этого метода ограничивается пакетом, хотя по умолчанию она не выходит за пределы класса, а в итоге он выдаст предупреждение о попытке ослабить права доступа.

Начиная с Java SE 5.0, можно принять более изящное решение, реализовав интерфейс `Comparable<Employee>`, как показано ниже.

```
class Employee implements Comparable<Employee>
{
    public int compareTo(Employee other)
    {
        return Double.compare(salary, other.salary);
    }
    . . .
}
```

Как видите, теперь тип `Object` не приходится приводить к требуемому типу.



СОВЕТ. Метод `compareTo()` из интерфейса `Comparable` возвращает целое значение. Если объекты не равнозначны, возвращается положительное или отрицательное числовое значение. Возможность использовать в качестве признака неравенства любое число, отличное от нуля, может оказаться полезной при сравнении целочисленных полей. Допустим, у каждого сотрудника имеется однозначный идентификационный номер. В этом случае можно вернуть разность идентификационных номеров `id - other.id`. Она будет отрицательной, если идентификационный номер первого сотрудника меньше идентификационного номера второго сотрудника, нулевой, если номера равны, а иначе — положительной. Следует, однако, иметь в виду, что пределы изменения целых значений должны быть достаточно малы, чтобы вычитание не привело к переполнению. Если заранее известно, что идентификационный номер не является отрицательным или его абсолютное значение не превышает величину $(Integer.Max_Value - 1) / 2$, то можно смело применять рассматриваемый здесь способ сравнения.

Разумеется, такой способ вычитания не подходит для чисел с плавающей точкой. Разность `salary - other.salary` может быть округлена до нуля, если размеры зарплат очень близки, но не одинаковы. В результате вызова `Double.compare(x, y)` просто возвращается значение `-1`, если `x < y`, или значение `1`, если `x > 0`.

Теперь вам должно быть ясно, что для сортировки объектов достаточно реализовать в классе метод `compareTo()`. И такой подход вполне оправдан. Ведь должен же существовать какой-то способ воспользоваться методом `sort()` для сравнения объектов. Но почему нельзя в классе `Employee` просто предусмотреть метод `compareTo()`, не реализуя интерфейс `Comparable`?

Дело в том, что Java — строго типизированный язык программирования. При вызове какого-нибудь метода компилятор должен убедиться, что этот метод действительно существует. В теле метода `sort()` могут находиться операторы, аналогичные следующим:

```
if (a[i].compareTo(a[j]) > 0)
{
    // переставить объекты a[i] и a[j]
    ...
}
```

Компилятору должно быть известно, что у объекта `a[i]` действительно имеется метод `compareTo()`. Если переменная `a` содержит массив объектов, реализующих интерфейс `Comparable`, то существование такого метода гарантируется, поскольку каждый класс, реализующий данный интерфейс, по определению должен предоставлять метод `compareTo()`.



НА ЗАМЕТКУ! На первый взгляд может показаться, что метод `sort()` из класса `Array` оперирует только массивами типа `Comparable[]` и что компилятор выдаст предупреждение, как только обнаружит вызов метода `sort()` для массива, элементы которого не реализуют данный интерфейс. Увы, это не так. Вместо этого метод `sort()` принимает массивы типа `Object[]` и выполняет приведение типов, как показано ниже.

```
// Такой подход принят в стандартной библиотеке,
// но применять его все же не рекомендуется
if (((Comparable) a[i]).compareTo(a[j]) > 0)
{
    // переставить объекты a[i] и a[j]
    . . .
}
```

Если объект `a[i]` не принадлежит классу, реализующему интерфейс `Comparable`, виртуальная машина сгенерирует исключение.

В листинге 6.1 приведен исходный код программы для сортировки массива, состоящего из объектов класса `Employee` (из листинга 6.2).

Листинг 6.1. Исходный код из файла `interfaces/EmployeeSortTest.java`

```
1 package interfaces;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируется применение интерфейса Comparable
7  * @version 1.30 2004-02-27
8  * @author Cay Horstmann
9  */
10 public class EmployeeSortTest
11 {
12     public static void main(String[] args)
13     {
14         Employee[] staff = new Employee[3];
15
16         staff[0] = new Employee("Harry Hacker", 35000);
17         staff[1] = new Employee("Carl Cracker", 75000);
18         staff[2] = new Employee("Tony Tester", 38000);
19
20         Arrays.sort(staff);
21
22         // вывести данные обо всех объектах типа Employee
23         for (Employee e : staff)
24             System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
25     }
26 }
```

Листинг 6.2. Исходный код из файла `interfaces/Employee.java`

```
1 package interfaces;
2
3 public class Employee implements Comparable<Employee>
4 {
5     private String name;
6     private double salary;
7
8     public Employee(String n, double s)
9     {
10         name = n;
11         salary = s;
12     }
13
14     public String getName()
15     {
16         return name;
17     }
18 }
```

```
18
19 public double getSalary()
20 {
21     return salary;
22 }
23
24 public void raiseSalary(double byPercent)
25 {
26     double raise = salary * byPercent / 100;
27     salary += raise;
28 }
29
30 /**
31  * Сравнивает зарплату сотрудников
32  * @param other Другой объект типа Employee
33  * @return Возвращает отрицательное числовое значение, если зарплата
34  *         данного сотрудника ниже, чем у другого; нулевое значение, если
35  *         зарплата одинаковая; а иначе – положительное числовое значение
36  */
37 public int compareTo(Employee other)
38 {
39     return Double.compare(salary, other.salary);
40 }
41 }
```

java.lang.Comparable<T> 1.0

- `int compareTo(T other)`

Сравнивает текущий объект с объектом *other* и возвращает отрицательное целое значение, если текущий объект меньше, чем объект *other*; нулевое значение, если объекты равны; в противном случае — положительное целое значение.

java.util.Arrays 1.2

- `static void sort(Object[] a)`

Сортирует элементы массива *a* по алгоритму слияния. Все элементы массива должны соответствовать классам, реализующим интерфейс `Comparable`, и быть совместимыми друг с другом.

java.lang.Integer 7

- `static int compare(int x, int y)`

Возвращает отрицательное целое значение, если $x < y$; нулевое значение — $x = y$; в противном случае — положительное целое значение.

java.lang.Double 7

- `static int compare(double x, double y)`

Возвращает отрицательное целое значение, если $x < y$; нулевое значение — $x = y$; в противном случае — положительное целое значение.



НА ЗАМЕТКУ! В Java имеется следующее стандартное требование: “Автор реализации метода должен гарантировать, что для всех объектов x и y выполняется условие $\text{sgn}(x.\text{compareTo}(y)) = -\text{sgn}(y.\text{compareTo}(x))$. (Это означает, что если при вызове $y.\text{compareTo}(x)$ генерируется исключение, то же самое должно происходить и при вызове $x.\text{compareTo}(y)$.)” Здесь sgn означает знак числа: $\text{sgn}(n)$ равно -1 , если число n отрицательное; 0 , если число n равно нулю; а также 1 , если число n положительное. Иными словами, если поменять местами явный и неявный параметры метода $\text{compareTo}()$, знак возвращаемого числового значения (но не обязательно его фактическая величина) также должен измениться на противоположный.

Что касается метода $\text{equals}()$, то при наследовании могут возникнуть определенные затруднения. В частности, класс `Manager` расширяет класс `Employee`, а следовательно, он реализует интерфейс `Comparable<Employee>`, а не интерфейс `Comparable<Manager>`, как показано ниже.

```
class Manager extends Employee
{
    public int compareTo(Employee other)
    {
        Manager otherManager = (Manager) other; // Так нельзя!
        . . .
    }
    . . .
}
```

Но в этом случае нарушается правило “антисимметрии”. Если объект x относится к классу `Employee`, а объект y — к классу `Manager`, то вызов $x.\text{compareTo}(y)$ не приведет к возникновению исключения, поскольку x и y будут сравнены как объекты класса `Employee`. А при противоположном вызове $y.\text{compareTo}(x)$ будет сгенерировано исключение `ClassCastException`. Аналогичная ситуация возникает при реализации метода $\text{equals}()$, которая обсуждалась в главе 5. Из этого затруднительного положения имеются два выхода.

Если у подклассов разные обозначения для сравнения, то такое сравнение недопустимо. Таким образом, каждый метод $\text{compareTo}()$ должен начинаться со следующей проверки:

```
if (getClass() != other.getClass()) throw new ClassCastException();
```

Если же существует общий алгоритм сравнения объектов подклассов, то в суперклассе следует реализовать единый метод $\text{compareTo}()$ и объявить его как `final`.

Допустим, в организации руководящие сотрудники считаются выше рядовых по рангу, независимо от зарплаты. Как в таком случае быть с другими подклассами, например `Executive` и `Secretary`? Если требуется учредить нечто вроде табеля о рангах, то в класс `Employee` следует ввести метод $\text{rank}()$. Тогда в каждом подклассе метод $\text{rank}()$ должен переопределяться, а результаты его работы учитываться при выполнении метода $\text{compareTo}()$.

Свойства интерфейсов

Интерфейсы — это не классы. В частности, с помощью операции `new` нельзя создать экземпляр интерфейса следующим образом:

```
x = new Comparable(...); // Неверно!
```

Но, несмотря на то, что конструировать интерфейсные объекты нельзя, объявлять интерфейсные переменные можно следующим образом:

```
Comparable x; // Верно!
```

При этом интерфейсная переменная должна ссылаться на объект класса, реализующего данный интерфейс, как в приведенном ниже фрагменте кода.

```
x = new Employee(...); // Верно, если класс Employee  
                        // реализует интерфейс Comparable
```

Как известно, в ходе операции `instanceof` проверяется, принадлежит ли объект заданному классу. Но с помощью этой операции можно также проверить, реализует ли объект заданный интерфейс:

```
if (anObject instanceof Comparable) { ... }
```

Аналогично классам, интерфейсы также могут образовывать иерархию наследования. Это позволяет создавать цепочки интерфейсов в направлении от более абстрактных к более специализированным. Допустим, имеется следующий интерфейс `Moveable`:

```
public interface Moveable  
{  
    void move(double x, double y);  
}
```

В таком случае можно создать интерфейс `Powered`, расширяющий интерфейс `Moveable` следующим образом:

```
public interface Powered extends Moveable  
{  
    double milesPerGallon();  
}
```

И хотя в интерфейсе не может быть ни полей экземпляров, ни статических методов, в нем можно объявлять константы, как показано ниже.

```
public interface Powered extends Moveable  
{  
    double milesPerGallon();  
    double SPEED_LIMIT = 95; // открытая статическая конечная константа  
}
```

Как и методы, поля для констант в интерфейсах автоматически становятся открытыми. А кроме того, они являются статическими и конечными (т.е. имеют по умолчанию модификаторы доступа `public static final`).



НА ЗАМЕТКУ! Если при объявлении метода в интерфейсе указать ключевое слово `public`, а поля обозначить как `public static final`, это не будет ошибкой. Некоторые программисты поступают так по привычке или для того, чтобы исходные коды их программ были более удобочитаемыми. Но в спецификации Java не рекомендуется употреблять лишние ключевые слова, и в данной книге мы следуем этим рекомендациям.

В некоторых интерфейсах объявляются только константы и ни одного метода. Например, в стандартной библиотеке содержится интерфейс `SwingConstants`, определяющий константы `NORTH`, `SOUTH`, `HORIZONTAL` и т.д. Любой класс, реализующий интерфейс `SwingConstants`, автоматически наследует эти константы. Его методы могут, например, непосредственно ссылаться на константу `NORTH`, не используя громоздкое обозначение `SwingConstants.NORTH`. Но многие специалисты считают, что такое использование интерфейсов со временем изживет себя. Мы также не рекомендуем применять их с этой целью.

В Java любой класс может иметь только один суперкласс, но в то же время любой класс может реализовывать несколько интерфейсов. Это позволяет максимально гибко определять поведение класса. Например, в Java имеется очень важный интерфейс `Cloneable`, подробнее рассматриваемый в следующем разделе. Так, если некоторый класс реализует интерфейс `Cloneable`, для создания точных копий его объектов можно применять метод `clone()` из класса `Object`. Допустим далее, что требуется не только создавать клоны объектов данного класса, но и сравнивать их. Тогда нужно просто реализовать в этом классе оба интерфейса, `Cloneable` и `Comparable`, следующим образом:

```
class Employee implements Cloneable, Comparable
```

Для разделения имен интерфейсов, задающих свойства классов, служит запятая.

Интерфейсы и абстрактные классы

Если вы помните содержание раздела главы 5, посвященного абстрактным классам, то у вас могут возникнуть следующие резонные вопросы: зачем разработчики языка Java создали механизм интерфейсов и почему бы не сделать интерфейс `Comparable` абстрактным классом, например, так, как показано ниже?

```
abstract class Comparable // Почему бы и нет?
{
    public abstract int compareTo(Object other);
}
```

В этом случае рассматриваемый здесь класс `Employee` мог бы просто расширять абстрактный класс и реализовывать метод `compareTo()` следующим образом:

```
class Employee extends Comparable // Почему бы и нет?
{
    public int compareTo(Object other) { ... }
}
```

К сожалению, это породило бы массу проблем, связанных с использованием абстрактного базового класса для выражения обобщенного свойства. Ведь класс может расширять только один класс. Допустим, класс `Employee` уже является подклассом какого-нибудь другого класса, скажем, `Person`. Это означает, что он уже не может расширять еще один класс следующим образом:

```
class Employee extends Person, Comparable // ОШИБКА!
```

Но в то же время каждый класс может реализовывать сколько угодно интерфейсов, как показано ниже.

```
class Employee extends Person implements Comparable // Верно!
```

В других языках программирования и, в частности, в C++ у классов может быть несколько суперклассов. Это языковое средство называется *множественным*

наследованием. Создатели Java решили не поддерживать множественное наследование, поскольку оно делает язык слишком сложным (как C++) или менее эффективным (как Eiffel). В то же время интерфейсы предоставляют большинство возможностей множественного наследования, не усложняя язык и не снижая его эффективность.



НА ЗАМЕТКУ C++! В C++ допускается множественное наследование. Это вызывает много осложнений, связанных с виртуальными базовыми классами, правилами доминирования и приведением типов указателей. Множественным наследованием пользуются лишь немногие программирующие на C++. Некоторые вообще им не пользуются, а остальные рекомендуют применять множественное наследование только в "примешанном" виде. Это означает, что первичный базовый класс описывает родительский объект, а дополнительные базовые классы (так называемые примеси) описывают вспомогательные свойства. Такой подход чем-то напоминает те классы в Java, у которых имеется только один базовый класс и дополнительные интерфейсы. Но в C++ примеси позволяют добавлять некоторые виды поведения по умолчанию, тогда как интерфейсы в Java на это не способны.

Клонирование объектов

После создания копии переменной оригинал и копия представляют собой ссылки на один и тот же объект (рис. 6.1). Это означает, что изменение одной переменной повлечет за собой изменение другой:

```
Employee original = new Employee("John Public", 50000);  
Employee copy = original;  
copy.raiseSalary(10); // Оригинал тоже изменился!
```

Если же требуется, чтобы переменная `copy` представляла новый объект, который в первый момент своего существования идентичен объекту `original`, но совершенно независим от него, в таком случае нужно воспользоваться методом `clone()` следующим образом:

```
Employee copy = original.clone();  
copy.raiseSalary(10); // Теперь оригинал не изменился!
```

Но не все так просто. Метод `clone()` является защищенным (`protected`) в классе `Object`, т.е. его нельзя вызвать непосредственно. И только класс `Employee` может клонировать объекты своего класса. Для этого ограничения имеется своя веская причина. Проанализируем, каким образом класс `Object` может реализовать метод `clone()`. Ему вообще ничего не известно об объекте, поэтому он может копировать лишь поля. Если все поля класса являются числовыми или имеют другой основной тип, их копирование выполняется прекрасно. Но если объект содержит ссылку на под-объект, то оригинал и клонированные объекты будут совместно использовать одни и те же данные.

Чтобы проиллюстрировать это явление, рассмотрим класс `Employee`, которым, начиная с главы 4, мы пользуемся для демонстрации различных аспектов работы с объектами. На рис. 6.2 показано, что происходит, когда метод `clone()` из класса `Object` применяется для клонирования объекта типа `Employee`. Как видите, операция клонирования по умолчанию является "неполной" — она не копирует объекты, на которые имеются ссылки в других объектах.

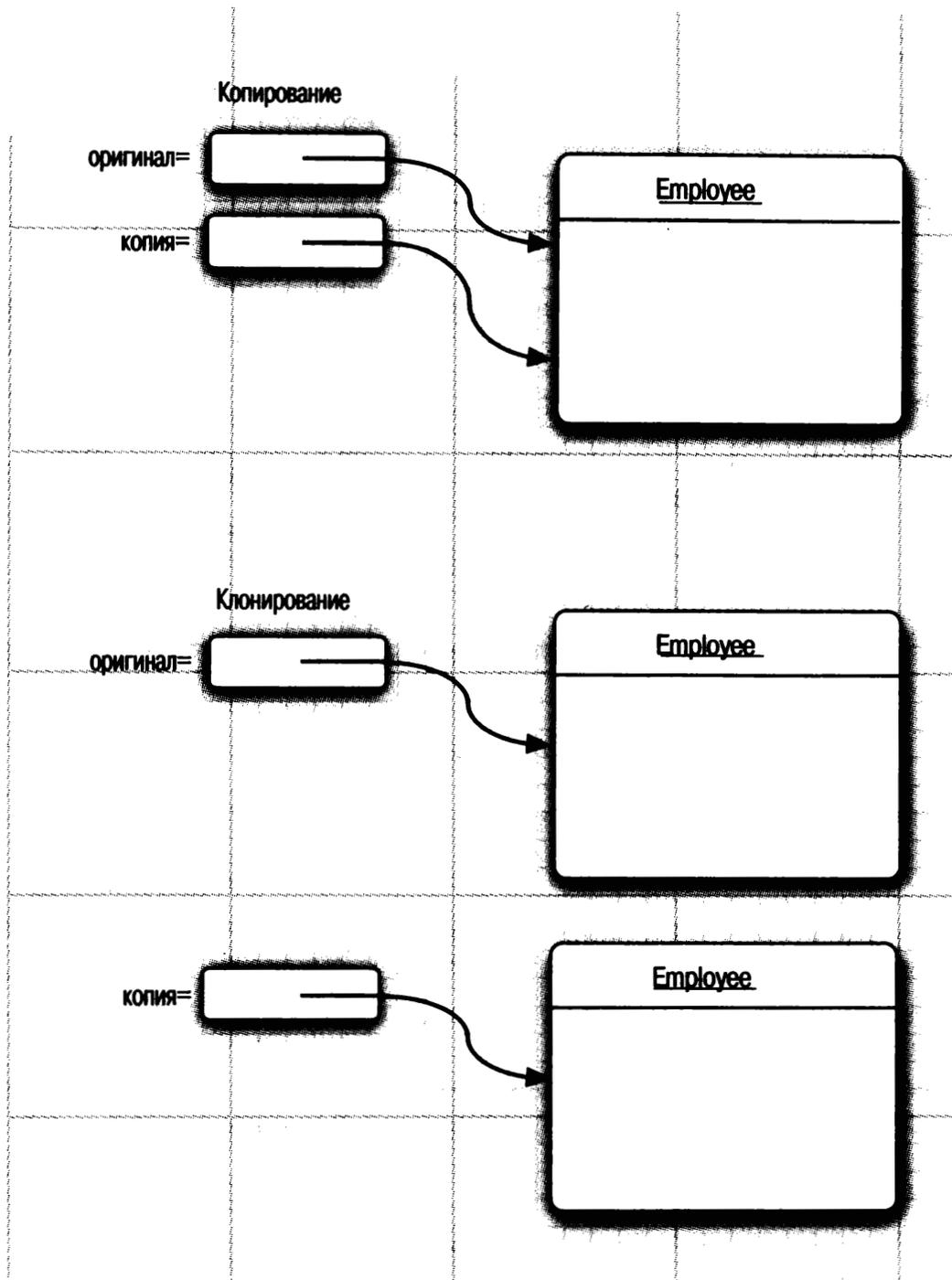


Рис. 6.1. Копирование и клонирование

Так ли уж плохо неполное копирование? Все зависит от конкретной ситуации. Если подобъект, используемый совместно как оригиналом, так и неполным клоном, является неизменяемым, это вполне безопасно. Такое случается, если подобъект является экземпляром неизменяемого класса, например `String`. С другой стороны, подобъект может оставаться постоянным на протяжении всего срока действия того объекта, который его содержит, не подвергаясь воздействию модифицирующих методов или методов, вычисляющих ссылку на него.

Но подобъекты зачастую подвергаются изменениям, поэтому приходится переопределять метод `clone()`, чтобы выполнить *полное* копирование, которое позволяет клонировать подобъекты наряду с содержащими их объектами. В данном примере поле `hireDay` ссылается на экземпляр изменяемого класса `Date`.

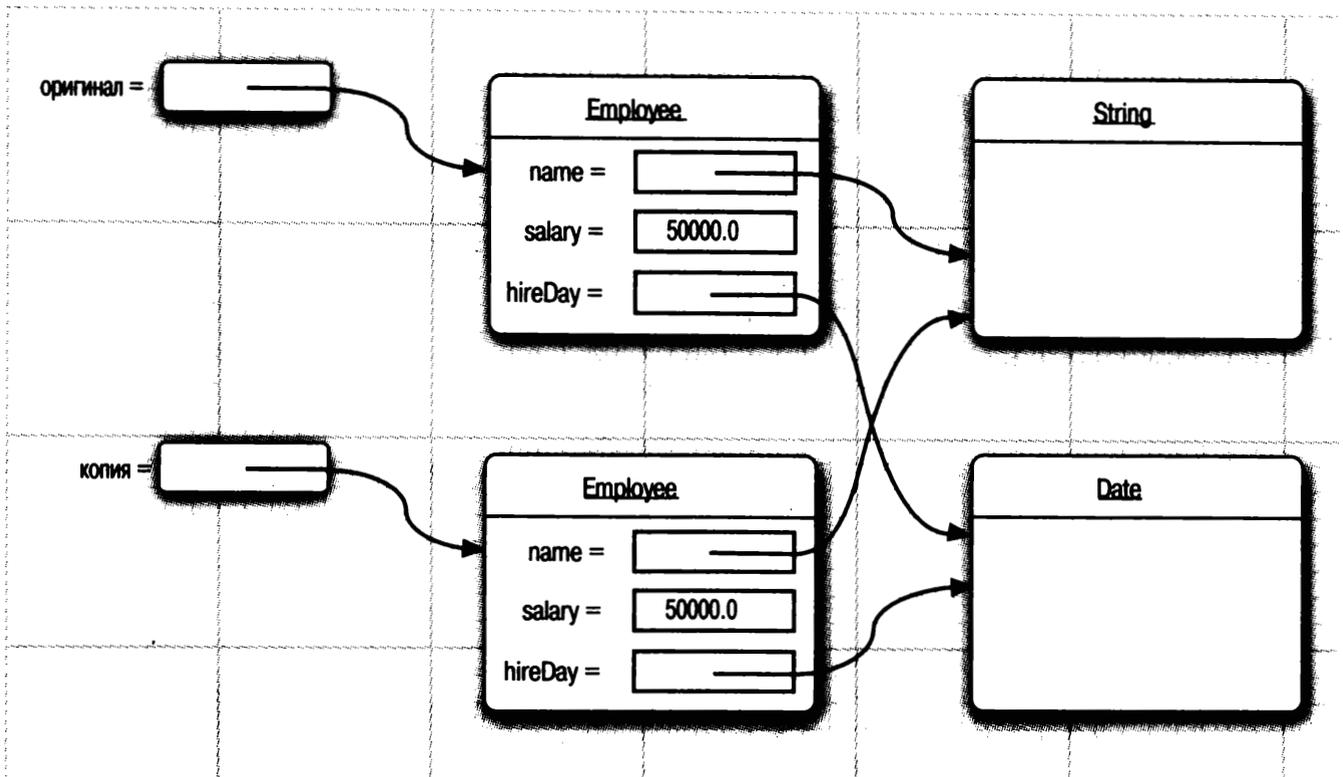


Рис. 6.2. Неполное копирование

Для каждого класса нужно принять следующие решения.

1. Достаточно ли метода `clone()`, предоставляемого по умолчанию?
2. Можно ли доработать предоставляемый по умолчанию метод `clone()` таким образом, чтобы вызывать его для изменяемых объектов?
3. Следует ли вообще отказаться от применения метода `clone()`?

По умолчанию принимается последнее решение. А для принятия первого и второго решений класс должен удовлетворять двум требованиям.

1. Реализация интерфейса `Cloneable`.
2. Переопределение метода `clone()` с модификатором доступа `public`.



НА ЗАМЕТКУ! Метод `clone()` объявлен в классе `Object` как `protected`, поэтому его нельзя просто вызвать по ссылке `anObject.clone()`. Но разве недоступны защищенные методы для любого подкласса, и не является ли каждый класс подклассом класса `Object`? К счастью, правила защищенного доступа не такие строгие (см. главу 5). Подкласс может вызвать защищенный метод `clone()` только для клонирования своих собственных объектов. Чтобы клонировать другие объекты, метод `clone()` следует переопределить как открытый и разрешить клонирование объектов любым другим методом.

В данном случае интерфейс `Cloneable` используется не совсем обычным образом. В частности, метод `clone()` не объявляется в нем, а наследуется от класса `Object`. Интерфейс служит меткой, указывающей на то, что в данном случае разработчик класса понимает, как выполняется процесс клонирования. В Java наблюдается настолько настороженное отношение к клонированию объектов, что если объект требует

выполнения данной операции, но не реализует интерфейс Cloneable, то генерируется исключение.



НА ЗАМЕТКУ! Интерфейс Cloneable — один из немногих помеченных интерфейсов в Java, иногда еще называемых *маркерными интерфейсами*. Напомним, что назначение обычных интерфейсов вроде Comparable — обеспечить реализацию в некотором классе конкретного метода или ряда методов, объявленных в данном интерфейсе. У помеченных интерфейсов отсутствуют методы, а их единственное назначение — разрешить выполнение операции instanceof для проверки типа следующим образом:

```
if (obj instanceof Cloneable) ...
```

Но пользоваться помеченными интерфейсами в прикладных программах не рекомендуется.

Даже если реализация метода clone() по умолчанию (неполное копирование) вполне подходит, все равно нужно реализовать также интерфейс Cloneable, переопределить метод clone() как открытый и сделать вызов super.clone(), как показано в следующем примере кода:

```
class Employee implements Cloneable
{
    // сделать метод открытым, изменить возвращаемый тип
    public Employee clone() throws CloneNotSupportedException
    {
        return (Employee) super.clone();
    }
    . . .
}
```



НА ЗАМЕТКУ! До появления версии Java SE 5.0 метод clone() всегда возвращал объект типа Object. Поддерживаемые теперь ковариантные возвращаемые типы позволяют указывать в методе clone() правильный тип возвращаемого значения.

Рассмотренный выше метод clone() не добавляет никаких новых функциональных возможностей к методу Object.clone(), реализующему неполное копирование. Чтобы реализовать полное копирование, придется приложить дополнительные усилия и организовать клонирование изменяемых полей экземпляра. Ниже приведен пример реализации метода clone(), выполняющего полное копирование.

```
class Employee implements Cloneable
{
    . . .
    public Employee clone() throws CloneNotSupportedException
    {
        // вызвать метод Object.clone()
        Employee cloned = (Employee) super.clone();
        // клонировать изменяемые поля
        cloned.hireDay = (Date) hireDay.clone();

        return cloned;
    }
}
```

Метод clone() из класса Object может генерировать исключение типа CloneNotSupportedException. Это происходит в том случае, если метод clone()

вызывается для объекта, не реализующего интерфейс `Cloneable`. Но поскольку классы `Employee` и `Date` реализуют этот интерфейс, то исключение не генерируется. Впрочем, компилятору об этом ничего неизвестно, и поэтому о возможном исключении приходится объявлять следующим образом:

```
public Employee clone() throws CloneNotSupportedException
```

Не лучше ли было бы вместо этого предусмотреть обработку исключения, как показано ниже?

```
public Employee clone()
{
    try
    {
        return super.clone();
    }
    catch (CloneNotSupportedException e) { return null; }
    // Этого не произойдет, так как данный класс реализует интерфейс Cloneable
}
```

Такое решение вполне подходит для конечных классов, объявляемых как `final`. А в остальном уместнее прибегнуть к помощи ключевого слова `throws`. В этом случае у подкласса останется возможность сгенерировать исключение типа `CloneNotSupportedException`, если он не в состоянии поддерживать клонирование.

Клонирова объекты подклассов, следует соблюдать особую осторожность. Так, если вы определите метод `clone()` в классе `Employee`, другие смогут воспользоваться им для клонирования объектов типа `Manager`. Сможет ли метод `clone()` из класса `Employee` справиться с подобной задачей? Это зависит от набора полей, объявленных в классе `Manager`. В данном случае никаких затруднений не возникнет, поскольку поле `bonus` относится к примитивному типу. Но ведь в класс `Manager` может быть введено поле, требующее полного копирования или вообще не допускающее клонирование. Нет никакой гарантии, что в подклассе реализован метод `clone()`, корректно решающий поставленную задачу. Именно поэтому метод `clone()` объявлен в классе `Object` как `protected`. Но если вы хотите, чтобы пользователи ваших классов могли вызывать метод `clone()`, то подобная "роскошь" остается для вас недоступной.

Следует ли реализовывать метод `clone()` в своих классах? Если пользователям требуется полное копирование, то ответ, конечно, должен быть положительным. Некоторые специалисты считают, что от метода `clone()` нужно вообще отказаться и реализовать вместо него другой метод, решающий аналогичную задачу. Мы согласны с тем, что метод `clone()` — не совсем удачное решение, но если вы передадите его функции другому методу, то столкнетесь с теми же трудностями. Как бы то ни было, клонирование применяется нечасто. Достаточно сказать, что метод `clone()` реализован менее чем в 5 процентах классов из стандартной библиотеки.

В программе, исходный код которой приведен в листинге 6.3, сначала клонируются объекты класса `Employee` (из листинга 6.4), затем вызываются два модифицирующих метода. Метод `raiseSalary()` изменяет значение в поле `salary`, а метод `setHireDay()` — состояние в поле `hireDay`. Ни один из модифицирующих методов не воздействует на исходный объект, поскольку метод `clone()` переопределен и осуществляет полное копирование.

Листинг 6.3. Исходный код из файла clone/CloneTest.java

```
1 package clone;
2
3 /**
4  * В этой программе демонстрируется клонирование
5  * @version 1.10 2002-07-01
6  * @author Cay Horstmann
7  */
8 public class CloneTest
9 {
10     public static void main(String[] args)
11     {
12         try
13         {
14             Employee original = new Employee("John Q. Public", 50000);
15             original.setHireDay(2000, 1, 1);
16             Employee copy = original.clone();
17             copy.raiseSalary(10);
18             copy.setHireDay(2002, 12, 31);
19             System.out.println("original=" + original);
20             System.out.println("copy=" + copy);
21         }
22         catch (CloneNotSupportedException e)
23         {
24             e.printStackTrace();
25         }
26     }
27 }
```

Листинг 6.4. Исходный код из файла clone/Employee.java

```
1 package clone;
2
3 import java.util.Date;
4 import java.util.GregorianCalendar;
5 public class Employee implements Cloneable
6 {
7     private String name;
8     private double salary;
9     private Date hireDay;
10
11     public Employee(String n, double s)
12     {
13         name = n;
14         salary = s;
15         hireDay = new Date();
16     }
17
18     public Employee clone() throws CloneNotSupportedException
19     {
20         // вызвать метод Object.clone()
21         Employee cloned = (Employee) super.clone();
22
23         // клонировать изменяемые поля
24         cloned.hireDay = (Date) hireDay.clone();
25     }
26 }
```

```
25
26     return cloned;
27 }
28
29 /**
30  * Устанавливает заданную дату приема на работу
31  * @param year Год приема на работу
32  * @param month Месяц приема на работу
33  * @param day День приема на работу
34  */
35 public void setHireDay(int year, int month, int day)
36 {
37     Date newHireDay = new GregorianCalendar(year, month - 1, day).getTime();
38
39     // Пример изменения поля экземпляра
40     hireDay.setTime(newHireDay.getTime());
41 }
42
43 public void raiseSalary(double byPercent)
44 {
45     double raise = salary * byPercent / 100;
46     salary += raise;
47 }
48
49 public String toString()
50 {
51     return "Employee[name=" + name + ",salary=" +
52         salary + ",hireDay=" + hireDay + "]";
53 }
54 }
```



НА ЗАМЕТКУ! У всех видов массивов имеется открытый, а не защищенный метод `clone()`. Им можно воспользоваться для создания нового массива, содержащего копии всех элементов, как в следующем примере кода:

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
int[] cloned = (int[]) luckyNumbers.clone();
cloned[5] = 12; // не изменяет элемент массива luckyNumbers[5]
```



НА ЗАМЕТКУ! В главе 1 второго тома данной книги представлен альтернативный механизм клонирования объектов с помощью средства сериализации объектов в Java. Этот механизм прост в реализации и безопасен, хотя и не очень эффективен.

Интерфейсы и обратные вызовы

Механизм *обратного вызова* широко применяется в программировании. При обратном вызове задаются действия, которые должны выполняться всякий раз, когда происходит некоторое событие. Например, можно задать действие, которое должно быть выполнено после щелчка на некоторой кнопке или при выборе определенного пункта меню. Но поскольку мы еще не обсуждали создание пользовательских интерфейсов, то рассмотрим похожую, но более простую ситуацию.

В пакет `javax.swing` входит класс `Timer`, который можно использовать для отсчета интервалов времени. Так, если в программе предусмотрены часы, то с помощью класса `Timer` можно отсчитывать каждую секунду и обновлять циферблат часов. При установке таймера задается интервал времени и указывается, что именно должно произойти по его истечении.

Как же указать таймеру, что именно он должен делать? Во многих языках программирования для этого задается имя функции, которую таймер должен периодически вызывать. Но в классах из стандартной библиотеки Java применяется объектно-ориентированный подход: таймеру нужно передать объект некоторого класса. После этого таймер вызывает один из методов для данного объекта. Передача объекта — более гибкий механизм, чем вызов функции, поскольку объект может нести дополнительную информацию.

Разумеется, таймер должен знать, какой именно метод следует вызвать. Для этого таймеру нужно указать объект класса, реализующего интерфейс `ActionListener` из пакета `java.awt.event`. Этот интерфейс объявляется следующим образом:

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

По истечении заданного интервала времени таймер вызывает метод `actionPerformed()`.

С НА ЗАМЕТКУ C++! Как пояснялось в главе 5, в Java имеются аналоги указателей на функции — объекты класса `Method`. Но их трудно использовать, они действуют медленнее и не могут быть проверены на типовую безопасность на стадии компиляции. Всякий раз, когда в C++ используется указатель на функцию, в Java для этой цели следует применять интерфейс.

Допустим, что через каждые десять секунд требуется выводить на экран сообщение о текущем времени, сопровождаемое звуковым сигналом. Для этого нужно сначала определить класс, реализующий интерфейс `ActionListener`, а затем ввести операторы, которые должны быть выполнены, в тело метода `actionPerformed()`:

```
class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        Date now = new Date();
        System.out.println("At the tone, the time is " + now);
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Обратите внимание на параметр `ActionEvent` метода `actionPerformed()`. Он содержит сведения о событии, например, об инициировавшем его объекте (более подробно этот вопрос рассматривается в главе 8). В данном примере подробные сведения о событии не важны, поэтому параметром `ActionEvent` можно просто пренебречь.

Затем нужно сконструировать объект данного класса и передать его конструктору класса `Timer` следующим образом:

```
ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener);
```

Первый параметр конструктора `Timer` представляет собой измеряемый в миллисекундах интервал времени между последовательными уведомлениями. Сообщение о текущем времени должно выводиться каждые десять секунд. А второй параметр является объектом класса `ActionListener`.

И наконец, таймер запускается следующим образом:

```
t.start();
```

Каждые десять секунд на экране будет появляться следующее сообщение, сопровождаемое звуковым сигналом:

```
At the tone, the time is Thu Apr 13 23:29:08 PDT 2000
```

В листинге 6.5 приведен исходный код программы, реализующей описанный алгоритм. После запуска таймера эта программа выводит на экран окно с сообщением и ожидает до тех пор, пока пользователь не щелкнет на кнопке ОК, чтобы завершить работу. Между тем каждые десять секунд на экран выводится текущее время.

Будьте терпеливы. После того как откроется диалоговое окно с запросом "Quit program?" (Завершить программу), первое сообщение таймера появится на экране только через десять секунд.

Следует заметить, что, помимо пакетов `javax.swing.*` и `java.util.*`, в данной программе импортируется также класс `javax.swing.Timer`. Обратите внимание на то, что при импортировании пакета явно указано имя класса. Это позволяет избежать неоднозначности. Дело в том, что существует класс `java.util.Timer`, который не имеет никакого отношения к решаемой здесь задаче.

Листинг 6.5. Исходный код из файла `timer/TimerTest.java`

```
1 package timer;
2 /**
3  @version 1.00 2000-04-13
4  @author Cay Horstmann
5  */
6
7 import java.awt.*;
8 import java.awt.event.*;
9 import java.util.*;
10 import javax.swing.*;
11 import javax.swing.Timer;
12 // имя этого класса специально указано при импорте пакета
13 // во избежание конфликта с классом java.util.Timer
14
15 public class TimerTest
16 {
17     public static void main(String[] args)
18     {
19         ActionListener listener = new TimePrinter();
20
21         // сконструировать таймер, вызывающий обработчик событий
22         // через каждые 10 секунд
23         Timer t = new Timer(10000, listener);
24         t.start();
25
26         JOptionPane.showMessageDialog(null, "Quit program?");
27         System.exit(0);
28     }
```

```
29 }
30
31 class TimePrinter implements ActionListener
32 {
33     public void actionPerformed(ActionEvent event)
34     {
35         Date now = new Date();
36         System.out.println("At the tone, the time is " + now);
37         Toolkit.getDefaultToolkit().beep();
38     }
39 }
```

javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message)`
Выводит на экран диалоговое окно со строкой сообщения и кнопкой ОК. Диалоговое окно выравнивается по центру компонента `parent`. Если параметр `parent` имеет пустое значение `null`, окно выравнивается по центру экрана.

javax.swing.Timer 1.2

- `Timer(int interval, ActionListener listener)`
Создает таймер, передающий сообщение приемнику событий `listener` по истечении интервала времени, продолжительность которого задается параметром `interval`.
- `void start()`
Запускает таймер, который затем вызывает метод `actionPerformed()` для приемников его событий.
- `void stop()`
Останавливает таймер. После остановки таймера метод `actionPerformed()` для приемников его событий больше не вызывается.

java.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`
Возвращает набор инструментальных средств, используемых по умолчанию. Этот набор содержит сведения о среде ГПИ.
- `void beep()`
Воспроизводит звуковой сигнал.

Внутренние классы

Внутренним называется один класс, определенный в другом классе. А зачем он вообще нужен? На то имеются следующие причины.

- Объект внутреннего класса имеет доступ к данным объекта, в котором он определен, включая закрытые данные.
- Внутренний класс можно скрыть от других классов того же пакета.
- *Анонимный* внутренний класс оказывается удобным в тех случаях, когда требуется определить обратный вызов в процессе выполнения программы, не прибегая к необходимости писать много кода.

В этом разделе довольно сложная тема внутренних классов будет обсуждаться в следующем порядке.

1. Сначала будет представлен простой внутренний класс, способный обращаться к полям экземпляра внешнего класса.
2. Затем будут обсуждаться специальные синтаксические правила, применяемые при объявлении внутренних классов.
3. Далее речь пойдет о преобразовании внутренних классов в обычные. Слабонервные читатели могут пропустить этот материал.
4. После этого рассматриваются *локальные внутренние* классы, способные обращаться к локальным переменным в области действия объемлющего класса.
5. Затем будет введено понятие *анонимного внутреннего* класса и показано, как пользоваться такими классами при организации обратных вызовов.
6. И наконец, будет показано, как можно пользоваться *статическими внутренними* классами для формирования вложенных вспомогательных классов.



НА ЗАМЕТКУ C++! В C++ имеются *вложенные классы*. Вложенный класс находится в области действия объемлющего класса. Ниже приведен типичный пример: в классе для связанного списка определен класс, содержащий связи, а также класс, в котором определяется позиция итератора.

```
class LinkedList
{
public:
    class Iterator // вложенный класс
    {
    public:
        void insert(int x);
        int erase();
        . . .
    };
    . . .
private:
    class Link // вложенный класс
    {
    public:
        Link* next;
        int data;
    };
    . . .
};
```

Вложение представляет собой отношение между классами, а не между объектами. Объект класса `LinkedList` не содержит подобъекты типа `Iterator` или `Link`.

У вложения классов имеются следующие два преимущества: возможность *управления именами* и *управления доступом*. Имя `Iterator` вложено в класс `LinkedList`, и поэтому оно известно исключительно как `LinkedList::Iterator` и не может конфликтовать с именами `Iterator` других классов. В Java это преимущество не играет такой роли, поскольку в этом языке подобное управление именами осуществляют *пакеты*. Следует заметить, что класс `Link` находится в *закрытом* разделе класса `LinkedList`. Он полностью скрыт от остальной части программы. Именно по этой причине его поля можно объявлять открытыми, и тогда методы из класса `LinkedList` получают к ним доступ на вполне законных основаниях, а для остальных методов эти поля останутся невидимыми. В Java такой вид управления доступом был невозможен до тех пор, пока не были внедрены внутренние классы.

Но у внутренних классов в Java имеется еще одно преимущество, которое делает их более полезными, чем вложенные классы C++. Объект внутреннего класса содержит неявную ссылку на объект того внешнего класса, который создал его. С помощью этой ссылки объект внутреннего класса получает доступ ко всему состоянию внешнего объекта. Данный механизм более подробно рассматривается далее в этой главе. Такая дополнительная ссылка отсутствует только у статических внутренних классов в Java. Именно они являются полным аналогом вложенных классов в C++.

Доступ к состоянию объекта с помощью внутреннего класса

Синтаксис, применяемый для внутренних классов, довольно сложен. Поэтому, для того чтобы продемонстрировать применение внутренних классов, рассмотрим простой, хотя надуманный в какой-то степени пример. Итак, реорганизуем класс `TimerTest` из рассмотренного ранее примера, чтобы сформировать из него класс `TalkingClock`. Для организации работы "говорящих часов" применяются два параметра: интервал между последовательными сообщениями и признак, позволяющий включать или отключать звуковой сигнал. Соответствующий код приведен ниже.

```
public class TalkingClock
{
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep) { . . . }
    public void start() { . . . }

    public class TimePrinter implements ActionListener
    // внутренний класс
    {
        . . .
    }
}
```

Обратите внимание на то, что класс `TimePrinter` теперь расположен в классе `TalkingClock`. Это не означает, что каждый экземпляр класса `TalkingClock` содержит поле типа `TimePrinter`. Как станет ясно в дальнейшем, объекты типа `TimePrinter` создаются методами из класса `TalkingClock`. Рассмотрим класс `TimePrinter` более подробно. Обратите внимание в приведенном ниже коде на то, что в методе `actionPerformed()` проверяется признак `beep` перед тем, как воспроизвести звуковой сигнал.

```
public class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
```

```

{
    Date now = new Date();
    System.out.println("At the tone, the time is " + now);
    if (beep) Toolkit.getDefaultToolkit().beep();
}
}

```

Теперь начинается самое интересное. Нетрудно заметить, что в классе `TimePrinter` отсутствует поле `beep`. Вместо этого метод `actionPerformed()` обращается к соответствующему полю объекта типа `TalkingClock`. А это уже нечто новое. Обычно метод обращается к полям объекта. Но оказывается, что внутренний класс имеет доступ не только к своим полям, но и к полям создавшего его объекта, т.е. экземпляра внешнего класса. Для того чтобы это стало возможным, внутренний класс должен содержать ссылку на объект внешнего класса, как показано на рис. 6.3.

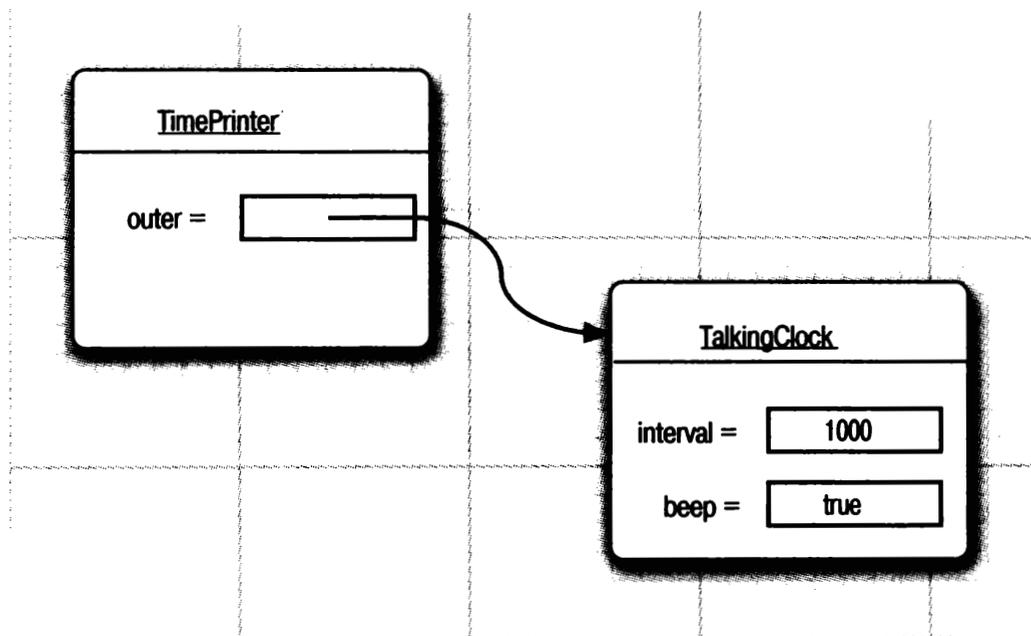


Рис. 6.3. Объект внутреннего класса содержит ссылку на объект внешнего класса

В определении внутреннего класса эта ссылка не присутствует явно. Для того чтобы продемонстрировать, каким образом она действует, введем в код ссылку `outer`. Тогда метод `actionPerformed()` будет выглядеть следующим образом:

```

public void actionPerformed(ActionEvent event)
{
    Date now = new Date();
    System.out.println("At the tone, the time is " + now);
    if (outer.beep) Toolkit.getDefaultToolkit().beep();
}

```

Ссылка на объект внешнего класса задается в конструкторе. Компилятор видоизменяет все конструкторы внутреннего класса, добавляя параметр для ссылки на внешний класс. А поскольку конструкторы в классе `TalkingClock` не определены, то компилятор автоматически формирует конструктор без аргументов, генерируя код, подобный следующему:

```

public TimePrinter(TalkingClock clock) // автоматически генерируемый код
{
    outer = clock;
}

```

Еще раз обращаем ваше внимание на то, что слово *outer* не является ключевым в Java. Оно используется только для иллюстрации механизма, задействованного во внутренних классах.

После того как метод `start()` создаст объект класса `TimePrinter`, компилятор передаст конструктору текущего объекта ссылку `this` на объект типа `TalkingClock` следующим образом:

```
ActionListener listener = new TimePrinter(this); // параметр добавляется
                                                // автоматически
```

В листинге 6.6 приведен исходный код завершеного варианта программы, проверяющей внутренний класс. Если бы `TimePrinter` был обычным классом, он должен был бы получить доступ к признаку `beep` через открытый метод из класса `TalkingClock`. Применение внутреннего класса усовершенствует код, поскольку отпадает необходимость предоставлять специальный метод доступа, представляющий интерес только для какого-нибудь другого класса.



НА ЗАМЕТКУ! Класс `TimePrinter` можно было бы объявить как `private`. И тогда конструировать объекты типа `TimePrinter` могли бы только методы `TalkingClock`. Закрытыми могут быть только внутренние классы. А обычные классы всегда доступны в пределах пакета или же открыты полностью.

Листинг 6.6. Исходный код из файла `innerClass/InnerClassTest.java`

```
1 package innerClass;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.util.*;
6 import javax.swing.*;
7 import javax.swing.Timer;
8 /**
9  * В этой программе демонстрируется применение внутренних классов
10 * @version 1.10 2004-02-27
11 * @author Cay Horstmann
12 */
13 public class InnerClassTest
14 {
15     public static void main(String[] args)
16     {
17         TalkingClock clock = new TalkingClock(1000, true);
18         clock.start();
19
20         // выполнять программу до тех пор, пока пользователь
21         // не щелкнет на кнопке OK
22         JOptionPane.showMessageDialog(null, "Quit program?");
23         System.exit(0);
24     }
25 }
26
27 /**
28 * Часы, выводящие время через регулярные промежутки
29 */
30 class TalkingClock
```

```
31 {
32     private int interval;
33     private boolean beep;
34
35     /**
36      * Конструирует "говорящие часы"
37      * @param interval Интервал между сообщениями (в миллисекундах)
38      * @param beep Истинно, если часы должны издавать звуковой сигнал
39      */
40     public TalkingClock(int interval, boolean beep)
41     {
42         this.interval = interval;
43         this.beep = beep;
44     }
45
46     /**
47      * Запускает часы
48      */
49     public void start()
50     {
51         ActionListener listener = new TimePrinter();
52         Timer t = new Timer(interval, listener);
53         t.start();
54     }
55     public class TimePrinter implements ActionListener
56     {
57         public void actionPerformed(ActionEvent event)
58         {
59             Date now = new Date();
60             System.out.println("At the tone, the time is " + now);
61             if (beep) Toolkit.getDefaultToolkit().beep();
62         }
63     }
64 }
```

Специальные синтаксические правила для внутренних классов

В предыдущем разделе ссылка на внешний класс была названа *outer* для того, чтобы стало понятнее, что это ссылка из внутреннего класса на внешний. На самом деле синтаксис для внешних ссылок немного сложнее. Так, приведенное ниже выражение обозначает внешнюю ссылку.

```
ВнешнийКласс.this
```

Например, во внутреннем классе `TimePrinter` можно создать метод `actionPerformed()` следующим образом:

```
public void actionPerformed(ActionEvent event)
{
    . . .
    if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

С другой стороны, конструктор внутреннего класса можно записать более явным образом, используя следующий синтаксис:

```
ОбъектВнешнегоКласса.new ВнутреннийКласс(параметры)
```

Например, в приведенной ниже строке кода ссылка на внешний класс из вновь созданного объекта типа `TimePrinter` получает ссылку `this` на метод, создающий объект внутреннего класса.

```
ActionListener listener = this.new TimePrinter();
```

Такой способ применяется чаще всего, хотя явное указание ссылки `this` здесь, как всегда, излишне. Тем не менее это позволяет явно указать другой объект в ссылке на объект внешнего класса. Так, если класс `TimePrinter` является открытым внутренним классом, его объекты можно создать для любых “говорящих часов”, как показано ниже.

```
TalkingClock jabberer = new TalkingClock(1000, true);  
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

Следует иметь в виду, что если ссылка на внутренний класс делается за пределами области действия внешнего класса, то она указывается следующим образом:

```
ВнешнийКласс.ВнутреннийКласс
```

О пользе, необходимости и безопасности внутренних классов

Внутренние классы были впервые внедрены в версии Java 1.1. Многие программисты встретили появление внутренних классов настороженно, считая их отступлением от принципов, положенных в основу Java. По их мнению, главным преимуществом языка Java над C++ является его простота. Внутренние классы действительно сложны. (Такой вывод вы, скорее всего, сделаете, ознакомившись с анонимными внутренними классами, которые будут рассматриваться далее в этой главе.) Взаимодействие внутренних классов с другими языковыми средствами не совсем очевидно, и особенно это касается вопросов управления доступом и безопасности.

Зачем же создатели языка Java пожертвовали преимуществами, которыми он выгодно отличался от других языков программирования, в пользу изящного и, безусловно, интересного механизма, выгода от которого, впрочем, сомнительна? Не пытаясь дать исчерпывающий ответ на этот вопрос, заметим только, что обращение с внутренними классами происходит на уровне *компилятора*, а не виртуальной машины. Для их обозначения используется знак `$`, разделяющий имена внешних и внутренних классов. Таким образом, для виртуальной машины внутренние классы неотличимы от внешних.

Например, класс `TimePrinter`, входящий в класс `TalkingClock`, преобразуется в файл `TalkingClock$TimePrinter.class`. Для того чтобы посмотреть, каким образом действует этот механизм, попробуйте провести следующий эксперимент: запустите на выполнение программу `ReflectionTest` (см. главу 5) и выполните рефлекссию класса `TalkingClock$TimePrinter`. С другой стороны, можно воспользоваться утилитой `javap` следующим образом:

```
javap -private ИмяКласса
```



НА ЗАМЕТКУ! Если вы пользуетесь UNIX, не забудьте экранировать знак `$`, указывая имя класса в командной строке. Следовательно, запускайте программу `ReflectionTest` или утилиту `javap` одним из двух следующих способов:

```
java reflection.ReflectionTest ВнутреннийКласс.TalkingClock\$TimePrinter
```

или

```
javap -private ВнутреннийКласс.TalkingClock\$TimePrinter
```

В итоге будет получен следующий результат:

```
public class TalkingClock$TimePrinter
{
    public TalkingClock$TimePrinter(TalkingClock);

    public void actionPerformed(java.awt.event.ActionEvent);

    final TalkingClock this$0;
}
```

Нетрудно заметить, что компилятор генерирует дополнительное поле `this$0` для ссылки на внешний класс. (Имя `this$0` синтезируется компилятором, поэтому сослаться на него нельзя.) Кроме того, у конструктора можно обнаружить параметр `TalkingClock`. Если компилятор автоматически выполняет преобразования, нельзя ли реализовать подобный механизм вручную? Попробуем сделать это, превратив `TimePrinter` в обычный класс, определяемый за пределами класса `TalkingClock`, а затем передав ему ссылку `this` на создавший его объект, как показано ниже.

```
class TalkingClock
{
    . . .
    public void start()
    {
        ActionListener listener = new TimePrinter(this);
        Timer t = new Timer(interval, listener);
        t.start();
    }
}

class TimePrinter implements ActionListener
{
    private TalkingClock outer;
    . . .
    public TimePrinter(TalkingClock clock)
    {
        outer = clock;
    }
}
```

Рассмотрим теперь метод `actionPerformed()`. Ему требуется доступ к полю `outer.beep` следующим образом:

```
if (outer.beep) . . . // ОШИБКА!
```

И здесь возникает ошибка. Внутренний класс может иметь доступ к закрытым данным лишь того внешнего класса, в который он входит. Но ведь класс `TimePrinter` уже не является внутренним, а следовательно, не имеет такого доступа.

Следовательно, внутренние классы, по существу, намного эффективнее, чем обычные, поскольку они обладают более высокими правами доступа. В связи с этим возникает следующий вопрос: каким образом внутренние классы получают дополнительные права доступа, если они преобразуются в обычные, а виртуальной машине вообще о них ничего не известно? Чтобы раскрыть эту тайну, воспользуемся еще раз программой `ReflectionTest`, отслеживающей поведение класса `TalkingClock`, получив следующий результат:

```
class TalkingClock
{
    private int interval;

    private boolean beep;

    public TalkingClock(int, boolean);

    static boolean access$0(TalkingClock);
    public void start();
}
```

Обратите внимание на статический метод `access$0`, добавленный компилятором во внешний класс. Этот метод возвращает значение из поля `beep` объекта, переданного ему в качестве параметра. (Имя этого метода может отличаться в зависимости от компилятора, например `access$000`.)

Этот метод вызывается из внутреннего класса. В методе `actionPerformed()` из класса `TimePrinter` имеется следующий оператор:

```
if (beep)
```

Он преобразуется компилятором в приведенный ниже вызов.

```
if (access$100(outer));
```

Не опасно ли это? В принципе опасно! Посторонний может легко вызвать метод `access$0()` и прочесть данные из закрытого поля `beep`. Конечно, `access$0` не является допустимым именем для метода в Java. Но злоумышленники, знакомые со структурой файлов классов, легко могут создать свой аналогичный файл и вызвать данный метод с помощью соответствующих команд виртуальной машины. Разумеется, такой файл должен формироваться вручную (например, с помощью редактора шестнадцатеричного кода). Но поскольку область действия секретных методов доступа ограничена пакетом, атакующий код должен размещаться в том же самом пакете, что и атакуемый класс.

Итак, если внутренний класс имеет доступ к закрытым полям, можно создать другой класс, добавить его в тот же пакет и получить доступ к закрытым данным. Правда, для этого требуется опыт и решительность. Программист не может получить такой доступ случайно, не создавая для этих целей специальный файл, содержащий видоизмененные классы.



НА ЗАМЕТКУ! Синтезированные методы и конструкторы могут быть довольно запутанными (материал этой врезки не для слабонервных, так что можете его пропустить). Допустим, `TimePrinter` превращен в закрытый внутренний класс. Но для виртуальной машины не существует внутренних классов, поэтому компилятор делает все возможное, чтобы произвести доступный в пределах пакета класс с закрытым конструктором следующим образом:

```
private TalkingClock$TimePrinter(TalkingClock);
```

Конечно, такой конструктор никто не сможет вызвать. Поэтому требуется второй конструктор, доступный в пределах пакета и вызывающий первый, как показано ниже.

```
TalkingClock$TimePrinter(TalkingClock, TalkingClock$1);
```

Компилятор преобразует вызов конструктора в методе `start()` из класса `TalkingClock` следующим образом:

```
new TalkingClock$TimePrinter(this, null)
```

Локальные внутренние классы

Если внимательно проанализировать исходный код класса TalkingClock, то можно обнаружить, что имя класса TimePrinter используется лишь однажды: при создании объекта данного типа в методе start(). В подобных случаях класс можно определить локально в отдельном методе, как выделено ниже полужирным.

```
public void start()
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            Date now = new Date();
            System.out.println("At the tone, the time is " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }

    ActionListener listener = new TimePrinter();
    Timer t = new Timer(interval, listener);
    t.start();
}
```

Локальные внутренние классы никогда не объявляются с помощью модификаторов доступа (например, public и protected). Их область действия всегда ограничивается блоком, в котором они объявлены. У локальных внутренних классов имеется большое преимущество: они полностью скрыты от внешнего кода и даже от остальной части класса TalkingClock. Ни одному из методов, за исключением start(), ничего неизвестно о классе TimePrinter.

Доступ к конечным переменным из внешних методов

Локальные внутренние классы выгодно отличаются от обычных внутренних классов еще и тем, что имеют доступ не только к полям своего внешнего класса, но и к локальным переменным! Но такие локальные переменные должны быть объявлены как final. Обратимся к характерному примеру, переместив параметры interval и beep из конструктора TalkingClock в метод start(), как выделено ниже полужирным.

```
public void start(int interval, final boolean beep)
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            Date now = new Date();
            System.out.println("At the tone, the time is " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }

    ActionListener listener = new TimePrinter();
    Timer t = new Timer(interval, listener);
    t.start();
}
```

Обратите внимание на то, что в классе `TimePrinter` больше не нужно хранить переменную экземпляра `beep`. Он просто ссылается на параметр метода, содержащего определение данного класса. Возможно, это не так уж и неожиданно. В конце концов, приведенная ниже строка кода находится в теле метода `start()`, так почему бы не иметь в ней доступ к переменной `beep`?

```
if (beep) . . .
```

Локальные внутренние классы обладают рядом особенностей. Чтобы понять их, рассмотрим подробнее логику их управления.

1. Вызывается метод `start()`.
2. При вызове конструктора внутреннего класса `TimePrinter` инициализируется объектная переменная `listener`.
3. Передается ссылка `listener` конструктору класса `Timer`, запускается таймер и метод `start()` прекращает свою работу. В этот момент параметр `beep` метода `start()` больше не существует.
4. Некоторое время спустя выполняется оператор `if (beep)...` в методе `actionPerformed()`.

Для того чтобы метод `actionPerformed()` выполнялся успешно, в классе `TimePrinter` должна быть создана копия поля `beep` до того, как оно перестанет существовать в качестве локальной переменной метода `start()`. Именно это и происходит. В данном примере компилятор синтезирует для локального внутреннего класса имя `TalkingClock$1TimePrinter`. Если применить снова программу `ReflectionTest` для анализа класса `TalkingClock$1TimePrinter`, то получим следующий результат:

```
class TalkingClock$1TimePrinter
{
    TalkingClock$1TimePrinter(TalkingClock, boolean);

    public void actionPerformed(java.awt.event.ActionEvent);

    final boolean val$beep;
    final TalkingClock this$0;
}
```

Обратите внимание на параметр конструктора, имеющий тип `boolean`, а также переменную экземпляра `val$beep`. При создании объекта переменная `beep` передается конструктору и размещается в поле `val$beep`. Для того чтобы это стало возможным, разработчикам компилятора пришлось немало потрудиться. Компилятор должен обнаруживать доступ к локальным переменным, создавать для каждой из них соответствующие поля, а затем копировать локальные переменные в конструкторе так, чтобы поля данных инициализировались копиями локальных переменных.

С точки зрения разработчика прикладных программ доступ к локальным переменным выглядит привлекательно. Благодаря такой возможности вложенные классы становятся проще и уменьшается количество полей, которые должны программироваться явным образом.

Как упоминалось выше, методы локального класса могут ссылаться только на локальные переменные, объявленные как `final`. По этой причине параметр `beep` в рассматриваемом здесь примере был объявлен конечным (т.е. `final`). Конечная

локальная переменная не может быть видоизменена. Этим гарантируется, что локальная переменная и ее копия, созданная в локальном классе, всегда имеют одно и то же значение.



НА ЗАМЕТКУ! Как было показано ранее, конечные переменные можно использовать в качестве констант следующим образом:

```
public static final double SPEED_LIMIT = 55;
```

Ключевое слово `final` можно применять и при объявлении локальных, статических переменных, а также переменных экземпляра. В любом случае оно обозначает одно и то же: значение присваивается данной переменной только один раз: сразу после ее создания. Изменить это значение впоследствии нельзя.

Но инициализировать конечную переменную одновременно с ее определением совсем не обязательно. Например, конечный параметр `beep` инициализируется сразу после его создания, когда вызывается метод `start()`. (Если этот метод вызывается неоднократно, то при каждом его вызове будет создаваться новый конечный параметр `beep`.) Значение переменной экземпляра `val$beep` в классе `TalkingClock$1TimePrinter` устанавливается лишь один раз: в конструкторе внутреннего класса. Конечную переменную, которая не была инициализирована при объявлении, часто называют *пустой конечной переменной*.

Ограничение `final` не совсем удобно. Допустим, требуется обновить счетчик в объемлющей области действия, чтобы подсчитать, насколько часто во время сортировки вызывается метод `compareTo()`:

```
int counter = 0;
Date[] dates = new Date[100];
for (int i = 0; i < dates.length; i++)
    dates[i] = new Date()
    {
        public int compareTo(Date other)
        {
            counter++; // ОШИБКА!
            return super.compareTo(other);
        }
    };
Arrays.sort(dates);
System.out.println(counter + " comparisons.");
```

Объявить переменную `counter` как `final` нельзя. Ведь совершенно очевидно, что ее придется обновлять. Ее нельзя заменить и на `Integer`, поскольку объекты типа `Integer` неизменяемы. В качестве выхода из этого положения можно воспользоваться массивом длиной в 1 элемент, как выделено ниже полужирным.

```
final int[] counter = new int[1];
for (int i = 0; i < dates.length; i++)
    dates[i] = new Date()
    {
        public int compareTo(Date other)
        {
            counter[0]++;
            return super.compareTo(other);
        }
    };
```

(Переменная типа массива также объявлена как `final`. Но это просто означает, что ее нельзя заставить ссылаться на другой массив. А элементы массива можно свободно изменять.)

Обнаруживая впервые внутренний класс, компилятор автоматически выполняет подобное преобразование для всех локальных переменных, видоизменяемых во внутреннем классе. Но некоторых напугала способность компилятора создавать без их ведома объекты в “куче”, и поэтому было наложено ограничение `final`. Возможно, в будущих версиях Java это решение будет пересмотрено.

Анонимные внутренние классы

Работая с локальными внутренними классами, можно воспользоваться еще одной интересной возможностью. Так, если требуется создать единственный объект некоторого класса, этому классу можно вообще не присваивать имени. Такой класс называется *анонимным внутренним классом*, как показано ниже.

```
public void start(int interval, final boolean beep)
{
    ActionListener listener = new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            Date now = new Date();
            System.out.println("At the tone, the time is " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    };
    Timer t = new Timer(interval, listener);
    t.start();
}
```

Следует признать, что синтаксис анонимных внутренних классов довольно сложен. На самом деле приведенный выше фрагмент кода означает следующее: создается новый объект класса, реализующего интерфейс `ActionListener`, где в фигурных скобках `{}` определен требующийся метод `actionPerformed()`. Ниже приведена общая форма определения анонимных внутренних классов.

```
new СуперТип(параметры конструирования объектов)
{
    методы и данные внутреннего класса
}
```

Здесь *СуперТип* может быть интерфейсом, например `ActionListener`, и тогда внутренний класс реализует данный интерфейс. *СуперТип* может быть также классом. В этом случае внутренний класс расширяет данный суперкласс.

Анонимный внутренний класс не может иметь конструкторов, поскольку имя конструктора должно совпадать с именем класса, а в данном случае у класса отсутствует имя. Вместо этого параметры, необходимые для создания объекта, передаются конструктору суперкласса. Так, если вложенный класс реализует какой-нибудь интерфейс, параметры конструктора можно не указывать. Тем не менее они должны быть указаны в круглых скобках следующим образом:

```
new ТипИнтерфейса()
{
    методы и данные
}
```

Следует внимательно и аккуратно проводить различие между созданием нового объекта некоторого класса и конструированием объекта анонимного внутреннего класса, расширяющего данный класс. Если за скобками со списком параметров, необходимых для создания объекта, следует открытая фигурная скобка, то определяется анонимный вложенный класс, как показано ниже.

```
Person queen = new Person("Mary");
// объект типа Person
Person count = new Person("Dracula") { . . . };
// объект внутреннего класса, расширяющего класс Person
```

Было ли внедрение анонимных внутренних классов в Java удачной идеей, или это только изящное средство создавать никому не понятные программы? Скорее всего, и то и другое вместе. Если внутренний класс невелик и сводится к нескольким строкам простого кода, можно, конечно, сберечь время на набор исходного кода программы, но сэкономленное в итоге время обернется впоследствии трудностями при чтении ее исходного кода.

В листинге 6.7 приведен исходный код завершенной версии программы, реализующей “говорящие часы”, где применяется анонимный внутренний класс. Сравнив эту версию программы с ее версией из листинга 6.6, вы можете сами убедиться, что применение анонимного внутреннего класса сделало программу немного короче, но не проще для понимания, хотя для этого требуются опыт и практика.

Листинг 6.7. Исходный код из файла anonymousInnerClass/AnonymousInnerClassTest.java

```
1 package anonymousInnerClass;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.util.*;
6 import javax.swing.*;
7 import javax.swing.Timer;
8
9 /**
10  * В этой программе демонстрируется применение анонимных внутренних классов
11  * @version 1.10 2004-02-27
12  * @author Cay Horstmann
13  */
14 public class AnonymousInnerClassTest
15 {
16     public static void main(String[] args)
17     {
18         TalkingClock clock = new TalkingClock();
19         clock.start(1000, true);
20
21         // выполнять программу до тех пор, пока пользователь
22         // не щелкнет на кнопке OK
23         JOptionPane.showMessageDialog(null, "Quit program?");
24         System.exit(0);
25     }
26 }
27
28 /**
29  * Часы, выводящие время через регулярные промежутки
30  */
31 class TalkingClock
32 {
```

```
33 /**
34  * Запускает часы
35  * @param interval Интервал между сообщениями (в миллисекундах)
36  * @param beep Истинно, если часы должны издавать звуковой сигнал
37  */
38 public void start(int interval, final boolean beep)
39 {
40     ActionListener listener = new ActionListener()
41     {
42         public void actionPerformed(ActionEvent event)
43         {
44             Date now = new Date();
45             System.out.println("At the tone, the time is " + now);
46             if (beep) Toolkit.getDefaultToolkit().beep();
47         }
48     };
49     Timer t = new Timer(interval, listener);
50     t.start();
51 }
52 }
```



НА ЗАМЕТКУ! Существует специальный прием, называемый *инициализацией в двойных фигурных скобках* и выгодно использующий преимущества синтаксиса внутренних классов. Допустим, требуется составить списочный массив и передать ему метод следующим образом:

```
ArrayList<String> friends = new ArrayList<>();
favorites.add("Harry");
favorites.add("Tony");
invite(friends);
```

Если списочный массив больше не понадобится, то было бы неплохо сделать его анонимным. Но как тогда вводить в него дополнительные элементы? А вот как:

```
invite(new ArrayList<String>() {{ add("Harry"); add("Tony"); }})
```

Обратите внимание на двойные фигурные скобки в приведенной выше строке кода. Внешние фигурные скобки образуют анонимный подкласс `ArrayList`, а внутренние фигурные скобки — блок конструирования объектов (см. главу 4).



ВНИМАНИЕ! Зачастую анонимный подкласс удобно сделать почти, но не совсем таким же, как и его суперкласс. Но в этом случае следует соблюдать особую осторожность в отношении метода `equals()`. Как рекомендовалось в главе 5, в методе `equals()` необходимо организовать следующую проверку:

```
if (getClass() != other.getClass()) return false;
```

Но анонимный подкласс ее не пройдет.



СОВЕТ. При выдаче регистрирующих или отладочных сообщений в них нередко требуется включить имя текущего класса, как в приведенной ниже строке кода.

```
System.err.println("Something awful happened in " + getClass());
```

Но такой прием не годится для статического метода. Ведь вызов метода `getClass()`, по существу, означает вызов `this.getClass()`. Но ссылка `this` на текущий объект для статического метода не годится. В таком случае можно воспользоваться следующим выражением:

```
new Object(){}.getClass().getEnclosingClass() // получить класс
                                                // статического метода
```

Здесь выражение `new Object(){}` создает объект анонимного подкласса, производного от класса `Object`, а метод `getEnclosingClass()` получает объемлющий его класс, т.е. класс, содержащий статический метод.

Статические внутренние классы

Иногда внутренний класс требуется лишь для того, чтобы скрыть его внутри другого класса, тогда как ссылка на объект внешнего класса не нужна. Подавить формирование такой ссылки можно, объявив внутренний класс статическим (т.е. `static`).

Допустим, в массиве требуется найти максимальное и минимальное число. Конечно, для этого можно было бы написать два метода: один — для нахождения максимального числа, а другой — для нахождения минимального числа. Но при вызове обоих методов массив просматривается дважды. Было бы намного эффективнее просматривать массив только один раз, одновременно определяя в нем как максимальное, так и минимальное число следующим образом:

```
double min = Double.MAX_VALUE;
double max = Double.MIN_VALUE;
for (double v : values)
{
    if (min > v) min = v;
    if (max < v) max = v;
}
```

Но в таком случае метод должен возвращать два значения. Сделать это можно, определив класс `Pair` с двумя полями для хранения числовых значений, как показано ниже.

```
class Pair
{
    private double first;
    private double second;

    public Pair(double f, double s)
    {
        first = f;
        second = s;
    }

    public double getFirst() { return first; }
    public double getSecond() { return second; }
}
```

Тогда метод `minmax()` сможет вернуть объект типа `Pair` следующим образом:

```
class ArrayAlg
{
    public static Pair minmax(double[] values)
    {
        . . .
        return new Pair(min, max);
    }
}
```

Таким образом, для получения максимального и минимального чисел достаточно вызвать методы `getFirst()` и `getSecond()`, как показано ниже.

```
Pair p = ArrayAlg.minmax(d);
System.out.println("min = " + p.getFirst());
System.out.println("max = " + p.getSecond());
```

Разумеется, имя `Pair` слишком широко распространено, и при выполнении крупного проекта у другого программиста может возникнуть такая же блестящая идея, вот только класс `Pair` у него будет содержать не числовые, а строковые поля. Это вполне вероятное затруднение можно разрешить, сделав класс `Pair` внутренним и определенным в классе `ArrayAlg`. Тогда подлинным именем этого класса будет не `Pair`, а `ArrayAlg.Pair`:

```
ArrayAlg.Pair p = ArrayAlg.minmax(d);
```

Но, в отличие от внутренних классов, применявшихся в предыдущих примерах, ссылка на другой объект в классе `Pair` не требуется. Ее можно подавить, объявив внутренний класс статическим:

```
class ArrayAlg
{
    public static class Pair
    {
        . . .
    }
    . . .
}
```

Разумеется, только внутренние классы можно объявлять статическими. Статический внутренний класс ничем не отличается от любого другого внутреннего класса, за исключением того, что его объект не содержит ссылку на создавший его объект внешнего класса. В данном примере следует применять статический внутренний класс, поскольку объект внутреннего класса создается в теле статического метода, как показано ниже.

```
public static Pair minmax(double[] d)
{
    . . .
    return new Pair(min, max);
}
```

Если бы класс `Pair` не был объявлен статическим, компилятор сообщил бы, что при инициализации объекта внутреннего класса объект типа `ArrayAlg` недоступен.



НА ЗАМЕТКУ! Статический вложенный класс применяется в тех случаях, когда доступ к объекту внутреннего класса не требуется. Некоторые программисты для обозначения статических внутренних классов пользуются термином *вложенные классы*.



НА ЗАМЕТКУ! Внутренние классы, определенные в интерфейсах, автоматически считаются статическими и открытыми (т.е. `static` и `public`).

В листинге 6.8 приведен весь исходный код класса `ArrayAlg` и вложенного в него класса `Pair`.

Листинг 6.8. Исходный код из файла `staticInnerClass/StaticInnerClassTest.java`

```
1 package staticInnerClass;
2
3 /**
4  * В этой программе демонстрируется применение
5  * статического внутреннего класса
6  * @version 1.01 2004-02-27
7  * @author Cay Horstmann
8  */
9 public class StaticInnerClassTest
10 {
11     public static void main(String[] args)
12     {
13         double[] d = new double[20];
14         for (int i = 0; i < d.length; i++)
15             d[i] = 100 * Math.random();
16         ArrayAlg.Pair p = ArrayAlg.minmax(d);
17         System.out.println("min = " + p.getFirst());
18         System.out.println("max = " + p.getSecond());
19     }
20 }
21 class ArrayAlg
22 {
23     /**
24     * Пара чисел с плавающей точкой
25     */
26     public static class Pair
27     {
28         private double first;
29         private double second;
30
31         /**
32         * Составляет пару из двух чисел с плавающей точкой
33         * @param f Первое число
34         * @param s Второе число
35         */
36         public Pair(double f, double s)
37         {
38             first = f;
39             second = s;
40         }
41
42         /**
43         * Возвращает первое число из пары
44         * @return Возврат первого числа
45         */
46         public double getFirst()
47         {
48             return first;
49         }
50
51         /**
52         * Возвращает второе число из пары
53         * @return Возврат второго числа
54         */
55         public double getSecond()
56         {
```

```
57     return second;
58   }
59 }
60
61 /**
62  * Определяет минимальное и максимальное числа в массиве
63  * @param values Массив чисел с плавающей точкой
64  * @return Пара, первым элементом которой является минимальное
65  *         число, а вторым элементом – максимальное число
66  */
67 public static Pair minmax(double[] values)
68 {
69     double min = Double.MAX_VALUE;
70     double max = Double.MIN_VALUE;
71     for (double v : values)
72     {
73         if (min > v) min = v;
74         if (max < v) max = v;
75     }
76     return new Pair(min, max);
77 }
78 }
```

Прокси-классы

В последнем разделе этой главы мы обсудим понятие *прокси-классов*, которые зачастую называют *классами-посредниками*. Они предназначены для того, чтобы создавать во время выполнения программы новые классы, реализующие заданные интерфейсы. Прокси-классы требуются, если на стадии компиляции еще неизвестно, какие именно интерфейсы следует реализовать. В прикладном программировании такая ситуация возникает крайне редко. Но в некоторых приложениях системного программирования гибкость, обеспечиваемая прокси-классами, может оказаться весьма уместной.

Допустим, требуется сконструировать объект класса, реализующего один или несколько интерфейсов, конкретные характеристики которых во время компиляции неизвестны. Допустим также, что требуется создать объект класса, реализующего эти интерфейсы. Сделать это не так-то просто. Для построения конкретного класса достаточно воспользоваться методом `newInstance()` из класса `Class` или механизмом рефлексии, чтобы найти конструктор этого класса. Но создать объект интерфейса нельзя. Следовательно, определить новый класс во время выполнения не удастся.

В качестве выхода из этого затруднительного положения в некоторых программах генерируется код, размещаемый в файле, вызывается компилятор, а затем полученный файл класса. Естественно, что это очень медленный процесс, который к тому же требует развертывания компилятора вместе с программой. Механизм прокси-классов предлагает более изящное решение. Прокси-класс может создавать во время выполнения совершенно новые классы и реализует те интерфейсы, которые указывает программист. В частности, в прокси-классе содержатся следующие методы.

- Все методы, которые требуют указанные интерфейсы.
- Все методы, определенные в классе `Object` (в том числе `toString()`, `equals()` и т.д.).

Но определить новый код для этих методов в ходе выполнения программы нельзя. Вместо этого программист должен предоставить *обработчик вызовов*, т.е. объект, реализующий интерфейс `InvocationHandler`. В этом интерфейсе единственный метод объявляется следующим образом:

```
Object invoke(Object proxy, Method method, Object[] args)
```

При вызове какого-либо метода для прокси-объекта автоматически вызывается метод `invoke()` обработчика вызовов, получающий объект класса `Method` и параметры исходного вызова. Обработчик вызовов должен быть в состоянии обработать вызов. Для создания прокси-объекта служит метод `newProxyInstance()` из класса `Proxy`. Этот метод получает следующие три параметра.

- *Загрузчик классов*. Модель безопасности в Java позволяет использовать загрузчики разных классов: системных классов, загружаемых из Интернета и т.д. Загрузчики классов обсуждаются в главе 9 второго тома данной книги. А до тех пор в приведенных далее примерах кода будет указываться пустое значение `null`, чтобы использовать загрузчик классов, предусмотренный по умолчанию.
- Массив объектов типа `Class` — по одному на каждый реализуемый интерфейс.
- Обработчик вызовов.

Остается решить еще два вопроса: как определить обработчик и что можно сделать с полученным в итоге прокси-объектом? Разумеется, ответы на эти вопросы зависят от конкретной задачи, которую требуется решить с помощью механизма прокси-объектов. В частности, их можно применять для достижения следующих целей.

- Переадресация вызовов методов на удаленный сервер.
- Связывание событий, происходящих в пользовательском интерфейсе, с определенными действиями, выполняемыми в программе.
- Отслеживание вызовов методов при отладке.

В рассматриваемом здесь примере программы прокси-объекты и обработчики вызовов применяются для отслеживания обращений к методам. С этой целью определяется приведенный ниже класс `TraceHandler`, заключающий некоторый объект в оболочку. Его метод `invoke()` лишь выводит на экран имя и параметры того метода, к которому выполнялось обращение, а затем вызывает сам метод, задавая в качестве неявного параметра объект, заключенный в оболочку.

```
class TraceHandler implements InvocationHandler
{
    private Object target;
    public TraceHandler(Object t)
    {
        target = t;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable
    {
        // вывести метод и его параметры
        . . .
        // вызвать конкретный метод
    }
}
```

```
        return m.invoke(target, args);
    }
}
```

Ниже показано, каким образом создается прокси-объект, позволяющий отслеживать вызов одного из его методов.

```
Object value = . . . ;
// сконструировать оболочку
InvocationHandler handler = new TraceHandler(value);
// сконструировать прокси-объект для одного или нескольких интерфейсов
Class[] interfaces = new Class[] { Comparable.class};
Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
```

Теперь каждый раз, когда метод вызывается для объекта `proxy`, выводятся его имя и параметры, а затем происходит обращение к соответствующему методу объекта `value`.

В программе, приведенной в листинге 6.9, прокси-объекты служат для отслеживания результатов двоичного поиска. Сначала заполняется массив, состоящий из прокси-объектов для целых чисел от **1** до **1000**. Затем для поиска случайного целого числа в массиве вызывается метод `binarySearch()` из класса `Arrays`. И наконец, на экран выводится элемент, совпадающий с критерием поиска, как показано ниже.

```
Object[] elements = new Object[1000];
// заполнить элементы прокси-объектами для целых чисел от 1 до 1000
for (int i = 0; i < elements.length; i++)
{
    Integer value = i + 1;
    elements[i] = Proxy.newProxyInstance(. . . ); // прокси-объект для
                                                    // конкретного значения
}

// сформировать случайное целое число
Integer key = new Random().nextInt(elements.length) + 1;

// выполнить поиск по критерию key
int result = Arrays.binarySearch(elements, key);

// вывести совпавший элемент, если таковой найден
if (result >= 0) System.out.println(elements[result]);
```

Класс `Integer` реализует интерфейс `Comparable`. Прокси-объекты принадлежат классу, определяемому во время выполнения. (Его имя выглядит как `$Proxy0`.) Этот класс также реализует интерфейс `Comparable`. Но в его методе `compareTo()` вызывается метод `invoke()` из обработчика вызовов прокси-объекта.



НА ЗАМЕТКУ! Как отмечалось в начале этой главы, в классе `Integer` фактически реализован интерфейс `Comparable<Integer>`. Но во время выполнения данные об обобщенных типах удаляются, а при создании прокси-объекта используется объект базового типа `Comparable`.

Метод `binarySearch()` осуществляет вызов, аналогичный следующему:

```
if (elements[i].compareTo(key) < 0) ...
```

Массив заполнен прокси-объектами, и поэтому из метода `compareTo()` вызывается метод `invoke()` из класса `TraceHandler`. В этом методе выводится имя вызванного

метода и его параметры, а затем вызывается метод `compareTo()` для заключенного в оболочку объекта типа `Integer`.

В конце программы из рассматриваемого здесь примера выводятся результаты ее работы, для чего служит следующая строка кода:

```
System.out.println(elements[result]);
```

Из метода `println()` вызывается метод `toString()` для прокси-объекта, а затем этот вызов также переадресуется обработчику вызовов. Ниже приведены результаты полной трассировки при выполнении программы.

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
288.compareTo(288)
288.toString()
```

Как видите, при двоичном поиске на каждом шаге интервал уменьшается вдвое. Обратите внимание на то, что и метод `toString()` представлен прокси-объектом, несмотря на то, что он не относится к интерфейсу `Comparable`. Как станет ясно в дальнейшем, некоторые методы из класса `Object` всегда представлены прокси-объектами.

Листинг 6.9. Исходный код из файла `proxy/ProxyTest.java`

```
1 package proxy;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5 /**
6  * В этой программе демонстрируется применение прокси-объектов
7  * @version 1.00 2000-04-13
8  * @author Cay Horstmann
9  */
10 public class ProxyTest
11 {
12     public static void main(String[] args)
13     {
14         Object[] elements = new Object[1000];
15
16         // заполнить массив elements прокси-объектами целых числе от 1 до 1000
17         for (int i = 0; i < elements.length; i++)
18         {
19             Integer value = i + 1;
20             InvocationHandler handler = new TraceHandler(value);
21             Object proxy = Proxy.newProxyInstance(null,
22                 new Class[] { Comparable.class }, handler);
23             elements[i] = proxy;
24         }
25
26         // сформировать случайное целое число
27         Integer key = new Random().nextInt(elements.length) + 1;
28
29         // выполнить поиск по критерию key
30         int result = Arrays.binarySearch(elements, key);
31
```

```
32 // вывести совпавший элемент, если таковой найден
33 if (result >= 0) System.out.println(elements[result]);
34 }
35 }
36
37 /**
38  * Обработчик вызовов, выводящий сначала имя метода и его параметры,
39  * а затем вызывающий исходный метод
40  */
41 class TraceHandler implements InvocationHandler
42 {
43     private Object target;
44
45     /**
46      * Конструирует объекты типа TraceHandler
47      * @param t неявный параметр вызова метода
48      */
49     public TraceHandler(Object t)
50     {
51         target = t;
52     }
53
54     public Object invoke(Object proxy, Method m, Object[] args)
55         throws Throwable
56     {
57         // вывести неявный параметр
58         System.out.print(target);
59         // вывести имя метода
60         System.out.print("." + m.getName() + "(");
61
62         // вывести явные параметры
63         if (args != null)
64         {
65             for (int i = 0; i < args.length; i++)
66             {
67                 System.out.print(args[i]);
68                 if (i < args.length - 1) System.out.print(", ");
69             }
70         }
71         System.out.println(")");
72
73         // вызвать конкретный метод
74         return m.invoke(target, args);
75     }
76 }
```

Свойства прокси-классов

Итак, показав прокси-классы в действии, вернемся к анализу некоторых их свойств. Напомним, что прокси-классы создаются во время выполнения. Но затем они становятся обычными классами, как и все остальные классы, обрабатываемые виртуальной машиной.

Все прокси-классы расширяют класс `Proxy`. Такой класс содержит только одну переменную экземпляра, которая ссылается на обработчик вызовов, определенный в суперклассе `Proxy`. Любые дополнительные данные, необходимые для выполнения задач, решаемых прокси-объектами, должны храниться в обработчике вызовов.

Например, в программе из листинга 6.9 при создании прокси-объектов, представляющих интерфейс `Comparable`, класс `TraceHandler` служит оболочкой для конкретных объектов.

Во всех прокси-классах переопределяются методы `toString()`, `equals()` и `hashCode()` из класса `Object`. Эти методы лишь вызывают метод `invoke()` для обработчика вызовов. Другие методы из класса `Object` (например, `clone()` и `getClass()`) не переопределяются. Имена прокси-классов не определены. В виртуальной машине формируются имена классов, начинающиеся со строки `$Proxy`.

Для конкретного загрузчика классов и заданного набора интерфейсов может существовать только один прокси-класс. Это означает, что, если дважды вызвать метод `newProxyInstance()` для одного и того же загрузчика классов и массива интерфейсов, будут получены два объекта одного и того же класса. Имя этого класса можно определить с помощью метода `getProxyClass()` следующим образом:

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

Прокси-класс всегда является открытым и конечным. Если все интерфейсы, которые реализуются прокси-классом, объявлены как `public`, этот класс не принадлежит ни одному конкретному пакету. В противном случае все интерфейсы, в объявлении которых не указан модификатор доступа `public`, а следовательно, и сам прокси-класс, должны принадлежать одному пакету. Вызвав метод `isProxyClass()` из класса `Proxy`, можно проверить, представляет ли объект типа `Class` определенный прокси-класс.

`java.lang.reflect.InvocationHandler` 1.3

- `Object invoke(Object proxy, Method method, Object[] args)`

Этот метод определяется для того, чтобы задать действие, которое должно быть выполнено при вызове какого-нибудь метода для прокси-объекта.

`java.lang.reflect.Proxy` 1.3

- `static Class getProxyClass(ClassLoader loader, Class[] interfaces)`
Возвращает прокси-класс, реализующий заданные интерфейсы.
- `static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)`
Создает новый экземпляр прокси-класса, реализующего заданные интерфейсы. Во всех методах вызывается метод `invoke()` для объекта, указанного в качестве обработчика вызовов.
- `static boolean isProxyClass(Class c)`
Возвращает логическое значение `true`, если `c` является прокси-классом.

Этой главой завершается изложение основ языка Java. С понятиями интерфейсов и внутренних классов вам придется встречаться еще не раз. В отличие от них, прокси-классы представляют интерес в основном для разработчиков инструментальных средств, а не прикладных программ. Итак, усвоив основы, вы можете теперь переходить к изучению графических средств и пользовательских интерфейсов, описание которых начинается в главе 7.

Программирование графики

В этой главе...

- ▶ Общие сведения о библиотеке Swing
- ▶ Создание фрейма
- ▶ Расположение фрейма
- ▶ Отображение данных в компоненте
- ▶ Двухмерные формы
- ▶ Окрашивание цветом
- ▶ Специальное шрифтовое оформление текста
- ▶ Вывод изображений

До сих пор было показано, как писать программы, входные данные для которых вводились с клавиатуры. Затем эти данные обрабатывались, а результаты выводились на консоль. Ныне такие программы уже не соответствуют запросам большинства пользователей. Современные программы не работают подобным образом, тем более — веб-страницы. В этой главе рассматриваются вопросы создания графического пользовательского интерфейса (ГПИ). Из нее вы, в частности, узнаете, каким образом можно изменить размеры и расположение окон на экране, отобразить в окне текст, набранный разными шрифтами, вывести на экран рисунок и т.д. Эти навыки окажутся весьма полезными при создании интересных программ, представленных в последующих главах книги.

Две последующие главы будут посвящены обработке событий вроде нажатия клавиши и щелчка кнопкой мыши, а также средствам создания элементов

пользовательского интерфейса, например, меню и кнопок. Проработав материал этой и двух последующих глав, вы овладеете основами разработки графических приложений. Более сложные вопросы программирования графики рассматриваются во втором томе данной книги. Если же программирование серверных приложений на Java интересует вас больше, чем разработка приложений с ГПИ, можете пропустить эти три главы.

Общие сведения о библиотеке Swing

В первую версию Java входила библиотека классов Abstract Window Toolkit (AWT), предоставляющая основные средства программирования ГПИ. Создание элементов ГПИ на конкретной платформе (Windows, Solaris, Macintosh и т.д.) библиотека AWT поручала встроенным инструментальным средствам. Так, если с помощью библиотеки AWT на экран нужно было вывести окно с текстом, то фактически оно отображалось базовыми средствами конкретной платформы. Теоретически созданные таким образом программы должны были работать на любых платформах, имея внешний вид, характерный для целевой платформы, — в соответствии с рыночным девизом компании Sun Microsystems: “Написано однажды, работает везде”.

Методика, основанная на использовании базовых средств конкретных платформ, отлично подходила для простых приложений. Но вскоре стало ясно, что с ее помощью крайне трудно создавать высококачественные переносимые графические библиотеки, зависящие от собственных интерфейсных элементов платформы. Элементы пользовательского интерфейса, например, меню, панели прокрутки и текстовые поля, на разных платформах могут вести себя по-разному. Следовательно, на основе этого подхода трудно создавать согласованные программы с предсказуемым поведением. Кроме того, некоторые графические среды (например, X11/Motif) не имеют такого богатого набора компонентов пользовательского интерфейса, как операционные системы Windows и Macintosh. Это, в свою очередь, делало библиотеки еще более зависимыми. В результате графические приложения, созданные с помощью библиотеки AWT, выглядели по сравнению с программами для Windows или Macintosh не так привлекательно и не имели таких функциональных возможностей. Хуже того, в библиотеке AWT на различных платформах обнаруживались *разные* ошибки. Разработчикам приходилось тестировать каждое приложение на каждой платформе, что на практике означало: “Написано однажды, отлаживается везде”.

В 1996 году компания Netscape создала библиотеку программ для разработки ГПИ, назвав ее IFC (Internet Foundation Classes). Эта библиотека была основана на совершенно других принципах. Элементы пользовательского интерфейса вроде меню, кнопок и тому подобному воспроизводились в пустом окне. А от оконной системы конкретной платформы требовалось лишь отображать окно и рисовать в нем графику. Таким образом, элементы ГПИ, созданные с помощью библиотеки IFC, выглядели и вели себя одинаково, но не зависели от той платформы, на которой запускалась программа. Компании Sun Microsystems и Netscape объединили свои усилия и усовершенствовали данную методику, создав библиотеку под кодовым названием Swing. С тех пор слово Swing стало официальным названием набора инструментальных средств для создания машино-независимого пользовательского интерфейса. Средства Swing стали доступны как расширение Java 1.1 и вошли в состав стандартной версии Java SE 1.2.

С тех пор, как известный джазовый исполнитель Дюк Эллингтон сказал: “It Don't Mean a Thing If It Ain't Got That Swing” (Ах, истина проста: без свинга жизнь пуста), слово Swing стало официальным названием платформенно-независимого набора инструментальных средств для разработки ГПИ. Swing является частью библиотеки классов Java Foundation Classes (JFC). Полный комплект JFC огромен и содержит намного больше, чем набор инструментальных средств Swing для разработки ГПИ. В состав JFC входят не только компоненты Swing, но и прикладные интерфейсы API для специальных возможностей, двумерной графики и перетаскивания.



НА ЗАМЕТКУ! Библиотека Swing не является полной заменой библиотеки AWT. Она построена на основе архитектуры AWT. Библиотека Swing просто дает больше возможностей для создания пользовательского интерфейса. При написании программы средствами Swing, по существу, используются основные ресурсы AWT. Здесь и далее под Swing подразумеваются машино-независимые классы для создания “рисованного” пользовательского интерфейса, а под AWT — базовые средства конкретной платформы для работы с окнами, включая обработку событий.

Разумеется, элементы пользовательского интерфейса из библиотеки Swing появляются на экране немного медленнее, чем аналогичные компоненты из библиотеки AWT. Но, как показывает опыт, на современных компьютерах этого практически незаметно. С другой стороны, у библиотеки Swing имеется ряд весьма существенных преимуществ.

- Содержит более богатый и удобный набор элементов пользовательского интерфейса.
- Намного меньше зависит от той платформы, на которой должна выполняться программа. Следовательно, она меньше подвержена ошибкам, характерным для конкретной платформы.
- Обеспечивает одинаковое восприятие конечными пользователями приложений с ГПИ на разных платформах.

Впрочем, третье достоинство библиотеки Swing может обернуться недостатком: если элементы пользовательского интерфейса на разных платформах выглядят одинаково, то (по крайней мере, на некоторых платформах) они обязательно будут *отличаться* от компонентов, реализуемых в данной системе, т.е. будут хуже восприниматься конечными пользователями.

В библиотеке Swing это затруднение разрешается очень изящно. Разработчики, использующиеся Swing, могут придать своей программе внешний вид в нужном стиле. Так, на рис. 7.1 и 7.2 показано, как выглядит одна и та же программа в средах Windows и GTK.

Более того, компания Sun Microsystems разработала независимый от платформы стиль под названием Metal (Металлический), который сообщество программирующих на Java прозвало “стилем Java”. Но большинство из них продолжают и далее употреблять название Metal. В этой книге мы последуем их примеру.

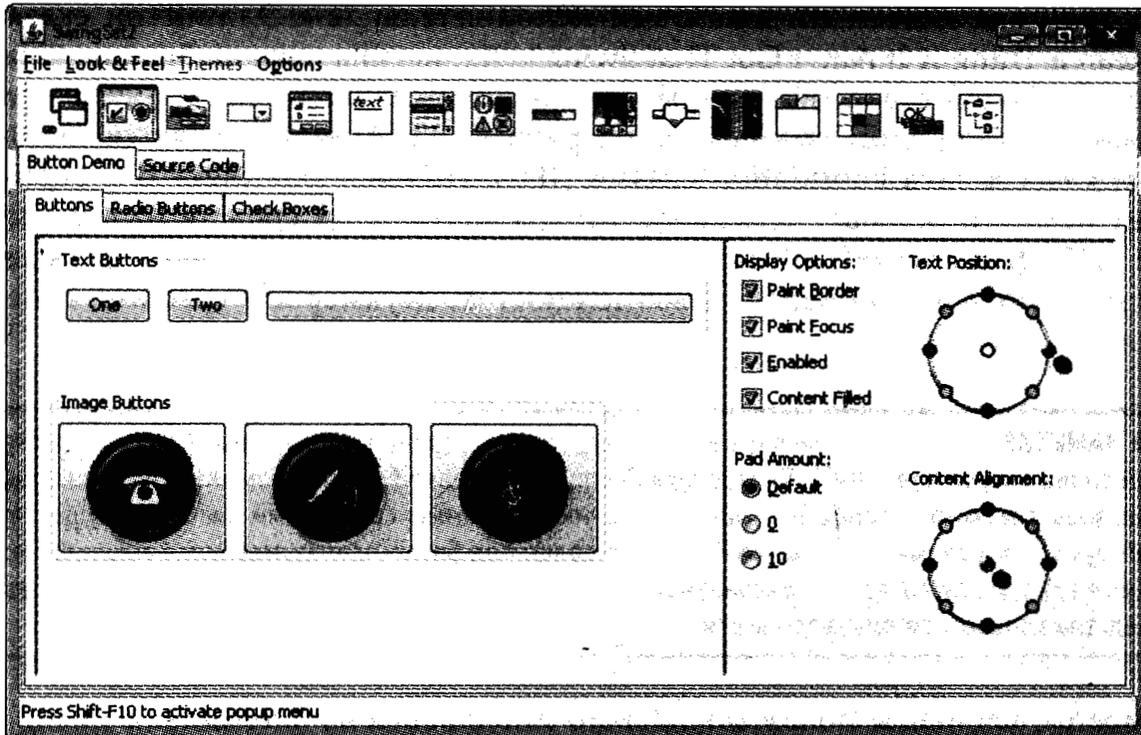


Рис. 7.1. Окно программы в среде Windows

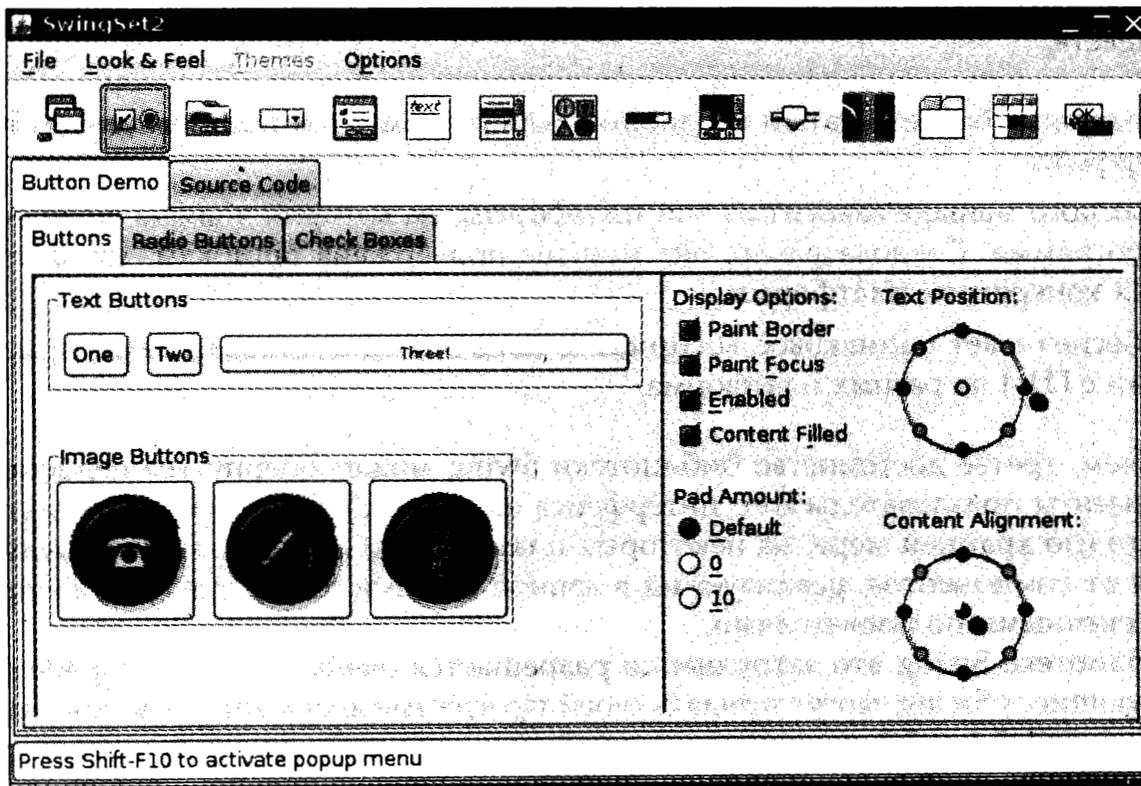


Рис. 7.2. Окно программы в среде GTK

Некоторые специалисты посчитали стиль Metal слишком “тяжелым”. Возможно, поэтому в версии JDK 5.0 он был несколько изменен (рис. 7.3). В настоящее время поддерживается несколько разновидностей стиля Metal, незначительно отличающихся цветами и шрифтами. По умолчанию используется вариант под названием Ocean (Океан). В версии Java SE 6 был усовершенствован собственный стиль Windows и GTK.

Теперь приложения Swing допускают настройку цветовой схемы и верно отображают пульсирующие кнопки и линейки прокрутки, которые стали выглядеть более современно.

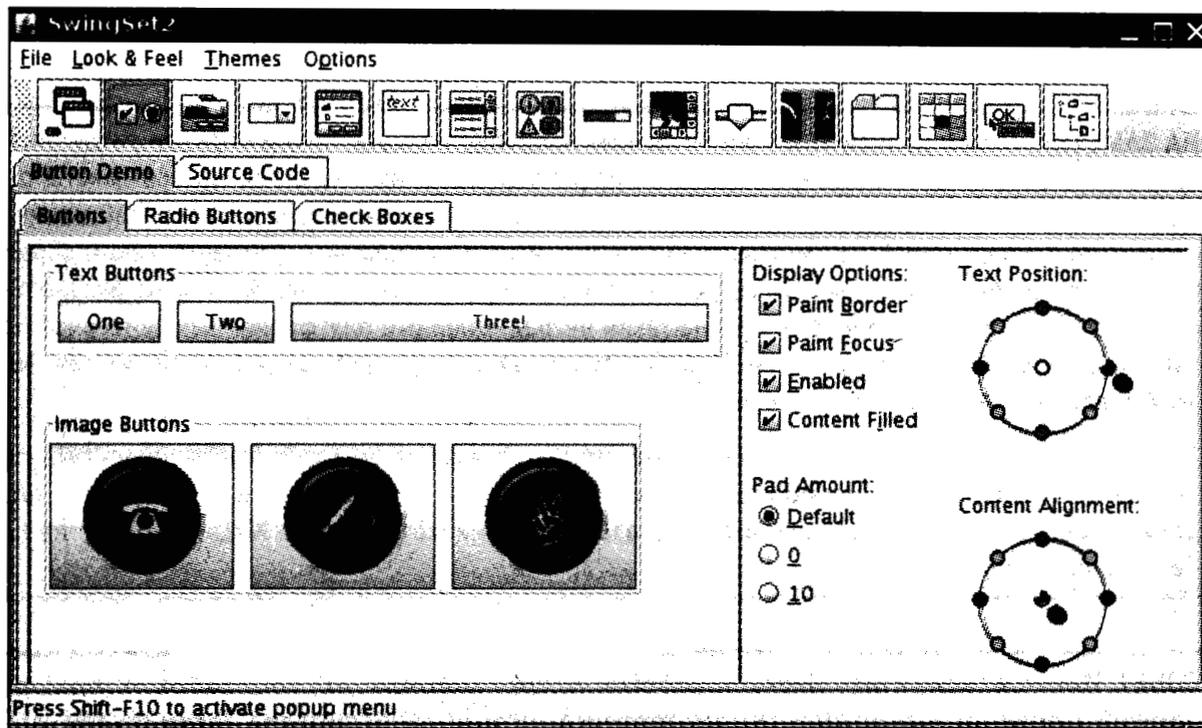


Рис. 7.3. Вид программы в варианте Ocean стиля Metal

В версии 7 предлагается новый, доступный по умолчанию стиль под названием Nimbus (Ореол; рис. 7.4). В этом стиле применяется векторная, а не растровая графика, и поэтому ее качество не зависит от разрешения экрана монитора.

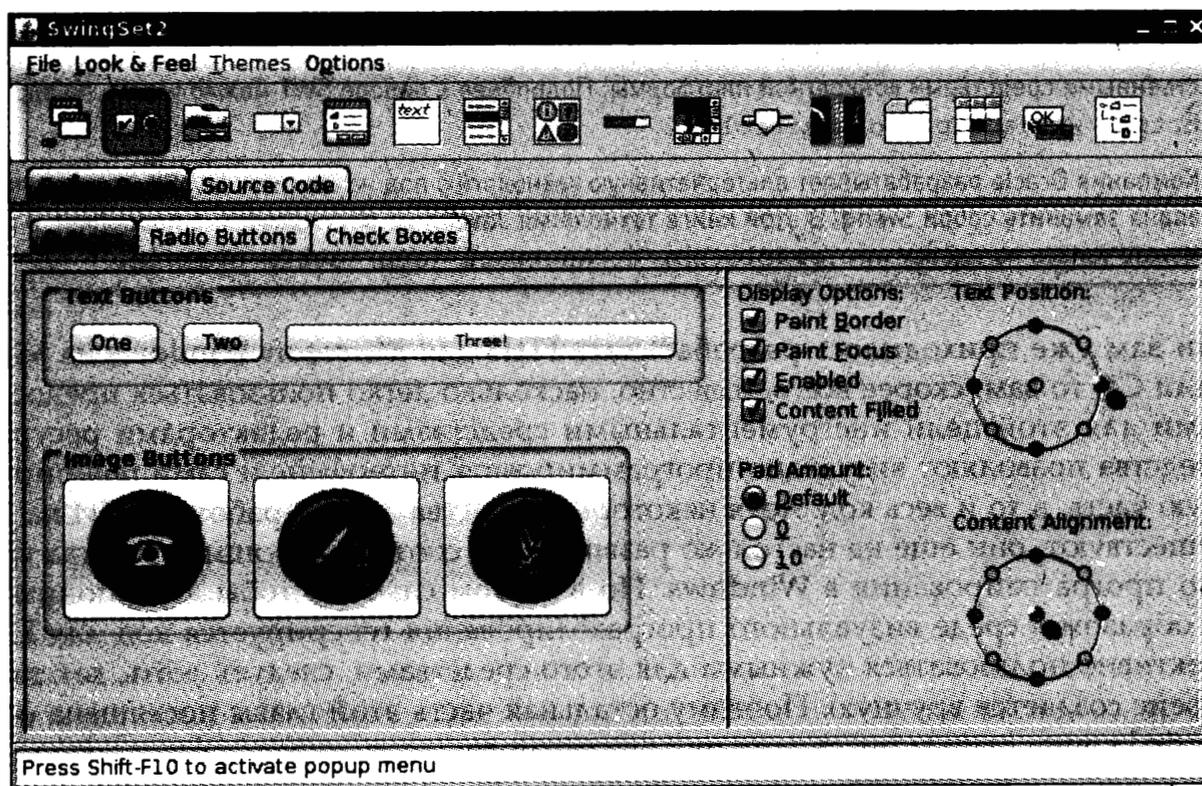


Рис. 7.4. Стиль Nimbus

Одни пользователи предпочитают, чтобы их приложения на Java выглядели и вели себя так, как принято на данной платформе, другие придерживаются стиля Metal или сторонних стилей. Как будет показано в главе 8, пользователям совсем не трудно предоставить возможность для выбора предпочтительного внешнего стиля приложений.



НА ЗАМЕТКУ! Объем данной книги не позволяет подробно описать, каким образом расширяются средства отображения пользовательских интерфейсов. Заметим лишь, что программисты сами могут изменять стили оформления ГПИ своих приложений и даже разрабатывать собственные, совершенно новые стили. Это длительный процесс, в ходе которого нужно указать, каким образом каждый компонент библиотеки Swing должен быть отображен на экране. Некоторые разработчики уже сделали это при переносе Java на такие нетрадиционные платформы, как сенсорные терминалы или карманные устройства. Набор интересных реализаций средств отображения Swing можно найти по адресу <http://www.javatoo.com>.

В версии Java SE 5.0 был реализован новый стиль под названием Synth (Синтетический), упрощающий настройку компонентов под различные платформы. Стиль Synth позволяет также определить новый стиль, для чего достаточно указать файлы изображений и XML-описания, не прибегая к программированию.



СОВЕТ. Имеется также стиль Napkin (Салфетка; <http://napkinlaf.sourceforge.net>), придающий элементам пользовательского интерфейса рисованный от руки вид. Таким стилем удобно пользоваться для демонстрации прототипов приложений заказчиком, ясно давая им понять, что приложение еще не завершено.



НА ЗАМЕТКУ! В настоящее время большинство приложений на Java снабжаются пользовательскими интерфейсами, созданными на основе Swing. Исключением из этого правила являются лишь программы, при разработке которых применялась ИСП Eclipse. В ней вместо Swing применяется набор графических компонентов SWT, где, как и в AWT, компоненты отображаются собственными средствами конкретной платформы. Подробнее о наборе SWT можно узнать по адресу <http://www.eclipse.org/articles/>.

Компания Oracle разрабатывает альтернативную технологию под названием JavaFX, которая призвана заменить собой Swing. В этой книге технология JavaFX не обсуждается, а подробнее о ней можно узнать по адресу www.oracle.com/technetwork/java/javafx/overview.

Если вам уже приходилось разрабатывать ГПИ для Microsoft Windows на Visual Basic или C#, то вам, скорее всего, известно, насколько легко пользоваться предоставляемыми для этой цели инструментальными средствами и редакторами ресурсов. Эти средства позволяют визуально программировать интерфейс и сами генерируют большую часть, а то и весь код. Хотя некоторые средства для разработки ГПИ на Java уже существуют, они еще не настолько развиты, как соответствующие средства визуального программирования в Windows. Но в любом случае, чтобы ясно понимать, каким образом в среде визуального программирования генерируется код для ГПИ, и эффективно пользоваться нужными для этого средствами, следует знать, как такой интерфейс создается вручную. Поэтому остальная часть этой главы посвящена основам отображения окон и рисования их содержимого.

Создание фрейма

Окно верхнего уровня (т.е. такое окно, которое не содержится внутри другого окна) в Java называется *фреймом*. В библиотеке AWT для такого окна предусмотрен класс `Frame`. А в библиотеке Swing аналогом его является класс `JFrame`. Класс `JFrame` расширяет класс `Frame` и представляет собой один из немногих компонентов библиотеки Swing, которые не отображаются на холсте. Кнопки, строка заголовков, пиктограммы и другие элементы оформления окон реализуются с помощью пользовательской оконной системы, а не библиотеки Swing.



ВНИМАНИЕ! Большинство имен компонентов библиотеки Swing начинаются с буквы **J**. В качестве примера можно привести классы `JButton` и `JFrame`. Они являются аналогами соответствующих компонентов библиотеки AWT (например, `Button` и `Frame`). Если пропустить букву **J** в имени применяемого компонента, программа скомпилируется и будет работать, но сочетание компонентов Swing и AWT в одном окне приведет к несогласованности внешнего вида и поведения элементов ГПИ.

В этом разделе будут рассмотрены самые распространенные приемы работы с классом `JFrame`. В листинге 7.1 приведен исходный код простой программы, отображающей на экране пустой фрейм, как показано на рис. 7.5.

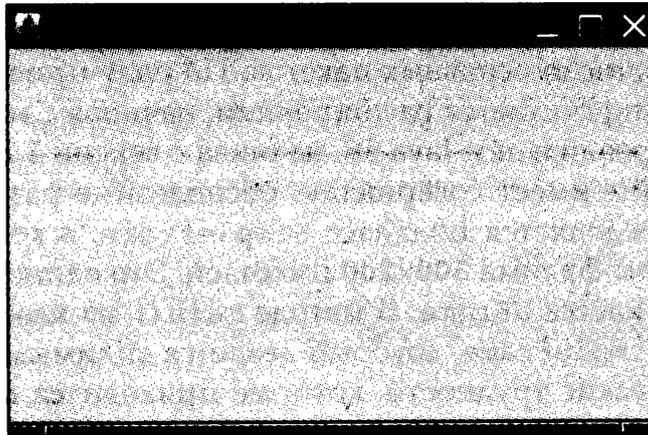


Рис. 7.5. Простейший отображаемый фрейм

Листинг 7.1. Исходный код из файла `simpleframe/SimpleFrameTest.java`

```
1 package simpleFrame;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * @version 1.32 2007-06-12
8  * @author Cay Horstmann
9  */
10 public class SimpleFrameTest
11 {
12     public static void main(String[] args)
```

```
13 {
14     EventQueue.invokeLater(new Runnable()
15     {
16         public void run()
17         {
18             SimpleFrame frame = new SimpleFrame();
19             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20             frame.setVisible(true);
21         }
22     });
23 }
24 }
25
26 class SimpleFrame extends JFrame
27 {
28     private static final int DEFAULT_WIDTH = 300;
29     private static final int DEFAULT_HEIGHT = 200;
30
31     public SimpleFrame()
32     {
33         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34     }
35 }
```

Проанализируем эту программу построчно. Классы библиотеки Swing находятся в пакете `javax.swing`. Имя пакета `javax` означает, что этот пакет является расширением Java и не входит в число основных пакетов. По ряду причин исторического характера библиотека Swing считается расширением, начиная с версии Java 1.1. Но она присутствует в каждой реализации Java SE, начиная с версии 1.2.

По умолчанию фрейм имеет совершенно бесполезные размеры: 0×0 пикселей. В данном примере определяется подкласс `SimpleFrame`, в конструкторе которого устанавливаются размеры фрейма 300×200 пикселей. Это единственное отличие подкласса `SimpleFrame` от класса `JFrame`. В методе `main()` из класса `SimpleFrameTest` создается объект типа `SimpleFrame`, который делается видимым.

Имеются две технические трудности, которые приходится преодолевать каждой Swing-программе. Прежде всего, компоненты Swing должны быть настроены в потоке диспетчеризации событий, т.е. в том потоке управления, который передает компонентам пользовательского интерфейса события вроде щелчков кнопками мыши и нажатий клавиш. В следующем фрагменте кода операторы выполняются в потоке диспетчеризации событий:

```
EventQueue.invokeLater(new Runnable()
{
    public void run()
    {
        операторы
    }
});
```

Подробнее потоки будут обсуждаться в главе 14. А до тех пор вы должны просто принять этот код как своего рода “волшебное заклинание”, с помощью которого запускаются Swing-программы.



НА ЗАМЕТКУ! Вам еще встретится немало Swing-программ, где пользовательский интерфейс не инициализируется в потоке диспетчеризации событий. Раньше инициализацию допускалось выполнять в главном потоке. К сожалению, в связи с усложнением компонентов Swing разработчики JDK не смогли больше гарантировать безопасность такого подхода. Вероятность ошибки чрезвычайно низка, но вам вряд ли захочется стать одним из тех немногих, кого угораздит столкнуться с такой прерывистой ошибкой. Лучше сделать все правильно, даже если код покажется поначалу загадочным и не совсем понятным.

Далее в рассматриваемом здесь примере определяется, что именно должно произойти, если пользователь закроет фрейм приложения. В данном случае программа должна завершить свою работу. Для этого служит следующая строка кода:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Если бы в программе использовалось несколько фреймов, завершать работу только потому, что пользователь закрыл один из них, было бы совершенно не обязательно. По умолчанию закрытый фрейм исчезает с экрана, а программа продолжает свою работу. (Было бы, конечно, неплохо, если бы программа завершалась после исчезновения *последнего* фрейма с экрана, но, к сожалению, Swing действует иначе.)

Простое создание фрейма не приводит к его автоматическому появлению на экране. В начале своего существования все фреймы невидимы. Это дает возможность добавлять во фрейм компоненты еще до того, как он впервые появится на экране. Для отображения фрейма на экране в методе `main()` вызывается метод `setVisible()`.



НА ЗАМЕТКУ! До появления версии Java SE 5.0 для отображения фрейма на экране можно было вызывать метод `show()`. Дело в том, что суперклассом для класса `JFrame` является класс `Window`, а для класса `Window` — класс `Component`, который также содержит метод `show()`. С версии Java SE 1.2 метод `Component.show()` не рекомендован к использованию, поэтому, для того, чтобы сделать компонент видимым, лучше делать вызов `setVisible(true)`. Следует, однако, иметь в виду, что до появления версии Java SE 1.4 никаких ограничений на применение метода `Window.show()` не накладывалось. Метод `show()` удобен тем, что он одновременно делает окно видимым и перемещает его на передний план. Увы, данное преимущество уже стало достоянием прошлого. В версии Java SE 5.0 метод `show()` также переведен в разряд не рекомендованных к применению.

После диспетчеризации операторов инициализации происходит выход из метода `main()`. Обратите внимание на то, что это не приводит к прекращению работы программы. Завершается лишь ее основной поток. Поток диспетчеризации событий, обеспечивающий нормальную работу программы, продолжает действовать до тех пор, пока она не завершится закрытием фрейма или вызовом метода `System.exit()`.

Окно выполняющейся программы показано на рис. 7.5. Как видите, такие элементы, как строка заголовка и пиктограммы для изменения размеров окна, отображаются операционной системой, а не компонентами библиотеки Swing. При запуске этой программы в средах Windows, GTK или Mac OS окно будет выглядеть по-разному. Все, что находится внутри фрейма, отображается средствами Swing. В данной программе фрейм просто заполняется фоном, цвет которого задается по умолчанию.



НА ЗАМЕТКУ! Начиная с версий Java SE 1.4, все строки заголовка и другие элементы оформления фреймов можно отключить, сделав вызов `frame.setUndecorated(true)`.

Расположение фрейма

В классе `JFrame` имеется лишь несколько методов, позволяющих изменить внешний вид фрейма. Разумеется, благодаря наследованию в классе `JFrame` можно использовать методы из его суперклассов, задающие размеры и расположение фрейма. К наиболее важным из них относятся следующие методы.

- Методы `setLocation()` и `setBounds()`, устанавливающие положение фрейма.
- Метод `dispose()`, закрывающий окно и освобождающий все системные ресурсы, использованные при его создании.
- Метод `setIconImage()`, сообщающий оконной системе, какая именно пиктограмма должна отображаться в строке заголовка, окне переключателя задач и т.п.
- Метод `setTitle()`, позволяющий изменить текст в строке заголовка.
- Метод `setResizable()`, получающий в качестве параметра логическое значение и определяющий, имеет ли пользователь право изменять размеры фрейма.

Иерархия наследования для класса `JFrame` показана на рис. 7.6.



СОВЕТ. В конце этого раздела описаны наиболее важные методы, позволяющие изменять внешний вид фреймов. Одни из них определены в классе `JFrame`, другие — в различных суперклассах для класса `JFrame`. В процессе разработки приложений часто приходится выбирать наиболее подходящий метод для решения той или иной задачи. К сожалению, найти нужные сведения в документации на JDK не так-то просто, особенно в переопределяемых методах. Например, метод `ToFront()` можно применять к объектам типа `JFrame`, но поскольку он наследуется от класса `Window`, то в документации на класс `JFrame` о нем ничего не сказано. Так, если вы предполагаете, что некоторый метод должен существовать, а в документации на класс, с которым вы работаете, он отсутствует, обратитесь к описанию методов суперклассов этого класса. В начале каждой страницы документации на прикладной интерфейс API имеются гипертекстовые ссылки на суперклассы. После описания новых и переопределенных методов в документации приводится также список всех наследуемых методов.

Как указано в документации на прикладной интерфейс API, методы для изменения размеров и формы фреймов следует искать в классе `Component`, который является предшественником всех объектов ГПИ; а также в классе `Window`, который является суперклассом для класса `Frame`. Например, метод `show()`, который служит для отображения фрейма на экране, находится в классе `Window`, а в классе `Component` имеется метод `setLocation()`, позволяющий изменить расположение компонента. В приведенной ниже строке кода левый верхний угол фрейма размещается в точке, находящейся на расстоянии x пикселей вправо и y пикселей вниз от точки начала отсчета $(0, 0)$ в левом верхнем углу экрана.

```
setLocation(x, y)
```

Аналогично метод `setBounds()` из класса `Component` позволяет одновременно изменить размер и расположение компонента (в частности, объекта типа `JFrame`) с помощью следующего вызова:

```
setBounds(x, y, width, height)
```

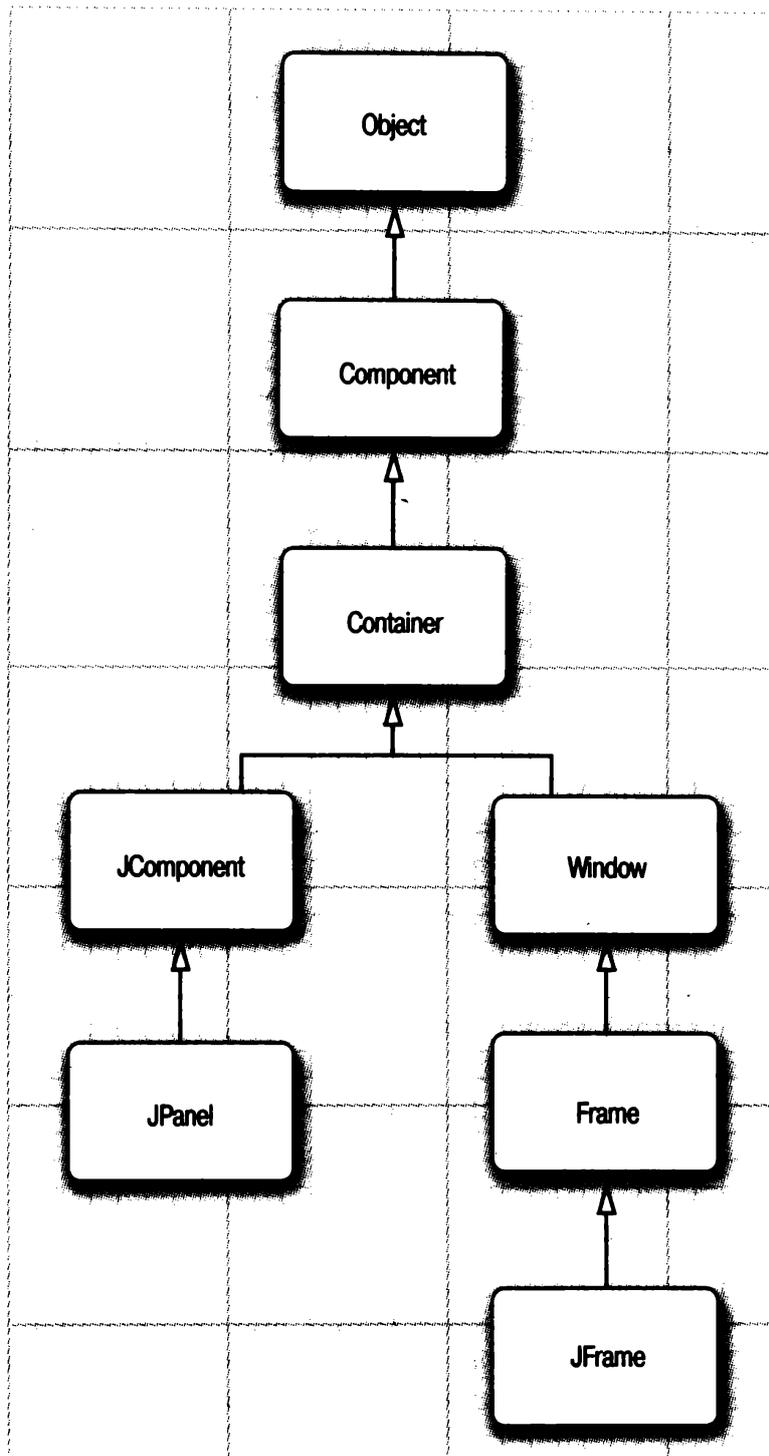


Рис. 7.6. Иерархия наследования классов фреймов и компонентов AWT и Swing

С другой стороны, можно предоставить оконной системе самой управлять расположением окон. Если перед отображением окна сделать следующий вызов:

```
setLoationByPlatform(true);
```

то оконная система сама выберет расположение (но не размеры) окна — как правило, с небольшим смещением относительно предыдущего окна.



НА ЗАМЕТКУ! Координаты расположения фрейма, задаваемые методами `setLocation()` и `setBounds()`, вычисляются относительно всего экрана. Как будет показано в главе 9, координаты других компонентов в контейнере определяются относительно самого контейнера.

Свойства фрейма

Многие методы из классов компонентов объединены в пары для получения и установки соответствующих *свойств*. Примером тому служат следующие методы из класса `Frame`:

```
public String getTitle()
public void setTitle(String title)
```

У каждого свойства имеется свое имя и тип. Имя свойства получается путем изменения первой буквы на строчную после слова `get` или `set` в имени соответствующего метода доступа. Например, в классе `Frame` имеется свойство `title` типа `String`. По существу, `title` является свойством фрейма. При его установке предполагается, что заголовок окна на пользовательском экране изменится. А при получении данного свойства предполагается, что будет возвращено установленное в нем значение.

Неизвестно (да и неважно), каким образом данное свойство реализуется в классе `Frame`. Возможно, для хранения заголовка окна просто используется базовый фрейм. А может быть, для этой цели служит поле экземпляра, как показано ниже.

```
private String title; // не требуется для свойства
```

Если в классе действительно имеется поле экземпляра, совпадающее по имени с нужным свойством, то неизвестно (да и неважно), каким образом реализованы методы получения и установки данного свойства. Возможно, они просто читают и записывают данные в поле экземпляра. А может быть, они делают нечто большее, уведомляя оконную систему всякий раз, когда изменяется заголовок окна.

Из правила получения и установки свойств имеется следующее единственное исключение: для свойств типа `boolean` имя метода получения начинается со слова `is`. Так, в приведенных ниже строках кода определяется свойство `locationByPlatform`. Более подробно свойства фрейма будут рассматриваться в главе 8 второго тома данной книги.

```
public boolean isLocationByPlatform()
public void setLocationByPlatform(boolean b)
```



НА ЗАМЕТКУ! Во многих языках программирования, включая `Visual Basic` и `C#`, имеется встроенная поддержка свойств фрейма. Вполне возможно, что в последующей версии `Java` также появится языковая конструкция для поддержки свойств фрейма.

Определение подходящих размеров фрейма

Напомним, что если размеры не заданы явно, то по умолчанию все фреймы имеют размеры `0x0` пикселей. Чтобы не усложнять рассмотренный выше пример, размеры фреймов были заданы таким образом, чтобы окно нормально отображалось на большинстве мониторов. Но при разработке приложений на профессиональном уровне сначала следует проверить разрешение экрана монитора, а затем написать код, изменяющий размеры фрейма в соответствии с полученной величиной. Ведь окно, которое отлично выглядит на экране портативного компьютера, на экране монитора с большим разрешением будет похоже на почтовую марку.

Для того чтобы определить размеры экрана, необходимо выполнить следующие действия. Сначала вызывается статический метод `getDefaultToolkit()` из класса

Toolkit, который возвращает объект типа Toolkit. (Класс Toolkit содержит много методов, предназначенных для взаимодействия с оконной системой конкретной платформы.) Затем вызывается метод `getScreenSize()`, который возвращает размеры экрана в виде объекта типа `Dimension`. Этот объект содержит ширину и высоту в открытых (!) переменных `width` и `height` соответственно. Ниже приведен фрагмент кода, с помощью которого определяются размеры экрана.

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
```

Из этих размеров фрейма используется лишь половина для указания оконной системе расположения фрейма следующим образом:

```
setSize(screenWidth / 2, screenHeight / 2);
setLocationByPlatform(true);
```

Кроме того, для фрейма предоставляется пиктограмма. Процесс отображения рисунков на экране также зависит от операционной системы, и поэтому для загрузки рисунка снова потребуется объект типа Toolkit. Загруженный в итоге рисунок устанавливается затем в качестве пиктограммы, как показано ниже.

```
Image img = new ImageIcon("icon.gif").getImage();
setIconImage(img);
```

В зависимости от конкретной операционной системы эта пиктограмма выводится по-разному. Например, в Windows пиктограмма отображается в левом верхнем углу окна, и ее можно увидеть в списке активных задач, если нажать комбинацию клавиш `<Alt+Tab>`.

В листинге 7.2 приведен исходный код программы, где выполняются все описанные выше действия по расположению фрейма. При выполнении этой программы обратите особое внимание на пиктограмму Core Java (Основы Java).

Листинг 7.2. Исходный код из файла `sizedFrame/SizedFrameTest.java`

```
1 package sizedFrame;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * @version 1.32 2007-04-14
8  * @author Cay Horstmann
9  */
10 public class SizedFrameTest
11 {
12     public static void main(String[] args)
13     {
14         EventQueue.invokeLater(new Runnable()
15         {
16             public void run()
17             {
18                 JFrame frame = new SizedFrame();
19                 frame.setTitle("SizedFrame");
20                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21                 frame.setVisible(true);
22             }
14 }
```

```
23     });
24     }
25 }
26
27 class SizedFrame extends JFrame
28 {
29     public SizedFrame()
30     {
31         // получить размеры экрана
32
33         Toolkit kit = Toolkit.getDefaultToolkit();
34         Dimension screenSize = kit.getScreenSize();
35         int screenHeight = screenSize.height;
36         int screenWidth = screenSize.width;
37
38         // задать ширину и высоту фрейма, предоставив платформе
39         // возможность самой выбрать местоположение фрейма
40
41         setSize(screenWidth / 2, screenHeight / 2);
42         setLocationByPlatform(true);
43
44         // задать пиктограмму для фрейма
45
46         Image img = new ImageIcon("icon.gif").getImage();
47         setIconImage(img);
48     }
49 }
```

Ниже дается ряд дополнительных рекомендаций по поводу обращения с фреймами.

- Если фрейм содержит только стандартные компоненты вроде кнопок и текстовых полей, просто вызовите метод `pack()`, чтобы установить размеры фрейма. Фрейм будет установлен с минимальными размерами, достаточными для размещения всех его компонентов. Зачастую главный фрейм программы приходится устанавливать с максимальными размерами. Начиная с версии Java SE 1.4, можно развернуть фрейм до максимума, просто сделав следующий вызов:

```
frame.setExtendedState(Frame.MAXIMIZED_BOTH);
```

- Целесообразно также запоминать расположение и размеры фрейма, заданные пользователем, чтобы восстановить эти значения при очередном запуске приложения. В главе 10 будет показано, как пользоваться для этой цели прикладным интерфейсом Preferences API.
- Если вы разрабатываете приложение, в котором выгодно используются преимущества многоэкранного дисплея, применяйте классы `GraphicsEnvironment` и `GraphicsDevice` для определения размеров отдельных экранов.
- Класс `GraphicsDevice` позволяет также выполнять приложение в полноэкранном режиме.

java.awt.Component 1.0

- `boolean isVisible()`
Получают или устанавливают свойство видимости `visible`. Компоненты являются видимыми изначально, кроме компонентов верхнего уровня типа `JFrame`.
- `void setVisible(boolean b)`
Получают или устанавливают свойство видимости `visible`. Компоненты являются видимыми изначально, кроме компонентов верхнего уровня типа `JFrame`.
- `void setSize(int width, int height) 1.1`
Устанавливает текущую ширину и высоту компонента.
- `void setLocation(int x, int y) 1.1`
Перемещает компонент в новую точку. Если компонент не относится к верхнему уровню, то его координаты `x` и `y` отсчитываются относительно контейнера, а иначе используется экранная система координат (например, для объектов типа `JFrame`).
- `void setBounds(int x, int y, int width, int height) 1.1`
Перемещает текущий компонент и изменяет его размеры. Расположение левого верхнего угла задают параметры `x` и `y`, а новый размер — параметры `width` и `height`.
- `Dimension getSize() 1.1`
- `void setSize(Dimension d) 1.1`
Получают или устанавливают свойство `size`, задающее размеры текущего компонента.

java.awt.Window 1.0

- `void toFront()`
Выводит окно поверх других окон на экране.
- `void toBack()`
Размещает данное окно позади всех остальных окон, выведенных на экран, соответственно переупорядочивая их.
- `boolean isLocationByPlatform() 5.0`
- `void setLocationByPlatform(boolean b) 5.0`
Получают или устанавливают свойство `locationByPlatform`. Если это свойство установлено до того, как данное окно отображено, то подходящее расположение выбирает платформа.

java.awt.Frame 1.0

- `boolean isResizable()`
- `void setResizable(boolean b)`
Получают или устанавливают свойство `resizable`. Если это свойство установлено, то пользователь может изменять размеры фрейма.

- `String getTitle()`
- `void setTitle(String s)`

Получают или устанавливают свойство `title`, определяющее текст в строке заголовка фрейма.

- `Image getIconImage()`
- `void setIconImage(Image image)`

Получают или устанавливают свойство `iconImage`, определяющее пиктограмму фрейма. Оконная система может отображать пиктограмму как часть оформления фрейма или в каком-нибудь другом месте.

- `boolean isUndecorated()` 1.4
- `void setUndecorated(boolean b)` 1.4

Получают или устанавливают свойство `undecorated`. Когда свойство установлено, фрейм отображается без таких подробностей оформления, как строка заголовка или кнопка закрытия. Этот метод должен быть вызван до отображения фрейма.

- `int getExtendedState()` 1.4
- `void setExtendedState(int state)` 1.4

Получают или устанавливают расширенное состояние окна. Переменная `state` может принимать следующие значения:

- `Frame.NORMAL`
- `Frame.ICONIFIED`
- `Frame.MAXIMIZED_HORIZ`
- `Frame.MAXIMIZED_VERT`
- `Frame.MAXIMIZED_BOTH`

java.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`
Возвращает объект типа `Toolkit`, предусмотренный по умолчанию.
- `Dimension getScreenSize()`
Загружает изображение из файла `filename`.

javax.swing.ImageIcon 1.2

- `ImageIcon(String имя_файла)`
Конструирует пиктограмму, изображение которой хранится в файле.
- `Image getImage()`
Получает изображение данной пиктограммы.

Отображение данных в компоненте

В этом разделе будет показано, как выводить данные во фрейме. Так, в примере программы из главы 3 символьная строка выводилась на консоль в текстовом режиме. А теперь сообщение "Not a Hello, World program" (Нетривиальная программа типа "Здравствуй, мир") будет выведено в фрейме, как показано на рис. 7.7.

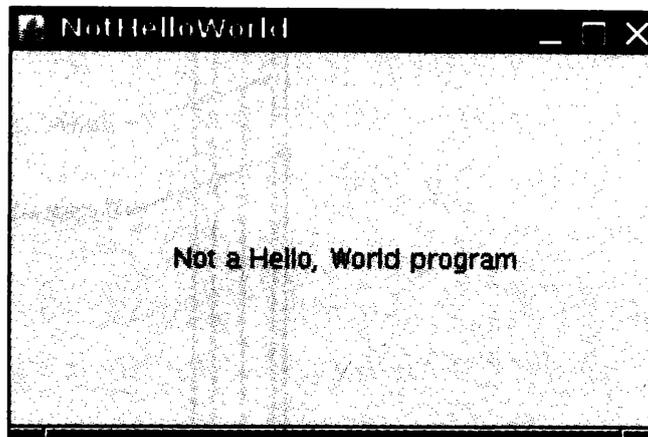


Рис. 7.7. Фрейм, в который выводятся данные

Строку сообщения можно вывести непосредственно во фрейм, но на практике никто так не поступает. В Java фреймы предназначены именно для того, чтобы служить контейнерами для компонентов (например, меню или других элементов пользовательского интерфейса). Как правило, рисунки выводятся в другом компоненте, который добавляется во фрейм.

Оказывается, что структура класса `JFrame` довольно сложная. Его внутреннее строение показано на рис. 7.8. Как видите, класс `JFrame` состоит из четырех областей, каждая из которых представляет собой отдельную панель. Корневая, многослойная и прозрачная панели не представляют особого интереса. Они нужны лишь для оформления меню и панели содержимого в определенном стиле. Наиболее интересной для применения библиотеки `Swing` является *панель содержимого*. При оформлении фрейма его компоненты добавляются на панели содержимого с помощью следующего кода:

```
Container contentPane = frame.getContentPane();
Component c = . . .;
contentPane.add(c);
```

В Java SE 1.4 и более старых версиях при попытке вызвать метод `add()` из класса `JFrame` генерировалось исключение с сообщением "Do not use `JFrame.add()`. Use `JFrame.getContentPane().add()` instead" (Не пользуйтесь методом `JFrame.add()`. Вместо него пользуйтесь методом `JFrame.getContentPane().add()`). В версии Java SE 5.0 был внедрен метод `JFrame.add()`, который лишь переадресует вызов методу `add()` для панели содержимого. Таким образом, начиная с версии Java SE 5.0, можно делать следующий вызов:

```
frame.add(c);
```

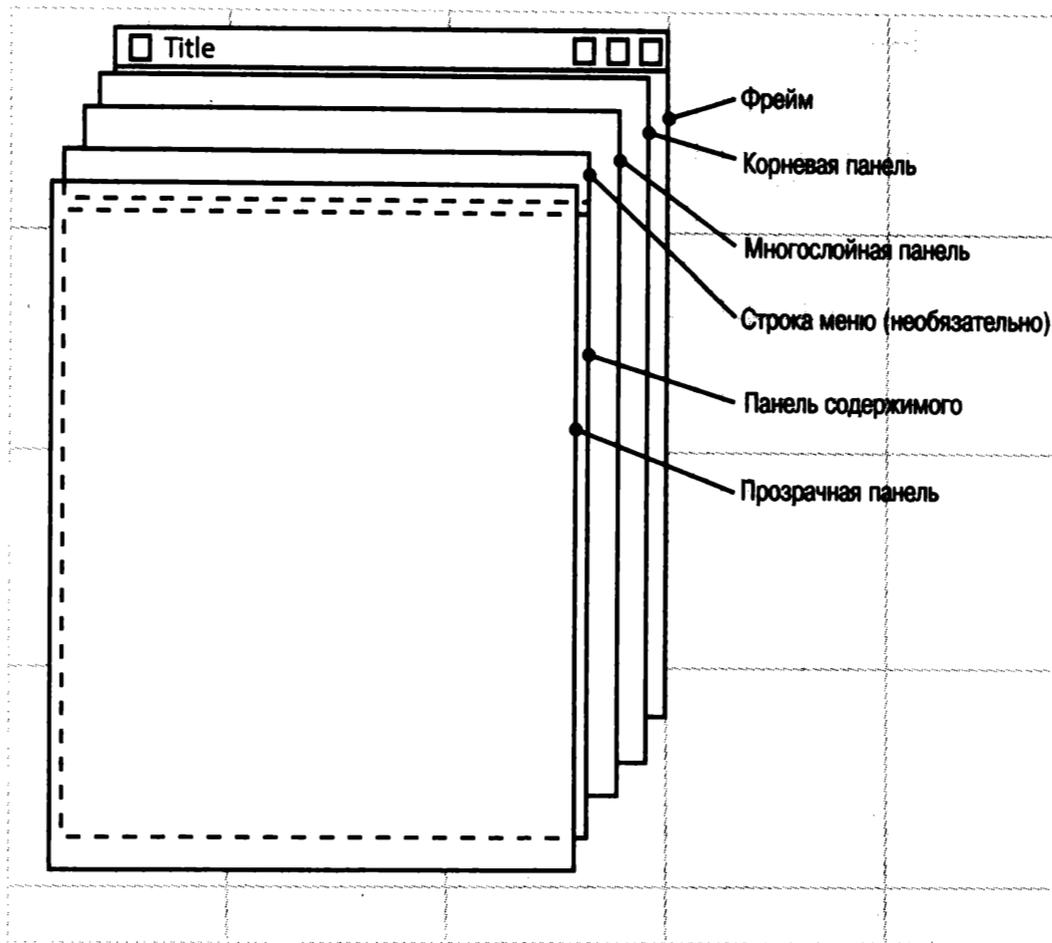


Рис. 7.8. Внутреннее строение класса JFrame

В данном случае требуется добавить во фрейм единственный компонент, в котором будет выводиться сообщение. Для того чтобы отобразить компонент, следует сначала определить класс, расширяющий класс `JComponent`, а затем переопределить метод `paintComponent()` этого класса. Метод `paintComponent()` получает в качестве параметра объект типа `Graphics`, который содержит набор установок для отображения рисунков и текста. В нем, например, задается шрифт и цвет текста. Все операции рисования графики в Java выполняются с помощью объектов класса `Graphics`. В этом классе предусмотрены методы для рисования узоров, графических изображений и текста.



НА ЗАМЕТКУ! Параметр типа `Graphics` сродни контексту устройства в среде Windows или графическому контексту в среде X11.

В приведенном ниже фрагменте кода демонстрируется в общих чертах, каким образом создается компонент для рисования графики.

```
class MyComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        код для рисования
    }
}
```

Всякий раз, когда возникает потребность перерисовать окно независимо от конкретной причины, обработчик событий уведомляет об этом компонент.

В результате выполняется метод `paintComponent()` для всех компонентов. Метод `paintComponent()` вообще не следует вызывать вручную. Когда требуется перерисовать окно приложения, он вызывается автоматически и вмешиваться в этот процесс не рекомендуется.

В каких случаях требуется автоматическая перерисовка? Она требуется, например, при увеличении или уменьшении размеров окна. Если пользователь открыл новое окно, перекрыв им уже существующее окно приложения, а затем закрыл его, оставшееся окно оказывается поврежденным и должно быть перерисовано. (Графическая система не сохраняет в памяти пиксели нижележащего окна.) И наконец, когда окно выводится на экран впервые, следует выполнить код, указывающий, как и где должны отображаться его исходные элементы.



СОВЕТ. Если необходимо принудительно перерисовать экран, вместо метода `paintComponent()` следует вызвать метод `repaint()`. Этот метод, в свою очередь, обратится к методу `paintComponent()` каждого компонента и передаст ему настроенный должным образом объект типа `Graphics`.

Как следует из приведенного выше фрагмента кода, у метода `paintComponent()` имеется один параметр типа `Graphics`. При выводе на экран размеры, сохраняемые в объекте типа `Graphics`, указываются в пикселях. Координаты $(0, 0)$ соответствуют левому верхнему углу компонента, на поверхности которого выполняется рисование.

Вывод текста на экран считается особой разновидностью рисования. Для этой цели в классе `Graphics` имеется метод `drawString()`, который вызывается следующим образом:

```
g.drawString(text, x, y)
```

В данном случае текстовая строка "Not a Hello, World Program" выводится на экран в исходном окне, занимающем приблизительно четверть экрана по ширине и половину его по высоте. Мы пока еще не обсуждали, каким образом задается размер текстовой строки, тем не менее, установим ее начало в точке с координатами $(75, 100)$. Это означает, что вывод первого символа из строки начинается с точки, отстоящей на 75 пикселей от правого края и на 100 пикселей от верхнего края окна. (На самом деле расстояние 100 пикселей отсчитывается от верхнего края до базовой линии строки, как поясняется далее в этой главе.) Метод `paintComponent()` выглядит приблизительно так:

```
class NotHelloWorldComponent extends JComponent
{
    public static final int MESSAGE_X = 75;
    public static final int MESSAGE_Y = 100;

    public void paintComponent(Graphics g)
    {
        g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
    }
    . . .
}
```

И наконец, компонент должен сообщить своим пользователям, насколько большим он должен быть. Для этого переопределяется метод `getPreferredSize()`, возвращающий объект класса `Dimension` с предпочтительными шириной и высотой, как показано ниже.

```
class NotHelloWorldComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    . . .
    public Dimension getPreferredSize()
        { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
}
```

Если при заполнении фрейма одним или несколькими компонентами требуется лишь воспользоваться их предпочтительными размерами, то вместо метода `setSize()` вызывается метод `pack()`:

```
class NotHelloWorldFrame extends JFrame
{
    public NotHelloWorldFrame()
    {
        add(new NotHelloWorldComponent());
        pack();
    }
}
```

Весь исходный код рассмотренной выше программы приведен в листинге 7.3.



НА ЗАМЕТКУ! Вместо расширения класса `JComponent` некоторые программисты предпочитают расширять класс `JPanel`, предназначенный в качестве контейнера для других компонентов, хотя рисовать можно и в нем самом. Следует только иметь в виду одно существенное отличие. Панель *непрозрачна*, а это означает, что она отвечает за рисование всех пикселей в ее границах. Простейший способ рисовать компоненты в этих границах — залить панель цветом фона, сделав вызов `super.paintComponent()` в методе `paintComponent()` каждого подкласса панели следующим образом:

```
class NotHelloWorldPanel extends JPanel
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        код для рисования
    }
}
```

Листинг 7.3. Исходный код из файла `notHelloWorld/NotHelloWorld.java`

```
1 package notHelloWorld;
2
3 import javax.swing.*;
4 import java.awt.*;
5
6 /**
7  * @version 1.32 2007-06-12
8  * @author Cay Horstmann
9  */
10 public class NotHelloWorld
11 {
12     public static void main(String[] args)
```

```
13 {
14     EventQueue.invokeLater(new Runnable()
15     {
16         public void run()
17         {
18             JFrame frame = new NotHelloWorldFrame();
19             frame.setTitle("NotHelloWorld");
20             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21             frame.setVisible(true);
22         }
23     });
24 }
25 }
26 /**
27  * Фрейм, содержащий панели сообщений
28  */
29 class NotHelloWorldFrame extends JFrame
30 {
31     public NotHelloWorldFrame()
32     {
33         add(new NotHelloWorldComponent());
34         pack();
35     }
36 }
37
38 /**
39  * Компонент, выводящий сообщение
40  */
41 class NotHelloWorldComponent extends JComponent
42 {
43     public static final int MESSAGE_X = 75;
44     public static final int MESSAGE_Y = 100;
45
46     private static final int DEFAULT_WIDTH = 300;
47     private static final int DEFAULT_HEIGHT = 200;
48
49     public void paintComponent(Graphics g)
50     {
51         g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
52     }
53
54     public Dimension getPreferredSize()
55     { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
56 }
```

javax.swing.JFrame 1.2

- Container getContentPane()
- Возвращает панель содержимого для данного объекта типа JFrame.
- Component add(Component c)

Добавляет указанный компонент на панели содержимого данного фрейма. (До появления версии Java SE 5.0 вызов этого метода приводил к генерированию исключения.)

java.awt.Component 1.0

- `void repaint()`
Вызывает перерисовку компонента. Перерисовка выполняется сразу же после того, как возникнут условия, позволяющие это сделать.
- `Dimension getPreferredSize()`
Этот метод переопределяется для возврата предпочтительных размеров данного компонента.

javax.swing.JComponent 1.2

- `void paintComponent(Graphics g)`
Этот метод переопределяется для описания способа рисования заданного компонента.

java.awt.Window 1.0

- `void pack()`
Изменяет размеры данного окна, принимая во внимание предпочтительные размеры его компонентов.

Двухмерные формы

В версии 1.0 в классе `Graphics` появились методы для рисования линий, прямоугольников, эллипсов и прочих двухмерных форм. Но эти операции рисования предоставляют слишком ограниченный набор функциональных возможностей. Например, в них нельзя изменить толщину линии и повернуть двухмерную форму на произвольный угол.

В версии Java SE 1.2 была внедрена библиотека *Java 2D*, в которой реализован целый ряд эффективных операций рисования двухмерной графики. В этой главе будут рассмотрены лишь основы работы с библиотекой *Java 2D*, а ее более развитые средства подробнее обсуждаются в главе 7 второго тома данной книги.

Для того чтобы нарисовать двухмерную форму средствами библиотеки *Java 2D*, нужно создать объект класса `Graphics2D`. Это подкласс, производный от класса `Graphics`. Начиная с версии Java SE 2, такие методы, как `paintComponent()`, автоматически получают объекты класса `Graphics2d`. Нужно лишь произвести приведение типов, как показано ниже.

```
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;
    . . .
}
```

При создании геометрических форм в библиотеке *Java 2D* применяется объектно-ориентированный подход. В частности, перечисленные ниже классы,

предназначенные для рисования линий, прямоугольников и эллипсов, реализуют интерфейс Shape.

```
Line2D  
Rectangle2D  
Ellipse2D
```



НА ЗАМЕТКУ! В библиотеке Java 2D поддерживается рисование и более сложных двухмерных форм, в том числе дуг, квадратичных и кубических кривых, а также произвольных линий. Подробнее об этом речь пойдет в главе 7 второго тома данной книги.

Для того чтобы нарисовать двухмерную форму, нужно сначала создать объект соответствующего класса, реализующего интерфейс Shape, а затем вызвать метод `draw()` из класса `Graphics2D`, как показано в следующем примере:

```
Rectangle2D rect = . . . ;  
g2.draw(rect);
```



НА ЗАМЕТКУ! До появления библиотеки Java 2D программирующие на Java пользовались для рисования двухмерных форм методами из класса `Graphics`, например, методом `drawRectangle()`. На первый взгляд старый способ вызова методов выглядит немного проще. Но, пользуясь библиотекой Java 2D, программист получает большую свободу действий, поскольку он может впоследствии уточнить и улучшить нарисованные формы, используя многочисленные средства, предусмотренные в этой библиотеке.

Использование классов из библиотеки Java 2D сопряжено с определенными трудностями. В отличие от методов рисования из версии Java 1.0, в которых применялись целочисленные координаты, выраженные в пикселях, в библиотеке Java 2D используются координаты, представленные числами с плавающей точкой. Во многих случаях это очень удобно, поскольку дает возможность сначала сформировать двухмерную форму с координатами в естественных единицах измерения (например, дюймах или миллиметрах), а затем перевести их в пиксели. Внутренние вычисления в библиотеке Java 2D выполняются в формате чисел с плавающей точкой и одинарной точностью. Этого вполне достаточно, поскольку главной целью вычислений геометрических форм является их вывод в пикселях на экран или на принтер. Все округления в ходе вычислений не выходят за пределы одного пикселя, что совершенно не влияет на внешний вид фигуры. Более того, вычисления в формате чисел с плавающей точкой и одинарной точностью на некоторых платформах выполняются быстрее, а числовые значения типа `float` занимают вдвое меньше памяти, чем числовые значения типа `double`.

И все же манипулировать числовыми значениями типа `float` иногда бывает неудобно, поскольку в вопросах приведения типов язык Java непреклонен и требует преобразовывать числовые значения типа `double` в числовые значения типа `float` явным образом. Рассмотрим для примера следующее выражение:

```
float f = 1.2; // ОШИБКА!
```

Это выражение не будет скомпилировано, поскольку константа `1.2` имеет тип `double`, а компилятор зафиксирует потерю точности. В таком случае при формировании константы с плавающей точкой придется явно указать суффикс `F` следующим образом:

```
float f = 1.2F; // Правильно!
```

Теперь рассмотрим следующий фрагмент кода:

```
rectangle2D r = ...
float f = r.getWidth(); // ОШИБКА!
```

Этот фрагмент кода не скомпилируется по тем же причинам. Метод `getWidth()` возвращает число, имеющее тип `double`. На этот раз нужно выполнить приведение типов:

```
float f = (float)r.getWidth(); // Правильно!
```

Указание суффиксов и приведение типов доставляет немало хлопот, поэтому разработчики библиотеки Java 2D решили предусмотреть *две* версии каждого класса для рисования фигур: в первой версии все координаты выражаются числовыми значениями типа `float` (для дисциплинированных программистов), а во второй — числовыми значениями типа `double` (для ленивых). (Относя себя ко второй группе, мы указываем координаты числовыми значениями типа `double` во всех примерах рисования двумерных форм, приведенных в этой книге.)

Разработчики библиотеки выбрали необычный и запутанный способ создания пакетов, соответствующих этим двум версиям. Рассмотрим для примера класс `Rectangle2D`. Это абстрактный класс, имеющий два следующих конкретных подкласса, каждый из которых является внутренним и статическим:

```
Rectangle2D.Float
Rectangle2D.Double
```

На рис. 7.9 приведена блок-схема наследования этих классов.

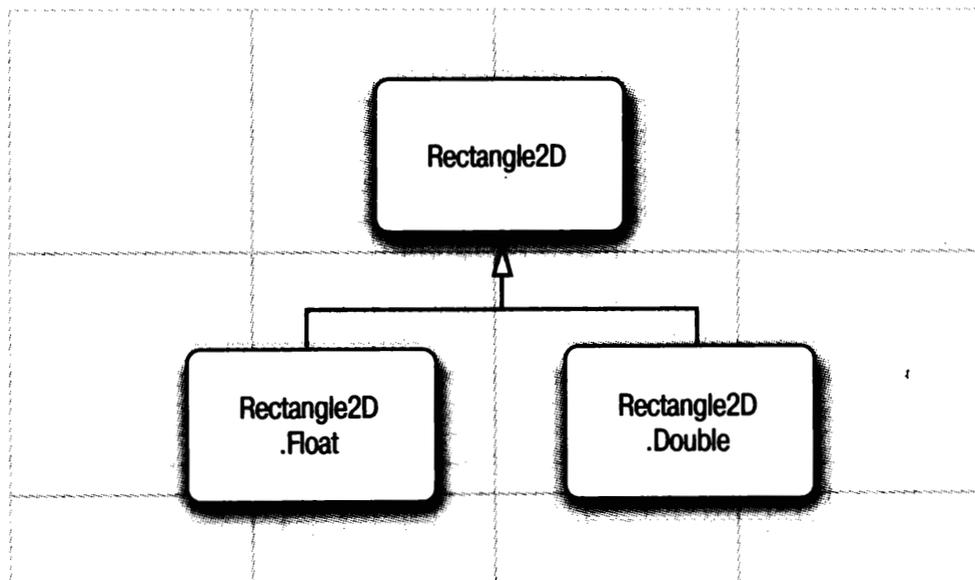


Рис. 7.9. Классы для рисования прямоугольников

Лучше всего совсем пренебречь тем, что эти два конкретных класса являются внутренними и статическими. Ведь это всего лишь уловка, чтобы избежать употребления таких имен, как `FloatRectangle2D` и `DoubleRectangle2D`. (Статические внутренние классы подробно рассматривались в главе 6.)

При построении объекта типа `Rectangle2d.Float` используются координаты, представленные числовыми значениями типа `float`, а в объектах типа `Rectangle2d.Double` они выражаются числовыми значениями типа `double`, как показано ниже.

```
Rectangle2D.Float floatRect =  
    new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);  
Rectangle2D.Double doubleRect =  
    new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

Оба класса, `Rectangle2D.Float` и `Rectangle2D.Double`, расширяют один и тот же класс `Rectangle2D`, а их методы переопределяют методы из их суперкласса, поэтому нет никакой нужды запоминать точный тип координат, указываемых при создании двухмерных форм. Так, для ссылок на прямоугольные объекты достаточно указывать переменные типа `Rectangle2D` следующим образом:

```
Rectangle2D floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);  
Rectangle2D doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

Таким образом, обращаться с внутренними классами приходится лишь при построении объектов, описывающих двухмерные формы. В приведенном выше фрагменте кода параметры конструктора задают координаты верхнего левого угла, ширину и высоту прямоугольника.



НА ЗАМЕТКУ! На самом деле в классе `Rectangle2D.Float` имеется один дополнительный метод `setRect(float x, float y, float h, float w)`, который не наследуется от класса `Rectangle2D`. Если вместо ссылок типа `Rectangle2D.Float` использовать ссылки типа `Rectangle2D`, этот метод станет недоступным. Но это небольшая потеря, поскольку в классе `Rectangle2D` имеется свой метод `setRect()` с параметрами типа `double`.

Параметры и значения, возвращаемые методами из класса `Rectangle2D`, относятся к типу `double`. Например, метод `getWidth()` возвращает значение типа `double`, даже если ширина хранится в виде числового значения типа `float` в объекте типа `Rectangle2D.Float`.



СОВЕТ. Чтобы не оперировать числовыми значениями типа `float`, пользуйтесь классами, в которых координаты выражаются числовыми значениями типа `double`. Но если вам нужно создать тысячи объектов двухмерных форм, то следует подумать об экономии памяти. И в этом вам помогут классы, в которых координаты задаются числовыми значениями типа `float`.

Все, что было сказано выше о классах `Rectangle2D`, относится к любым другим классам, предназначенным для рисования двухмерных форм. Кроме того, существует еще и класс `Point2D` с подклассами `Point2D.Float` и `Point2D.Double`. Ниже приведен пример создания объекта точки.

```
Point2D p = new Point2D.Double(10, 20);
```



СОВЕТ. Класс `Point2D` очень полезен. Обращение с объектами класса `Point2D` больше соответствует объектно-ориентированному принципу программирования, чем оперирование отдельными координатами `x` и `y`. Многие конструкторы и методы принимают параметры, имеющие тип `Point2D`. И при всякой возможности рекомендуется пользоваться именно этими объектами, поскольку они делают вычисления геометрических форм более понятными.

Классы `Rectangle2D` и `Ellipse2D` являются производными от одного и того же суперкласса `RectangularShape`. Как известно, эллипс не является прямоугольной фигурой, но он ограничен прямоугольником (рис. 7.10).

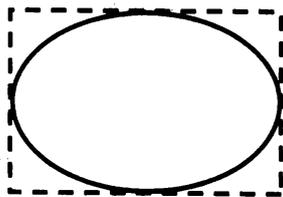


Рис. 7.10. Прямоугольник, ограничивающий эллипс

В классе `RectangularShape` определено более 20 методов, общих для подобных двумерных форм. К их числу относятся такие полезные методы, как `getWidth()`, `getHeight()`, `getCenterX()` и `getCenterY()` (но, к сожалению, на момент написания этой книги в этом классе пока еще отсутствовал метод `getCenter()`, возвращающий центр геометрической формы в виде объекта типа `Point2D`).

Следует также заметить, что в новую библиотеку вошли старые классы из библиотеки Java 1.0, заняв свое место в иерархии наследования. Так, классы `Rectangle` и `Point`, в объектах которых координаты прямоугольника и его центра хранятся в виде целых значений, расширяют классы `Rectangle2D` и `Point2D`.

На рис. 7.11 показаны отношения между классами, представляющими двумерные формы. Но подклассы `Float` и `Double` на этом рисунке не показаны. Классы, унаследованные от предыдущей версии, представлены закрашенными прямоугольниками.

Объекты типа `Rectangle2D` и `Ellipse2D` создаются довольно просто. Для этого достаточно указать следующие входные данные:

- координаты x и y левого верхнего угла;
- ширину и высоту двумерной формы.

Для эллипсов эти параметры относятся к ограничивающим их прямоугольникам. Например, в приведенной ниже строке кода создается эллипс, ограниченный прямоугольником, левый верхний угол которого находится в точке с координатами (150, 200), а также шириной и высотой 100 и 50 пикселей соответственно.

```
Ellipse2D e = new Ellipse2D.Double(150, 200, 100, 50);
```

Но иногда бывает нелегко определить, где должен находиться левый верхний угол двумерной формы. Нередко прямоугольник задается противоположными вершинами, но они совсем не обязательно соответствуют левому верхнему и нижнему правому углу. В этом случае построить прямоугольник нельзя:

```
Rectangle2D rect =
    new Rectangle2D.Double(px, py, qx - px, qy - py); // ОШИБКА!
```

Если параметр p не соответствует координатам верхнего левого угла, одна или обе разности координат окажутся отрицательными, а прямоугольник — пустым. В этом случае нужно сначала создать пустой прямоугольник, а затем вызвать метод `setFrameFromDiagonal()` следующим образом:

```
Rectangle2D rect = new Rectangle2D.Double();
rect.setFrameFromDiagonal(px, py, qx, qy);
```

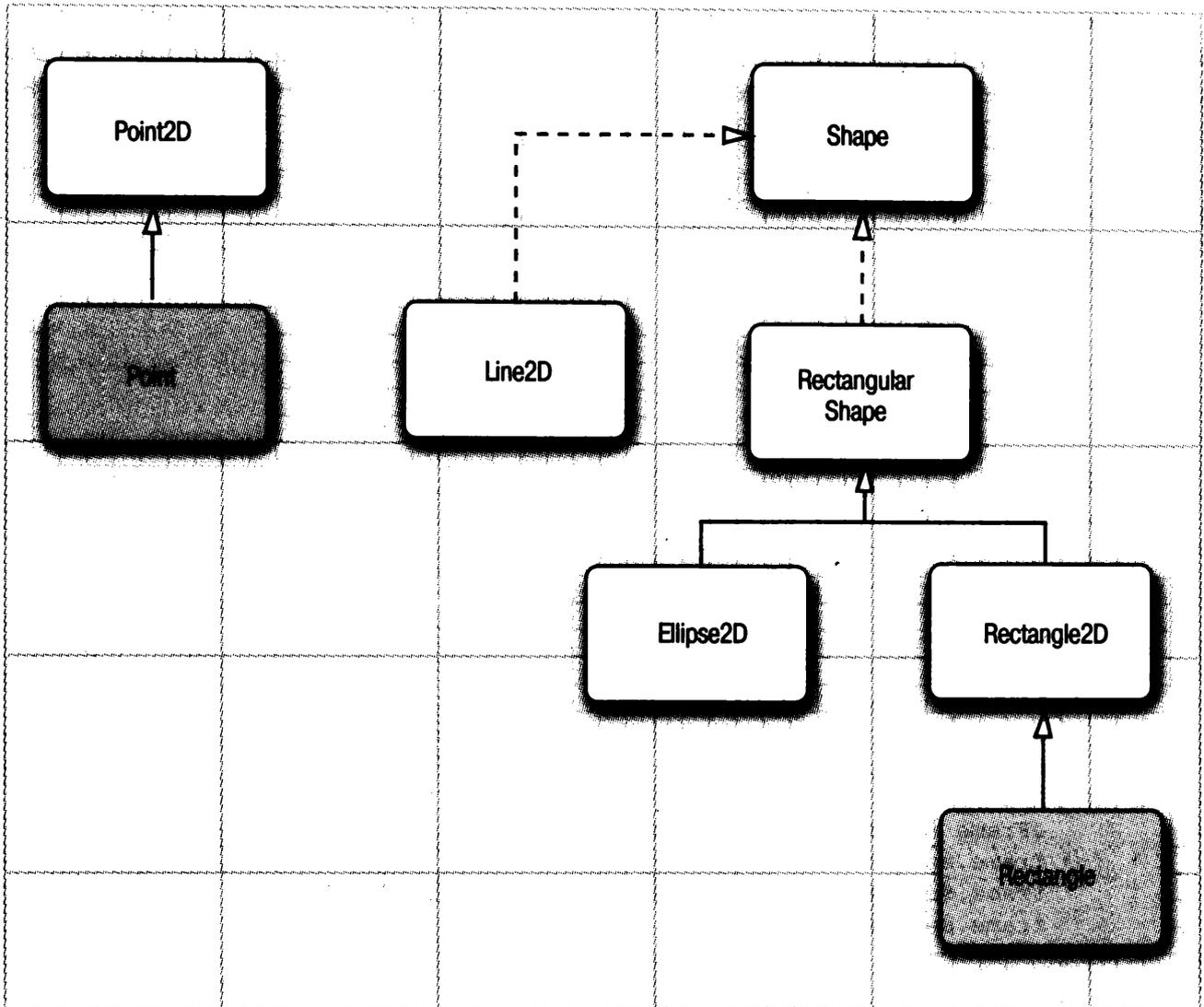


Рис. 7.11. Отношения между классами, представляющими двухмерные формы

А еще лучше, если противоположные вершины известны и представлены объектами *p* и *q* типа *Point2D*. В этом случае можно воспользоваться следующим оператором:

```
rect.setFrameFromDiagonal(p, q);
```

При создании эллипса обычно известны центр, ширина и высота ограничивающего прямоугольника, но не координаты его левого верхнего угла. Существует метод *setFrameFromCenter()*, использующий центр, но ему нужно задать одну из четырех угловых точек. Следовательно, эллипс лучше создать следующим образом:

```
Ellipse2D ellipse = new Ellipse2D.Double(centerX - width / 2,
                                         centerY - height / 2, width, height);
```

Для того чтобы нарисовать линию, нужно задать начальную и конечную точки в виде объектов типа *Point2D* или в виде пары чисел, как показано ниже.

```
Line2D line = new Line2D.Double(start, end);
```

или

```
Line2D line = new Line2D.Double(startX, startY, endX, endY);
```

В программе, исходный код которой приведен в листинге 7.4, рисуется прямоугольник, эллипс, вписанный в прямоугольник, диагональ прямоугольника, а также окружность, центр которой совпадает с центром прямоугольника. Результат выполнения этой программы показан на рис. 7.12.

Листинг 7.4. Исходный код из файла draw/DrawTest.java

```
1 package draw;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import javax.swing.*;
6
7 /**
8  * @version 1.32 2007-04-14
9  * @author Cay Horstmann
10 */
11 public class DrawTest
12 {
13     public static void main(String[] args)
14     {
15         EventQueue.invokeLater(new Runnable()
16         {
17             public void run()
18             {
19                 JFrame frame = new DrawFrame();
20                 frame.setTitle("DrawTest");
21                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22                 frame.setVisible(true);
23             }
24         });
25     }
26 }
27
28 /**
29  * Фрейм, содержащий панель с нарисованными двумерными формами
30 */
31 class DrawFrame extends JFrame
32 {
33     public DrawFrame()
34     {
35         add(new DrawComponent());
36         pack();
37     }
38 }
39
40 /**
41  * Компонент, отображающий прямоугольники и эллипсы
42 */
43 class DrawComponent extends JComponent
44 {
45     private static final int DEFAULT_WIDTH = 400;
46     private static final int DEFAULT_HEIGHT = 400;
47
48     public void paintComponent(Graphics g)
49     {
```

```
50 Graphics2D g2 = (Graphics2D) g;
51
52 // нарисовать прямоугольник
53 double leftX = 100;
54 double topY = 100;
55 double width = 200;
56 double height = 150;
57
58 Rectangle2D rect = new Rectangle2D.Double(leftX, topY, width, height);
59 g2.draw(rect);
60 // нарисовать вписанный эллипс
61 Ellipse2D ellipse = new Ellipse2D.Double();
62 ellipse setFrame(rect);
63 g2.draw(ellipse);
64
65 // нарисовать диагональную линию
66 g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY + height));
67
68 // нарисовать окружность с тем же самым центром
69 double centerX = rect.getCenterX();
70 double centerY = rect.getCenterY();
71 double radius = 150;
72
73 Ellipse2D circle = new Ellipse2D.Double();
74 circle.setFrameFromCenter(centerX, centerY, centerX + radius,
75     centerY + radius);
76 g2.draw(circle);
77 }
78
79 public Dimension getPreferredSize()
80     { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
81 }
```

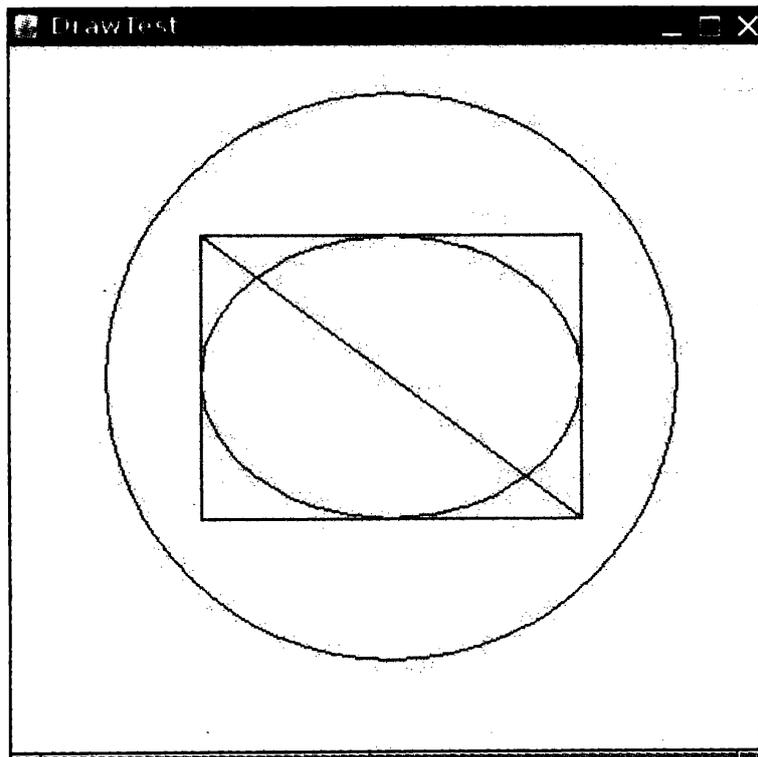


Рис. 7.12. Рисование геометрических двухмерных форм

java.awt.geom.RectangularShape 1.2

- double getCenterX()
- double getCenterY()
- double getMinX()
- double getMinY()
- double getMaxX()
- double getMaxY()

Возвращают координаты центра, наименьшую и наибольшую координату **x** и **y** описанного прямоугольника.

- double getWidth()
- double getHeight()

Возвращают ширину и высоту описанного прямоугольника.

- double getX()
- double getY()

Возвращают координаты **x** и **y** левого верхнего угла описанного прямоугольника.

java.awt.geom.Rectangle2D.Double 1.2

- Rectangle2D.Double(double *x*, double *y*, double *w*, double *h*)

Строит прямоугольник заданных координат верхнего левого угла, ширины и высоты.

java.awt.geom.Rectangle2D.Float 1.2

- Rectangle2D.Float(float *x*, float *y*, float *w*, float *h*)

Строит прямоугольник заданных координат верхнего левого угла, ширины и высоты.

java.awt.geom.Ellipse2D.Double 1.2

- Ellipse2D.Double(double *x*, double *y*, double *w*, double *h*)

Создает эллипс с ограничивающим прямоугольником заданных координат верхнего левого угла, ширины и высоты.

java.awt.geom.Point2D.Double 1.2

- Point2D.Double(double *x*, double *y*)

Рисует точку с заданными координатами.

java.awt.geom.Line2D.Double 1.2

- `Line2D.Double(Point2D start, Point2D end)`
- `Line2D.Double(double startX, double startY, double endX, double endY)`
Рисует линию с заданными координатами начальной и конечной точек.

Окрашивание цветом

Метод `setPaint()` из класса `Graphics2D` позволяет выбрать цвет, который будет применяться при всех дальнейших операциях рисования в графическом контексте. Ниже приведен пример применения этого метода в коде.

```
g2.setPaint(Color.RED);  
g2.drawString("Warning!", 100, 100);
```

Окрашивать цветом можно внутренние участки замкнутых геометрических форм вроде прямоугольников или эллипсов. Для этого вместо метода `draw()` достаточно вызвать метод `fill()` следующим образом:

```
Rectangle2D rect = . . . ;  
g2.setPaint(Color.RED);  
g2.fill(rect); // заполнить прямоугольник красным цветом
```

Для окрашивания разными цветами следует выбрать определенный цвет, нарисовать или заполнить форму, затем выбрать новый цвет, нарисовать или заполнить форму другую фигуру и т.д.



НА ЗАМЕТКУ! В методе `fill()` окрашивание цветом осуществляется на один пиксель меньше вправо и вниз. Так, если сделать вызов `new Rectangle2D.Double(0, 0, 10, 20)`, чтобы нарисовать новый прямоугольник, то в нарисованную форму войдут пиксели с координатами $x = 10$ и $y = 20$. Если же заполнить ту же самую прямоугольную форму выбранным цветом, то пиксели с этими координатами не будут окрашены.

Цвет определяется с помощью класса `Color`. Класс `java.awt.Color` содержит константы, которые соответствуют 13 стандартным цветам: `BLACK` (черный), `BLUE` (синий), `CYAN` (голубой), `DARK_GRAY` (темно-серый), `GRAY` (серый), `GREEN` (зеленый), `LIGHT_GRAY` (светло-серый), `MAGENTA` (пурпурный), `ORANGE` (оранжевый), `PINK` (розовый), `RED` (красный), `WHITE` (белый), `YELLOW` (желтый).



НА ЗАМЕТКУ! До появления версии Java SE 1.4 имена констант, определяющих цвет, задавались строчными буквами, например `Color.red`. Это довольно странно, поскольку константам принято присваивать имена, состоящие из прописных букв. Начиная с версии Java SE 1.4, стандартные названия цветов указываются как прописными, так и строчными буквами (для сохранения обратной совместимости).

Имеется возможность указать произвольный цвет по его красной, зеленой и синей составляющим, создав объект класса `Color`. В конструкторе класса `Color` составляющие цветов задаются целыми значениями в пределах от 0 до 255 (т.е. одним байтом) следующим образом:

```
Color(int redness, int greenness, int blueness)
```

Ниже приведен пример установки специального цвета.

```
g2.setPaint(new Color(0, 128, 128)); // скучный сине-зеленый цвет
g2.drawString("Welcome!", 75, 125);
```



НА ЗАМЕТКУ! Кроме сплошного цвета, можно выбирать более сложные "палитры" с изменением оттенков или рисунков. Более подробно этот вопрос будет обсуждаться в главе, посвященной расширенным средствам AWT, второго тома данной книги. Если вместо объекта типа `Graphics2D` используется объект типа `Graphics`, то для установки цвета нужно применять метод `setColor()`.

Для того чтобы установить *цвет фона*, следует вызвать метод `setBackground()` из класса `Component`, предшественника класса `Jpanel`:

```
MyComponent p = new MyComponent();
p.setBackground(Color.PINK);
```

Существует также метод `setForeground()`. Он задает цвет переднего плана, который используется при рисовании компонента.



СОВЕТ. Методы `brighter()` и `darker()` из класса `Color` позволяют устанавливать более светлые или темные оттенки текущего цвета соответственно. Используя метод `brighter()`, можно, например, без труда выделить заголовок или пункт меню. На самом деле одного вызова метода `brighter()` для увеличения яркости цвета явно недостаточно. Для того чтобы достичь заметного эффекта, желательно вызвать его трижды: `c.brighter().brighter().brighter()`.

Для большинства известных цветов в классе `SystemColor` предусмотрены символические имена. Константы в этом классе определяют цвета, которые используются для окраски различных элементов пользовательского интерфейса. Например, в приведенной ниже строке кода для фона фрейма выбирается цвет, заданный по умолчанию для всех окон пользовательского интерфейса.

```
p.setBackground(SystemColor.window)
```

Фон заполняет окно при каждой перерисовке. Пользоваться цветами, заданными в классе `SystemColor`, особенно удобно, если требуется, чтобы пользовательский интерфейс был выдержан в той же цветовой гамме, что и рабочий стол. В табл. 7.1 приведены названия системных цветов и их описание.

Таблица 7.1. Системные цвета

Цвет	Описание
<code>desktop</code>	Цвет фона рабочего стола
<code>activeCaption</code>	Цвет фона заголовков
<code>activeCaptionText</code>	Цвет текста заголовков
<code>activeCaptionBorder</code>	Цвет обрамления активных заголовков
<code>inactiveCaption</code>	Цвет фона неактивных заголовков
<code>inactiveCaptionText</code>	Цвет текста неактивных заголовков
<code>inactiveCaptionBorder</code>	Цвет обрамления неактивных заголовков

Окончание табл. 7.1

Цвет	Описание
window	Цвет фона окна
windowBorder	Цвет обрамления окна
windowText	Цвет текста в окне
menu	Цвет меню
menuText	Цвет текста меню
text	Цвет фона текста
textText	Цвет текста
textInactiveText	Цвет текста неактивных элементов управления
textHighlight	Цвет фона выделенного текста
textHighlightText	Цвет выделенного текста
control	Цвет фона элементов управления
controlText	Цвет текста элементов управления
controlLtHighlight	Цвет слабого выделения элементов управления
controlHighlight	Цвет выделения элементов управления
controlShadow	Цвет тени элементов управления
controlDkShadow	Темный цвет тени элементов управления
scrollbar	Цвет фона полосы прокрутки
info	Цвет фона информационных сообщений
infoText	Цвет текста информационных сообщений

java.awt.Color 1.0

- `Color(int r, int g, int b)`

Создает объект класса `Color`.

Параметры: *r* Значение красного цвета (0–255)
 g Значение зеленого цвета (0–255)
 b Значение синего цвета (0–255)

java.awt.Graphics 1.0

- `Color getColor()`
- `void setColor(Color c)`

Получают или устанавливают текущий цвет. При выполнении последующих графических операций будет использоваться новый цвет.

Параметры: *c* Новый цвет

java.awt.Graphics2D 1.2

- `Paint getPaint()`
- `void setPaint(Paint p)`

Получают или устанавливают атрибуты рисования для данного графического контекста. Класс `Color` реализует интерфейс `Paint`. Этот метод можно использовать для задания сплошного цвета при рисовании.

- `void fill(Shape s)`
Заполняет текущую нарисованную форму.

java.awt.Component 1.0

- `Color getBackground()`
- `void setBackground(Color c)`

Получают или устанавливают цвет фона.

Параметры: `c` Новый цвет фона

- `Color getForeground()`
- `void setForeground(Color c)`

Получают или устанавливают цвет переднего плана.

Параметры: `c` Новый цвет переднего плана

Специальное шрифтовое оформление текста

Программа, приведенная в начале этой главы, выводила на экран текстовую строку, выделенную шрифтом по умолчанию. Но нередко требуется, чтобы текст отображался разными шрифтами. Шрифты определяются *начертанием* символов. Название начертания состоит из названия *гарнитуры* шрифтов (например, Helvetica) и необязательного суффикса (например, Bold для полужирного начертания). Так, названия Helvetica и Helvetica Bold считаются частью одной и той же гарнитуры шрифтов Helvetica.

Для того чтобы выяснить, какие именно шрифты доступны на отдельном компьютере, следует вызвать метод `getAvailableFamilyNames()` из класса `GraphicsEnvironment`. Этот метод возвращает массив строк, состоящих из названий всех доступных шрифтов. Чтобы создать экземпляр класса `GraphicsEnvironment`, описывающего графическую среду, вызывается статический метод `getLocalGraphicsEnvironment()`. Таким образом, приведенная ниже краткая программа выводит названия всех шрифтов, доступных в отдельной системе.

```
import java.awt.*;
public class ListFonts
{
    public static void main(String[] args)
    {
        String[] fontNames = GraphicsEnvironment
```

```
.getLocalGraphicsEnvironment()  
.getAvailableFontFamilyNames();  
for (String fontName : fontNames)  
    System.out.println(fontName);  
}  
}
```

В одной из систем список шрифтов начинается такими строками:

```
Abadi MT Condensed Light  
Arial  
Arial Black  
Arial Narrow  
Arioso  
Baskerville  
Binner Gothic  
. . .
```

И так далее для следующих семидесяти шрифтов.

Названия шрифтов представляют собой торговые марки, а сами шрифты могут быть защищены авторскими правами. Распространение шрифтов часто сопряжено с оплатой. Разумеется, аналогично дешевым подделкам дорогих духов, существуют имитации фирменных шрифтов. Например, имитация шрифта Helvetica распространяется вместе с операционной системой Windows под названием Arial.

В качестве общего основания в библиотеке AWT приняты следующие пять *логических* названий шрифтов:

```
SansSerif  
Serif  
Monospaced  
Dialog  
DialogInput
```

Эти названия всегда приводятся к шрифтам, фактически существующим на отдельной клиентской машине. Например, в Windows шрифт SanSerif приводится к шрифту Arial. Кроме того, в JDK компании Oracle всегда входят три гарнитуры шрифтов под названием "Lucida Sans", "Lucida Bright" и "Lucida Sans Typewriter".

Для того чтобы отобразить букву заданным шрифтом, сначала нужно создать объект класса Font, а затем указать название шрифта, его стиль и размер. Ниже показано, каким образом создается объект класса Font.

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
```

Третий параметр задает размер шрифта. Для обозначения размера шрифта служит единица измерения, называемая *пунктом*. В одном дюйме содержится 72 пункта. В конструкторе класса Font вместо фактического названия начертания можно использовать логическое название шрифта. Затем нужно указать стиль (т.е. простой, **полужирный**, *курсив* или **полужирный курсив**), задав второй аргумент конструктора равным одному из следующих значений:

```
Font.PLAIN  
Font.BOLD  
Font.ITALIC  
Font.BOLD + Font.ITALIC
```



НА ЗАМЕТКУ! Приведение логических шрифтов к названиям физических шрифтов определено в файле `fontconfig.properties`, находящемся в подкаталоге `jre/lib` установки Java. Подробнее об этом файле можно узнать по адресу <http://docs.oracle.com/javase/7/docs/technotes/guides/intl/fontconfig.html>.

Файлы со шрифтами можно вводить в формате TrueType, OpenType или PostScript Type 1. С этой целью следует создать отдельный поток ввода конкретного шрифта — обычно из файла на диске или с веб-сайта по указанному адресу (потоки ввода-вывода будут подробнее рассматриваться в главе 1 второго тома данной книги). Затем нужно вызвать статический метод `Font.CreateFont()` следующим образом:

```
URL url = new URL("http://www.fonts.com/Wingbats.ttf");
InputStream in = url.openStream();
Font f1 = Font.createFont(Font.TRUETYPE_FONT, in);
```

Этот шрифт имеет простое начертание и размер 1 пункт. Чтобы получить шрифт нужного размера, следует вызвать метод `deriveFont()`, как показано ниже.

```
Font df = f.deriveFont(14.0F);
```



ВНИМАНИЕ! Существуют две перегружаемые версии метода `deriveFont()`. В одной из них (с параметром типа `float`) задается размер шрифта, а в другой (с параметром типа `int`) — стиль шрифта. Таким образом, при вызове `f.deriveFont(14)` задается стиль, а не размер шрифта! (В результате будет установлен курсив, поскольку двоичное представление числа 14 содержит единицу в разряде, соответствующем константе `ITALIC`, но не `BOLD`.)

В Java шрифты состоят из букв, цифр, знаков препинания и ряда других символов. Так, если вывести шрифтом Dialog символ `'\u2297'`, на экране появится знак **U**. Доступными являются только символы, указанные в наборе символов уникада. Ниже приведен фрагмент кода для вывода на экран символьной строки "Hello, World", набранной полужирным шрифтом SanSerif размером 14 пунктов.

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(sansbold14);
String message = "Hello, World!";
g2.drawString(message, 75, 100);
```

А теперь требуется выровнять текстовую строку по *центру* компонента. Для этого нужно знать ширину и высоту строки в пикселях. Процесс выравнивания зависит от трех факторов.

- Используемый шрифт (в данном случае полужирный шрифт sans serif размером 14 пунктов).
- Символы строки (в данном случае "Hello, World!").
- Устройство, на котором будет отображаться строка (в данном случае экран монитора пользователя).

Чтобы получить объект, представляющий характеристики устройства вывода, следует вызвать метод `getFontRenderContext()` из класса `Graphics2D`. Он возвращает объект класса `FontRenderContext`. Этот объект нужно передать методу `getStringBounds()` из класса `Font`, как показано ниже. А метод `getStringBounds()` возвращает прямоугольник, ограничивающий текстовую строку.

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);
```

Чтобы выяснить, от чего зависят размеры этого прямоугольника, следует рассмотреть некоторые основные термины, применяемые при текстовом наборе (рис. 7.13). *Базовая линия* — это воображаемая линия, которая касается снизу таких символов, как **e**. *Подъем* — максимальное расстояние от базовой линии до верхушек *надстрочных элементов*, например, верхнего края буквы **b** или **k**. *Спуск* — это расстояние от базовой линии до *подстрочного элемента*, например, нижнего края буквы **p** или **g**.

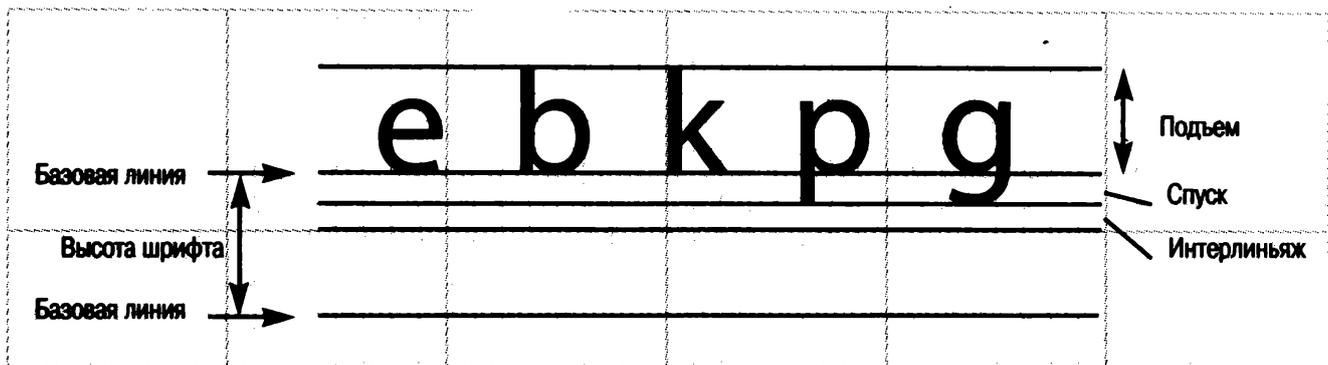


Рис. 7.13. Основные термины, применяемые при формировании текстовой строки

Интерлиньяж — это разность между спуском предыдущей строки и подъемом следующей. *Высота шрифта* — это расстояние между соседними базовыми линиями, равное сумме спуска, подъема и интерлиньяжа.

Ширина прямоугольника, возвращаемого методом `getStringBounds()`, задает протяженность строки по горизонтали. Высота прямоугольника равна сумме спуска, подъема и интерлиньяжа. Начало отсчета прямоугольника находится на базовой линии строки. Координата *y* отсчитывается от базовой линии. Для верхней части прямоугольника она является отрицательной. Таким образом, ширину, высоту и подъем строки можно вычислить следующим образом:

```
double stringWidth = bounds.getWidth();
double stringHeight = bounds.getHeight();
double ascent = -bounds.getY();
```

Если же требуется выяснить, чему равен интерлиньяж или спуск, то следует вызвать метод `getLineMetrics()` из класса `Font`, как показано ниже. Этот метод возвращает объект класса `LineMetrics`, у которого имеются методы для определения указанных выше характеристик.

```
LineMetrics metrics = f.getLineMetrics(message, context);
float descent = metrics.getDescent();
float leading = metrics.getLeading();
```

В приведенном ниже фрагменте эти данные используются для выравнивания строки по центру содержащего ее компонента.

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);
```

```
// определить координаты (x, y) верхнего левого угла текста
double x = (getWidth() - bounds.getWidth()) / 2;
double y = (getHeight() - bounds.getHeight()) / 2;
```

```
// сложить подъем с координатой y, чтобы достигнуть базовой линии
```

```
double ascent = -bounds.getY();
double baseY = y + ascent;
g2.drawString(message, (int) x, (int) baseY);
```

Для того чтобы стало понятнее, каким образом происходит выравнивание текста по центру, следует иметь в виду, что метод `getWidth()` возвращает ширину компонента. Часть этой ширины, определяемую при вызове `bounds.getWidth()`, занимает строка выводимого сообщения. Оставшееся пространство нужно равномерно распределить по обеим сторонам строки. Следовательно, размер незаполненного пространства с каждой стороны должен равняться половине разности между шириной компонента и длиной строки. Этими же соображениями следует руководствоваться при выравнивании текстовой строки по высоте.



НА ЗАМЕТКУ! Когда требуется вычислить размеры компонента текста за пределами метода `paintComponent()`, контекст воспроизведения шрифта нельзя получить из объекта типа `Graphics2D`. В этом случае нужно вызвать сначала метод `getFontMetrics()` из класса `JComponent`, а затем метод `getFontRenderContext()` следующим образом:

```
FontRenderContext context = getFontMetrics(f).getFontRenderContext();
```

Для того чтобы показать, насколько правильно расположена текстовая строка, в рассматриваемом здесь примере программы отображаются базовая линия и ограничивающий прямоугольник. На рис. 7.14 приведен результат выполнения этой программы на экране, а ее исходный код — в листинге 7.5.

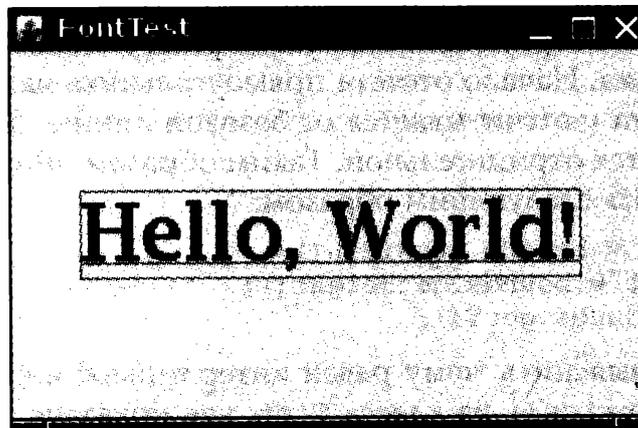


Рис. 7.14. Тестовая строка, отображаемая на экране с базовой линией и ограничивающим прямоугольником

Листинг 7.5. Исходный код из файла `font/FontTest.java`

```
1 package font;
2
3 import java.awt.*;
4 import java.awt.font.*;
5 import java.awt.geom.*;
6 import javax.swing.*;
7
8 /**
9  * @version 1.33 2007-04-14
10 * @author Cay Horstmann
11 */
```

```
12 public class FontTest
13 {
14     public static void main(String[] args)
15     {
16         EventQueue.invokeLater(new Runnable()
17         {
18             public void run()
19             {
20                 JFrame frame = new FontFrame();
21                 frame.setTitle("FontTest");
22                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23                 frame.setVisible(true);
24             }
25         });
26     }
27 }
28 /**
29  * Фрейм с компонентом текстового сообщения
30  */
31 class FontFrame extends JFrame
32 {
33     public FontFrame()
34     {
35         add(new FontComponent());
36         pack();
37     }
38 }
39
40 /**
41  * Компонент, отображающий текстовое сообщение, выровненное
42  * по центру в прямоугольной рамке
43  */
44 class FontComponent extends JComponent
45 {
46     private static final int DEFAULT_WIDTH = 300;
47     private static final int DEFAULT_HEIGHT = 200;
48
49     public void paintComponent(Graphics g)
50     {
51         Graphics2D g2 = (Graphics2D) g;
52
53         String message = "Hello, World!";
54
55         Font f = new Font("Serif", Font.BOLD, 36);
56         g2.setFont(f);
57
58         // определить размеры текстового сообщения
59         FontRenderContext context = g2.getFontRenderContext();
60         Rectangle2D bounds = f.getStringBounds(message, context);
61
62         // определить координаты (x, y) верхнего левого угла текста
63         double x = (getWidth() - bounds.getWidth()) / 2;
64         double y = (getHeight() - bounds.getHeight()) / 2;
65
66         // сложить подъем с координатой y, чтобы достигнуть базовой линии
67         double ascent = -bounds.getY();
68         double baseY = y + ascent;
69
70         // вывести текстовое сообщение
71         g2.drawString(message, (int) x, (int) baseY);
```

```

72
73     g2.setPaint(Color.LIGHT_GRAY);
74
75     // нарисовать базовую линию
76     g2.draw(new Line2D.Double(x, baseY, x + bounds.getWidth(), baseY));
77     // нарисовать ограничивающий прямоугольник
78     Rectangle2D rect =
79         new Rectangle2D.Double(x, y, bounds.getWidth(), bounds.getHeight());
80     g2.draw(rect);
81 }
82
83 public Dimension getPreferredSize()
84     { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
85 }

```

java.awt.Font 1.0

- `Font(String name, int style, int size)`
Создает новый объект типа `Font` для описания шрифта.
Параметры: *name* Название шрифта, т.е. название начертания (например, Helvetica Bold) или логическое название шрифта (Serif или SansSerif)
style Стиль шрифта (Font.PLAIN, Font.BOLD, Font.ITALIC или Font.BOLD + Font.ITALIC)
size Размер шрифта (например, 12 пунктов)
- `String getFontName()`
Возвращает название начертания шрифта (например, Helvetica Bold).
- `String getFamily()`
Возвращает название гарнитуры шрифтов (например, Helvetica).
- `String getName()`
Возвращает логическое название шрифта (например, SansSerif), если оно присвоено шрифту при его создании; в противном случае — название начертания шрифта.
- `Rectangle2D getStringBounds(String s, FontRenderContext context) 1.2`
Возвращает прямоугольник, ограничивающий данную строку. Координата *y* прямоугольника отсчитывается от базовой линии. Координата *y* верхней части прямоугольника равна отрицательному подъему. Высота прямоугольника равна сумме подъема, спуска и интерлиньяжа. Ширина прямоугольника равна ширине строки.
- `LineMetrics getLineMetrics(String s, FontRenderContext context) 1.2`
Возвращает объект типа `LineMetrics`, описывающий типографские характеристики текстовой строки, чтобы определить ее протяженность.
- `Font deriveFont(int style) 1.2`
- `Font deriveFont(float size) 1.2`
- `Font deriveFont(int style, float size) 1.2`
Возвращают новый объект типа `Font` для описания шрифта, совпадающего с текущим шрифтом, за исключением размера и стиля, задаваемых в качестве параметров.

java.awt.font.LineMetrics 1.2

- `float getAscent()`
Получает подъем шрифта — расстояние от базовой линии до верхушек прописных букв.
- `float getDescent()`
Получает спуск — расстояние от базовой линии до подстрочных элементов букв.
- `float getLeading()`
Получает интерлиньяж — расстояние от нижнего края предыдущей строки до верхнего края следующей строки.
- `float getHeight()`
Получает общую высоту шрифта — расстояние между двумя базовыми линиями текста, равное сумме спуска, интерлиньяжа и подъема.

java.awt.Graphics 1.0

- `Font getFont()`
- `void setFont(Font font)`
Получают или устанавливают шрифт в текущем контексте. Этот шрифт будет использоваться в последующих операциях отображения текста.
Параметры: `font` Заданный шрифт
- `void drawString(String str, int x, int y)`
Выводит текстовую строку, выделенную текущим шрифтом и цветом.
Параметры: `str` Выводимая текстовая строка
`x` Координата `x` начала строки
`y` Координата `y` базовой линии строки

java.awt.Graphics2D 1.2

- `FontRenderContext getFontRenderContext()`
Получает контекст воспроизведения, в котором задаются характеристики шрифта для текущего графического контекста.
- `void drawString(String str, float x, float y)`
Выводит текстовую строку, выделенную текущим шрифтом и цветом.
Параметры: `str` Выводимая текстовая строка
`x` Координата `x` начала строки
`y` Координата `y` базовой линии строки

javax.swing.JComponent 1.2

- `FontMetrics getFontMetrics(Font f)` 5.0

Получает типографские характеристики заданного шрифта. Класс `FontMetrics` является предшественником класса `LineMetrics`.

java.awt.FontMetrics 1.0

- `FontRenderContext getFontRenderContext()` 1.2

Получает контекст воспроизведения для шрифта.

Вывод изображений

Как было показано ранее, простые изображения образуются рисованием линий и форм. А сложные изображения вроде фотографических обычно формируются внешними средствами, например, при сканировании или обработке в специальных графических редакторах. (Из второго тома данной книги вы узнаете, что изображения можно также формировать по отдельным пикселям.)

Если изображения хранятся в файлах на компьютере или в Интернете, их можно ввести, а затем отобразить на экране с помощью объектов класса `Graphics`. Ввести изображения можно разными способами. Для этой цели можно, например, воспользоваться упоминавшимся ранее файлом `ImageIcon` следующим образом:

```
Image image = new ImageIcon(filename).getImage();
```

Теперь переменная `image` содержит ссылку на объект, инкапсулирующий данные изображения. Используя этот объект, изображение можно далее вывести на экран с помощью метода `drawImage()` из класса `Graphics`, как показано ниже.

```
public void paintComponent(Graphics g)
{
    . . .
    g.drawImage(image, x, y, null);
}
```

Программа из листинга 7.6 делает немного больше, многократно выводя указанное изображение в окне рядами. Результат выполнения этой программы приведен на рис. 7.15. Вывод изображения рядами осуществляется с помощью метода `paintComponent()`. Сначала одна копия изображения воспроизводится в левом верхнем углу окна, а затем вызывается метод `copyArea()`, который копирует его по всему окну, как показано ниже.

```
for (int i = 0; i * imageWidth <= getWidth(); i++)
    for (int j = 0; j * imageHeight <= getHeight(); j++)
        if (i + j > 0)
            g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth,
                j * imageHeight);
```

В листинге 7.6 приведен весь исходный код программы для вывода изображений.

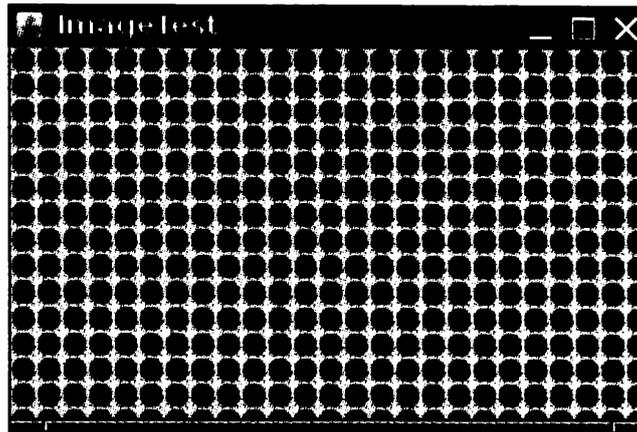


Рис. 7.15. Окно с воспроизводимым рядом графическим изображением

Листинг 7.6. Исходный код из файла `image/ImageTest.java`

```
1 package image;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * @version 1.33 2007-04-14
8  * @author Cay Horstmann
9  */
10 public class ImageTest
11 {
12     public static void main(String[] args)
13     {
14         EventQueue.invokeLater(new Runnable()
15         {
16             public void run()
17             {
18                 JFrame frame = new ImageFrame();
19                 frame.setTitle("ImageTest");
20                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21                 frame.setVisible(true);
22             }
23         });
24     }
25 }
26
27 /**
28  * Фрейм с компонентом изображения
29  */
30 class ImageFrame extends JFrame
31 {
32     public ImageFrame()
33     {
34         add(new ImageComponent());
35         pack();
36     }
37 }
38
```

```

39 /**
40  * Компонент, выводющий изображение рядами
41  */
42 class ImageComponent extends JComponent
43 {
44     private static final int DEFAULT_WIDTH = 300;
45     private static final int DEFAULT_HEIGHT = 200;
46
47     private Image image;
48
49     public ImageComponent()
50     {
51         image = new ImageIcon("blue-ball.gif").getImage();
52     }
53
54     public void paintComponent(Graphics g)
55     {
56         if (image == null) return;
57
58         int imageWidth = image.getWidth(this);
59         int imageHeight = image.getHeight(this);
60
61         // воспроизвести изображение в левом верхнем углу
62         g.drawImage(image, 0, 0, null);
63
64         // воспроизвести изображение рядами по всему компоненту
65
66         for (int i = 0; i * imageWidth <= getWidth(); i++)
67             for (int j = 0; j * imageHeight <= getHeight(); j++)
68                 if (i + j > 0)
69                     g.copyArea(0, 0, imageWidth, imageHeight,
70                               i * imageWidth, j * imageHeight);
71     }
72
73     public Dimension getPreferredSize()
74     { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
75 }

```

java.awt.Graphics 1.0

- boolean drawImage(Image img, int x, int y, ImageObserver observer)

Выводит немасштабированное изображение. *Примечание:* этот метод может вернуть управление до того, как изображение будет выведено полностью.

Параметры:	<i>img</i>	Выводимое изображение
	<i>x</i>	Координата <i>x</i> левого верхнего угла
	<i>y</i>	Координата <i>y</i> левого верхнего угла
	<i>observer</i>	Объект для уведомления о ходе воспроизведения (может быть пустой ссылкой типа null)

- `boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)`

Выводит масштабированное изображение. Система масштабирует изображение, чтобы уместить его в области заданной ширины и высоты. *Примечание:* этот метод может вернуть управление до того, как изображение будет выведено полностью.

Параметры:	<i>img</i>	Выводимое изображение
	<i>x</i>	Координата <i>x</i> левого верхнего угла
	<i>y</i>	Координата <i>y</i> левого верхнего угла
	<i>width</i>	Требуемая ширина изображения
	<i>height</i>	Требуемая высота изображения
	<i>observer</i>	Объект для уведомления о ходе воспроизведения (может быть пустой ссылкой типа <code>null</code>)

- `void copyArea(int x, int y, int width, int height, int dx, int dy)`
Копирует область экрана.

Параметры:	<i>x</i>	Координата <i>x</i> левого верхнего угла исходной области
	<i>y</i>	Координата <i>y</i> левого верхнего угла исходной области
	<i>width</i>	Ширина исходной области
	<i>height</i>	Высота исходной области
	<i>dx</i>	Расстояние по горизонтали от исходной до целевой области
	<i>dy</i>	Расстояние по вертикали от исходной до целевой области

На этом введение в программирование графики на Java завершено. Более сложные приемы программирования двумерной графики и манипулирования изображениями будут изложены во втором томе. В следующей главе речь пойдет о том, как заставить программу реагировать на ввод данных пользователем.

Компоненты пользовательского интерфейса в Swing

В этой главе...

- ▶ Библиотека Swing и шаблон проектирования “модель–представление–контроллер”
- ▶ Введение в компоновку пользовательского интерфейса
- ▶ Ввод текста
- ▶ Компоненты для выбора разных вариантов
- ▶ Меню
- ▶ Расширенные средства компоновки
- ▶ Диалоговые окна

Предыдущая глава была в основном посвящена рассмотрению модели обработки событий Java. Проработав ее материал, вы приобрели знания и навыки, без которых невозможно создать приложение с ГПИ. А в этой главе будут рассмотрены самые важные инструментальные средства, требующиеся для создания полнофункциональных ГПИ.

Сначала будут вкратце рассмотрены архитектурные принципы, положенные в основу библиотеки Swing. Для того чтобы эффективно пользоваться современными компонентами пользовательского интерфейса, нужно как следует разбираться в их функционировании. Затем будет показано, как применять обычные компоненты пользовательского интерфейса из библиотеки Swing, включая текстовые поля,

кнопки-переключатели и меню. Далее вы ознакомитесь с тем, как пользоваться возможностями диспетчеров компоновки в Java, чтобы размещать компоненты в окне независимо от визуального стиля ГПИ. И в заключение будет показано, каким образом диалоговые окна создаются средствами Swing.

В этой главе описываются основные компоненты библиотеки Swing — текстовые поля, кнопки и полосы прокрутки. Это самые важные и наиболее употребительные компоненты пользовательского интерфейса. А более сложные компоненты будут рассматриваться во втором томе данной книги.

Библиотека Swing и шаблон проектирования “модель–представление–контроллер”

Итак, начнем эту главу с раздела, в котором описывается архитектура компонентов библиотеки Swing. Сначала мы обсудим понятие *шаблонов проектирования*, а затем обратимся непосредственно к шаблону “модель–представление–контроллер”, который оказал огромное влияние на архитектуру Swing.

Шаблоны проектирования

Решая конкретную задачу, большинство людей руководствуются своим опытом или спрашивают совета у других. Шаблоны проектирования — это способ представления накопленного опыта в структурированном виде.

В последние годы разработчики программного обеспечения стали собирать каталоги таких шаблонов. Пионеров этой области вдохновили шаблоны проектирования, созданные архитектором Кристофером Александером (Christopher Alexander). В своей книге *Timeless Way of Building* (Строительство на века; изд-во Oxford University Press, 1979 г.) он составил каталог шаблонов для проектирования общественных и частных жилых зданий.

Рассмотрим место у окна в качестве характерного примера шаблона из области строительства. Всем нравится сидеть в удобном кресле на балконе, у эркера или большого окна с низким подоконником. В комнате, где нет такого места, редко чувствуешь себя уютно. В такой комнате человеку хочется удовлетворить два желания: сидеть удобно и сидеть лицом к свету.

Разумеется, если удобные места, т.е. места, где вам больше всего хотелось бы сесть, находятся далеко от окна, то разрешить этот конфликт невозможно. Следовательно, в каждой комнате, где человек проводит продолжительное время, должно быть, по крайней мере, одно окно в подходящем для этого месте (рис. 9.1).

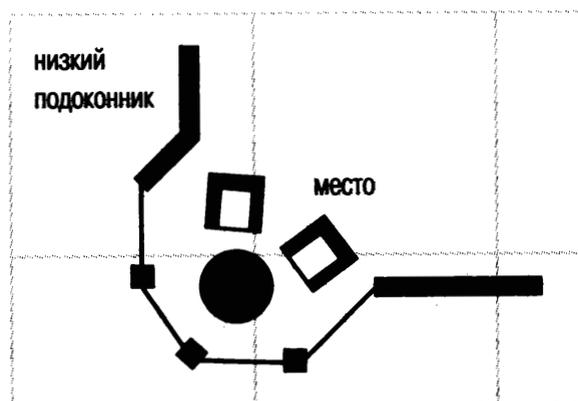


Рис. 9.1. Место у окна

Каждый шаблон в каталоге Александера, как и в каталоге шаблонов программного обеспечения, следует определенному формату. Сначала описывается контекст, т.е. ситуация, которая порождает задачу проектирования. Затем ставится задача, обычно в виде конфликтующих между собой ограничений. И в заключение предлагается ее решение, в котором достигается компромисс между противоречивыми требованиями.

В приведенном выше примере шаблона контекстом является комната, в которой человек проводит значительную часть дня. Конфликтующие ограничения заключаются в его желании сидеть удобно и лицом к свету. Решение заключается в создании “места у окна”.

В шаблоне проектирования “модель–представление–контроллер” контекстом является система пользовательского интерфейса, которая представляет информацию и получает данные от пользователя. В ней существует несколько противоречий. К ним могут относиться разные визуальные представления одних и тех же данных, которые должны одновременно обновляться. Визуальное представление может изменяться в зависимости от стиля пользовательского интерфейса. Механизмы взаимодействия также могут изменяться, например, для того, чтобы поддерживать выполнение команд, которые вводятся голосом. Решение этой задачи заключается в распределении ответственности между тремя разными взаимодействующими составляющими шаблона: моделью, представлением и контроллером.

Шаблон проектирования “модель–представление–контроллер” — далеко не единственный шаблон проектирования, применяемый в библиотеках AWT и Swing. Ниже перечислен ряд примеров других шаблонов.

- Контейнеры и компоненты служат примерами шаблона “Компоновщик”.
- Прокручиваемая панель — пример шаблона “Декоратор”.
- Диспетчеры компоновки следуют шаблону “Стратегия”.

Следует особо подчеркнуть, что шаблоны проектирования стали частью общей культуры программирования. Программисты в любой части света знают, что такое шаблон “модель–представление–контроллер” и что собой представляет шаблон “Декоратор”. Таким образом, шаблоны стали эффективным средством для описания задач проектирования программного обеспечения.

Формальное описание многочисленных полезных программных шаблонов можно найти в академической книге Эриха Гаммы (Erich Gamma) *Design Patterns — Elements of Reusable Object-Oriented Software* (издательство Addison-Wesley, 1995 г.; в русском переводе книга вышла под названием *Приемы объектно-ориентированного проектирования. Паттерны проектирования*). Кроме того, рекомендуем прочитать блестящую книгу Франка Бушмана (Frank Buschmann) *A System of Patterns* (Система шаблонов; издательство John Wiley & Sons, 1996 г.), которая менее академична и более доступна для широкого круга читателей.

Шаблон проектирования “модель–представление–контроллер”

Напомним, из чего состоят компоненты ГПИ, например, кнопка, флажок, текстовое поле или сложное окно управления древовидной структурой элементов. Каждый из этих компонентов обладает следующими характеристиками.

- *Содержимое*, например, состояние кнопки (нажата или отпущена) или текст в поле редактирования.

- *Внешний вид* (цвет, размер и т.д.).
- *Поведение* (реакция на события).

Даже у таких, на первый взгляд, простых компонентов, как кнопки, эти характеристики тесно связаны между собой. Очевидно, что внешний вид кнопки зависит от визуального стиля интерфейса в целом. Кнопка в стиле Metal отличается от кнопки в стиле Windows или Motif. Кроме того, внешний вид зависит от состояния кнопки: в нажатом состоянии кнопка должна выглядеть иначе, чем в отпущенном. Состояние, в свою очередь, зависит от событий. Если пользователь щелкнул на кнопке, она считается нажатой.

Разумеется, когда вы используете кнопку в своих программах, то рассматриваете ее как таковую, не особенно вдаваясь в подробности ее внутреннего устройства и характеристик. В конце концов, технические подробности — удел программиста, реализовавшего эту кнопку. Но программисты, реализующие кнопки и прочие элементы ГПИ, должны тщательно обдумывать их внутреннее устройство и функционирование, чтобы все эти компоненты правильно работали независимо от выбранного визуального стиля.

Для решения подобных задач разработчики библиотеки Swing обратились к хорошо известному шаблону проектирования “модель–представление–контроллер” (Model-View-Controller — MVC). В основу этого шаблона положены принципы ООП, описанные в главе 5, один из которых гласит: не перегружать объект слишком большим количеством задач. Класс, предусмотренный для кнопки, не обязан делать сразу все. Поэтому стиль компонента связывается с одним объектом, а его содержимое — с другим. Шаблон проектирования “модель–представление–контроллер” позволяет достичь этой цели, реализовав три отдельных класса.

- Класс *модели*, в котором хранится содержимое.
- Класс *представления*, который отображает содержимое.
- Класс *контроллера*, обрабатывающий вводимые пользователем данные.

Шаблон проектирования “модель–представление–контроллер” точно обозначает взаимодействие этих классов. Модель хранит содержимое и не реализует пользовательский интерфейс. Содержимое кнопки тривиально — это небольшой набор признаков, означающих, нажата кнопка или отпущена, активизирована или неактивизирована и т.д. Содержимым текстового поля является символьная строка, не совпадающая с представлением. Так, если содержимое превышает длину текстового поля, пользователь видит лишь часть отображаемого текста (рис. 9.2).

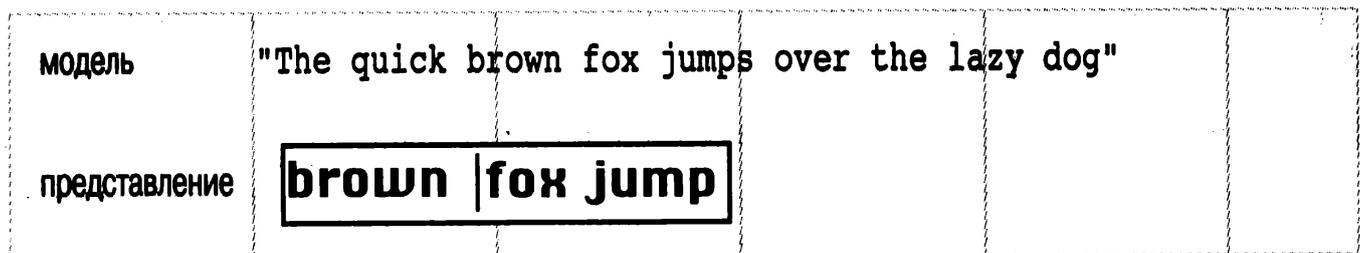


Рис. 9.2. Модель и представление текстового поля

Модель должна реализовывать методы, изменяющие содержимое и раскрывающие его смысл. Например, модель текстового поля имеет методы для ввода символов в строке, их удаления и возврата текста в виде строки. Следует иметь в виду, что модель совершенно невидима. Отображать данные, хранящиеся в модели, — задача представления.



НА ЗАМЕТКУ! Термин *модель*, по-видимому, выбран не совсем удачно, поскольку его часто связывают со способом представления абстрактного понятия. Например, авиаконструкторы и конструкторы автомобилей строят модели для того, чтобы имитировать настоящие самолеты и автомобили. Но эта аналогия не подходит к шаблону “модель–представление–контроллер”. В данном случае модель хранит содержимое, а представление отвечает за ее полное или неполное визуальное отображение. Намного более точной аналогией является натурщица, позирующая художнику. Художник должен смотреть на модель и создавать ее изображение. В зависимости от стиля, в котором работает художник, представление может оказаться классическим портретом, картиной в стиле импрессионизма или совокупностью фигур в стиле кубизма.

Одно из преимуществ шаблона “модель–представление–контроллер” состоит в том, что модель может иметь несколько представлений, каждое из которых отражает отдельный аспект ее содержимого. Например, редактор HTML-разметки документов часто предлагает одновременно *два* представления одних и тех же данных: WYSIWYG (Что видишь на экране, то и получишь при печати) и ряд дескрипторов (рис. 9.3). Когда контроллер обновляет модель, изменяются оба представления. Получив уведомление об изменении, представление обновляется автоматически. Разумеется, для таких простых компонентов пользовательского интерфейса, как кнопки, нет никакой необходимости предусматривать несколько представлений одной и той же модели.

Контроллер обрабатывает события, связанные с поступающей от пользователя информацией, например, щелчки кнопками мыши и нажатия клавиш, а затем решает, преобразовывать ли эти события в изменения модели или представления. Так, если пользователь нажмет клавишу символа при вводе в текстовом поле, контроллер вызовет из модели команду “вставить символ”. Затем модель уведомит представление обновить изображение. Представлению вообще неизвестно, почему изменился текст. Но если пользователь нажал клавишу управления курсором, то контроллер может отдать представлению команду на прокрутку. Прокрутка не изменяет текст, поэтому модели ничего неизвестно об этом событии. Взаимодействие модели, представления и контроллера схематически показано на рис. 9.4.

Вам как программисту, пользующемуся компонентами Swing, совсем не обязательно вникать во внутреннее строение шаблона “модель–представление–контроллер”. У каждого компонента пользовательского интерфейса имеется класс-оболочка (например, `JButton` или `JTextField`), в котором хранятся модель и ее представление. Если вас интересует содержимое (например, текст в соответствующем поле), класс-оболочка запросит модель и возвратит ответ. А если вас интересует изменение внешнего вида (например, положение курсора в текстовом поле), то класс-оболочка направит соответствующий запрос представлению. Но иногда класс-оболочка не полностью справляется с задачей выдачи нужных команд. И тогда приходится обращаться к нему, чтобы извлечь модель и работать непосредственно с ней. (Работать непосредственно с представлением необязательно — это задача кода, задающего визуальный стиль.)

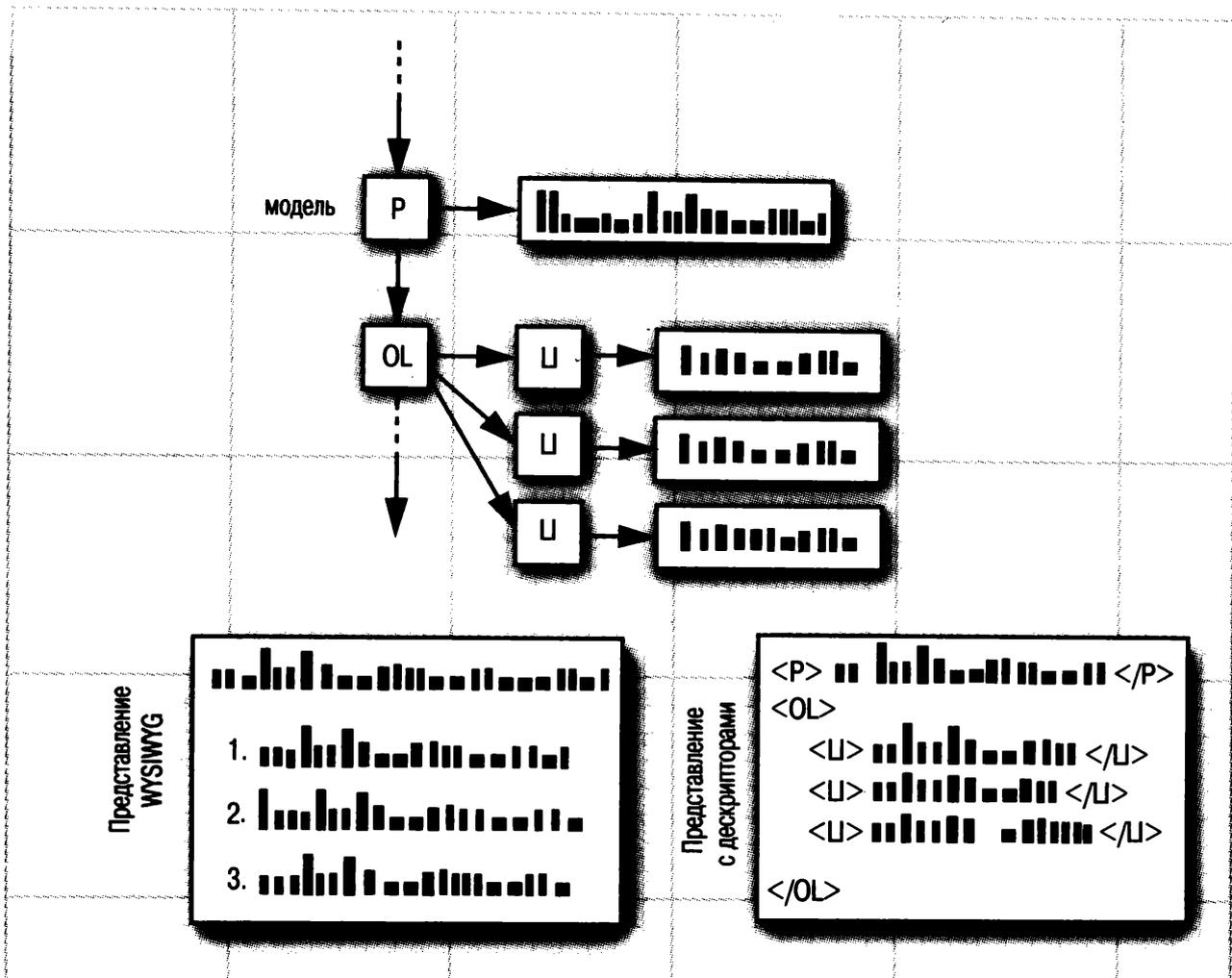


Рис. 9.3. Два разных представления одной и той же модели

Помимо того, что шаблон “модель–представление–контроллер” выполняет возложенные на него задачи, он оказался очень привлекательным для разработчиков Swing еще и потому, что облегчает поддержку различных стилей пользовательского интерфейса. Модель кнопки или текстового поля не зависит от стиля, но от него полностью зависит визуальное представление компонента. Контроллер также может изменяться. Например, в устройствах, управляемых голосом, контроллер должен обрабатывать ряд событий, резко отличающихся от стандартных событий, поступающих от клавиатуры и мыши. Отделяя модель от пользовательского интерфейса, разработчики Swing получили возможность повторно использовать код модели и даже менять визуальный стиль пользовательского интерфейса в ходе выполнения программы.

Разумеется, шаблоны — это лишь руководство к действию, а не догма. Нет такого шаблона, который был бы пригодным на все случаи жизни. Например, следовать шаблону “место у окна” может быть непросто, если требуется перестроить больничную палату. Аналогично разработчики Swing обнаружили, что на практике не всегда удается точно следовать шаблону “модель–представление–контроллер”. Модели легко разделяются, и каждый компонент пользовательского интерфейса имеет свой класс модели. Но задачи, стоящие перед представлением и контроллером, не всегда четко отделяются одна от другой, поэтому при распределении их между различными классами возникают трудности. Конечно, пользователю этих классов совершенно безразлично, как они реализованы. Как упоминалось выше, о моделях зачастую можно вообще не думать, а просто пользоваться классами оболочками.

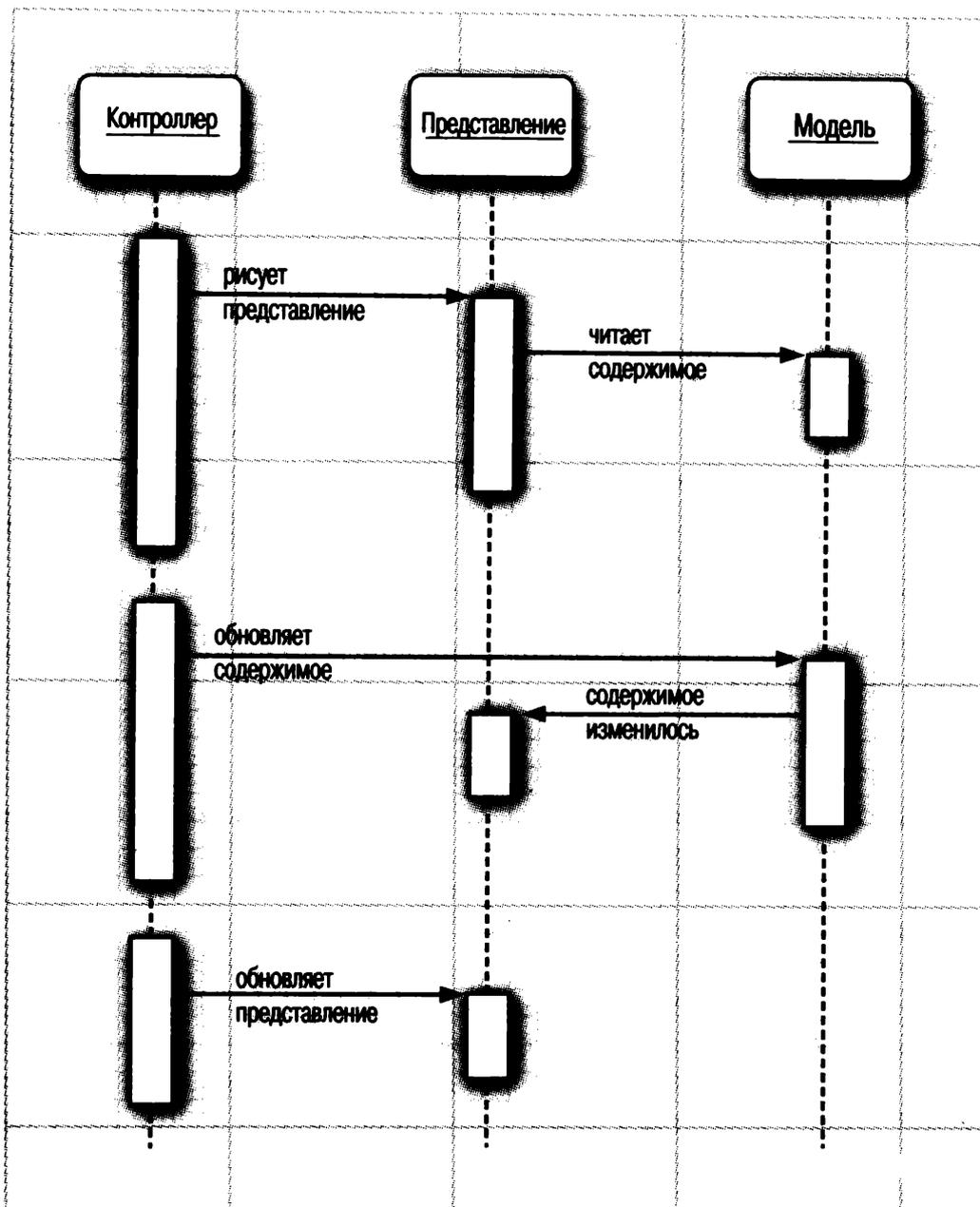


Рис. 9.4. Взаимодействие объектов модели, представления и контроллера

Анализ кнопок в Swing по шаблону “модель–представление–контроллер”

В примерах кода из предыдущей главы было показано, как пользоваться кнопками при построении ГПИ, но при этом ничего не говорилось о модели, представлении и контроллере. Кнопка — едва ли не самый простой элемент пользовательского интерфейса, и поэтому ее удобно выбрать в качестве наглядного пособия для изучения шаблона “модель–представление–контроллер”. Аналогичный подход применяется и к более сложным компонентам Swing.

Для большинства компонентов классы модели реализуют интерфейсы, имена которых оканчиваются словом `Model`. В частности, для кнопок используется интерфейс `ButtonModel`. Классы, реализующие этот интерфейс, могут определять состояние разнотипных кнопок. Кнопки настолько просты, что для них в библиотеке Swing предусмотрен отдельный класс `DefaultButtonModel`, реализующий данный интерфейс. Понять, какого рода данные поддерживаются в модели кнопки, можно, рассмотрев свойства интерфейса `ButtonModel` (табл. 9.1).

Таблица 9.1. Свойства интерфейса `ButtonModel`

Имя свойства	Значение
<code>actionCommand</code>	Символьная строка команды действия, связанного с кнопкой
<code>mnemonic</code>	Мнемоническое обозначение данной кнопки
<code>armed</code>	Логическое значение <code>true</code> , если кнопка была нажата, а курсор мыши еще находится на кнопке
<code>enabled</code>	Логическое значение <code>true</code> , если кнопка доступна
<code>pressed</code>	Логическое значение <code>true</code> , если кнопка была нажата, а кнопка мыши еще не отпущена
<code>rollover</code>	Логическое значение <code>true</code> , если курсор мыши находится на кнопке
<code>selected</code>	Логическое значение <code>true</code> , если кнопка включена (используется для флажков и кнопок-переключателей)

В каждом объекте типа `JButton` хранится объект модели кнопки, который можно извлечь оттуда следующим образом:

```
JButton button = new JButton("Blue");
ButtonModel model = button.getModel();
```

На практике подробности, касающиеся состояния кнопки, интересуют лишь представление, которое рисует ее на экране. Но из класса `JButton` можно извлечь и другую полезную информацию — в частности, заблокирована ли кнопка. (Для этого класс `JButton` запрашивает модель кнопки.)

Обратимся еще раз к интерфейсу `ButtonModel` и попробуем определить, что в нем *отсутствует*. Оказывается, что в модели *не* хранится название кнопки и ее пиктограмма. Поэтому анализ модели не позволяет судить о внешнем виде кнопки. (При реализации групп кнопок-переключателей, рассматриваемых далее в этой главе, данная особенность может стать источником серьезных осложнений для разработчика прикладной программы.)

Стоит также отметить, что одна и та же модель (а именно `DefaultButtonModel`) используется для поддержки нажимаемых кнопок, флажков кнопок-переключателей и даже для пунктов меню. Разумеется, каждая из этих разновидностей кнопок имеет свое собственное представление и отдельный контроллер. Если реализуется интерфейс в стиле `Metal`, то в качестве представления в классе `JButton` используется класс `BasicButtonUI`, а качестве контроллера — класс `ButtonUIListener`. В общем, у каждого компонента библиотеки `Swing` имеется связанный с ним объект представления, название которого заканчивается на `UI`. Но не у всех компонентов `Swing` имеется свой собственный объект контроллера.

Итак, прочитав это краткое введение в класс `JButton`, вы можете спросить: а что на самом деле представляет собой класс `JButton`? Это просто класс-оболочка, производный от класса `JComponent` и содержащий объект типа `DefaultButtonModel`, некоторые данные, необходимые для отображения (например, метку кнопки и ее пиктограмму), а также объект типа `BasicButtonUI`, реализующий представление кнопки.

Введение в компоновку пользовательского интерфейса

Прежде чем перейти к обсуждению таких компонентов Swing, как текстовые поля и кнопки-переключатели, рассмотрим вкратце, каким образом они размещаются во фрейме. В отличие от Visual Basic, в JDK отсутствует визуальный конструктор форм, и поэтому вам придется самостоятельно писать код для размещения компонентов пользовательского интерфейса во фрейме.

Разумеется, если вы программируете на Java в соответствующей ИСР, то в этой среде, скорее всего, предусмотрены средства, автоматизирующие некоторые из задач компоновки ГПИ. Несмотря на это, вы обязаны ясно представлять, каким образом размещаются компоненты. Ведь даже те ГПИ, которые автоматически построены с помощью самых совершенных инструментальных средств, обычно нуждаются в ручной доработке.

Вернемся для начала к примеру программы из предыдущей главы, где кнопки служили для изменения цвета фона во фрейме (рис. 9.5).

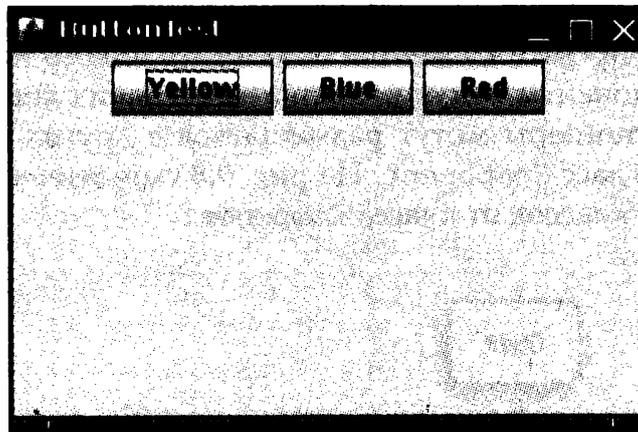


Рис. 9.5. Панель с тремя кнопками

Кнопки содержатся в объекте типа `JPanel` и управляются *диспетчером поточной компоновки* — стандартным диспетчером для компоновки панели. На рис. 9.6 показано, что происходит, когда на панели вводятся дополнительные кнопки. Как видите, если кнопки не помещаются в текущем ряду, они переносятся в новый ряд. Более того, кнопки будут отцентрованы на панели, даже если пользователь изменит размеры фрейма (рис. 9.7).

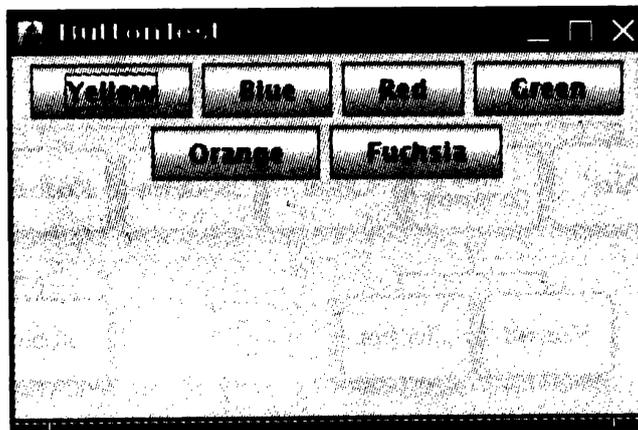


Рис. 9.6. Панель с шестью кнопками, расположенными подряд диспетчером поточной компоновки



Рис. 9.7. При изменении размеров панели автоматически изменяется расположение кнопок

В целом компоненты размещаются в контейнерах, а диспетчер компоновки определяет порядок расположения и размеры компонентов в контейнере. Классы кнопок, текстовых полей и прочих элементов пользовательского интерфейса расширяют класс `Component`. Компоненты могут размещаться в таких контейнерах, как панели. А поскольку одни контейнеры могут размещаться в других контейнерах, то класс `Container` расширяет класс `Component`. На рис. 9.8 схематически показана иерархия наследования всех этих классов от класса `Component`.

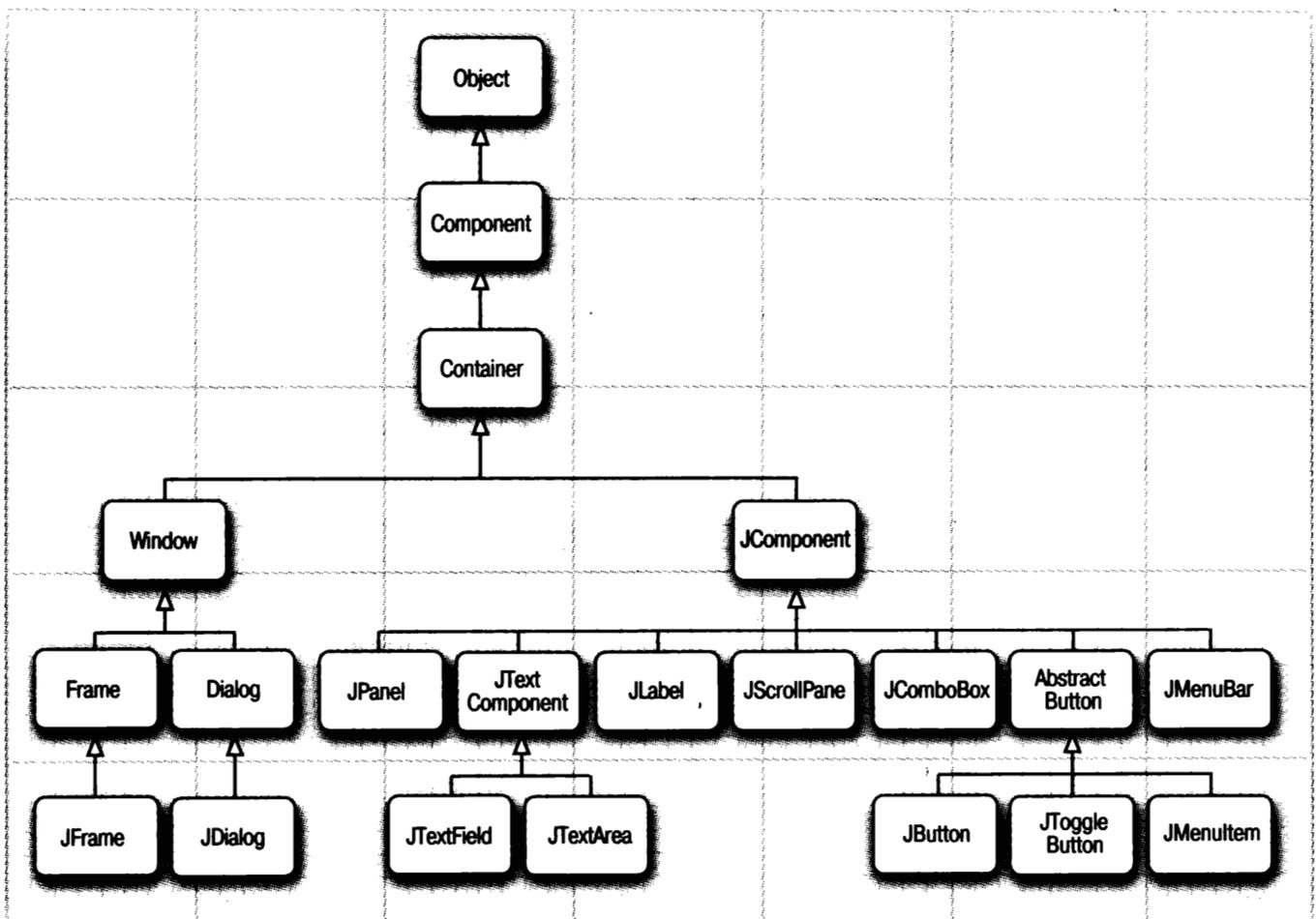


Рис. 9.8. Иерархия наследования от класса `Component`



НА ЗАМЕТКУ! К сожалению, иерархия наследования выглядит не совсем ясной по двум причинам. Во-первых, окна верхнего уровня, например типа `JFrame`, являются подклассами, производными от класса `Container`, а следовательно, и от класса `Component`, но их нельзя размещать в других контейнерах. Более того, класс `JComponent` является производным от класса `Container`, а не от класса `Component`, и поэтому другие компоненты можно добавлять в контейнер типа `JButton`. (Хотя эти компоненты и не будут отображаться.)

У каждого контейнера имеется свой диспетчер компоновки по умолчанию, но ничто не мешает установить свой собственный диспетчер компоновки. Например, в приведенной ниже строке кода класс `GridLayout` используется для размещения компонентов на панели. Когда компоненты вводятся в контейнер, метод `add()` контейнера принимает компонент и директивы по размещению, необходимые для диспетчера компоновки.

```
panel.setLayout(new GridLayout(4, 4));
```

java.awt.Container 1.0

- `void setLayout(LayoutManager m)`
Задает диспетчер компоновки для данного контейнера.
- `Component add(Component c)`
- `Component add(Component c, Object constraints) 1.1`
Вводят компонент в данный контейнер и возвращают ссылку на него.

Параметры: `c` Вводимый компонент
`constraints` Идентификатор, понятный диспетчеру компоновки

java.awt.FlowLayout 1.0

- `FlowLayout()`
- `FlowLayout(int align)`
- `FlowLayout(int align, int hgap, int vgap)`

Конструируют новый объект типа `FlowLayout`.

Параметры: `align` Выравнивание по левому (LEFT), правому (RIGHT) краю или по центру (CENTER)
`hgap` Расстояние между компонентами по горизонтали в пикселях (при отрицательных значениях компоненты перекрываются)
`vgap` Расстояние между компонентами по вертикали в пикселях (при отрицательных значениях компоненты перекрываются)

Граничная компоновка

Диспетчер граничной компоновки по умолчанию выбирается для панели содержимого, присутствующей в объекте типа `JFrame`. В отличие от диспетчера поточной компоновки, который полностью управляет расположением каждого компонента, диспетчер граничной компоновки позволяет выбрать место для каждого компонента. Компонент можно разместить в центре панели, в ее верхней или нижней части, а также слева или справа, как по сторонам света (рис. 9.9).

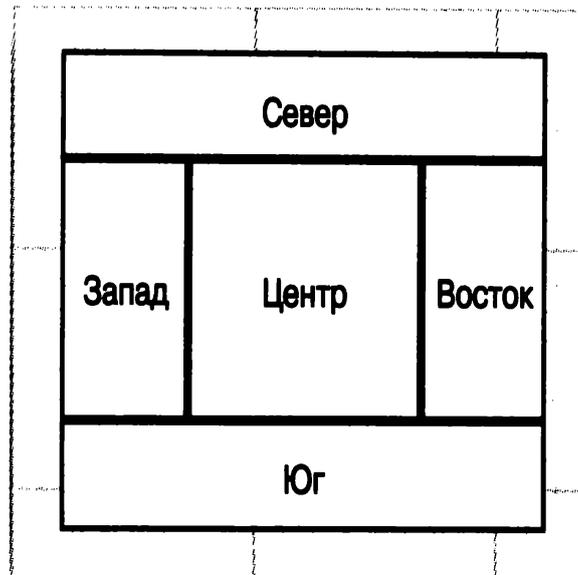


Рис. 9.9. Граничная компоновка

Например:

```
frame.add(component, BorderLayout.SOUTH);
```

При размещении компонентов сначала выделяется место по краям контейнера, а оставшееся свободное пространство считается центральной областью. При изменении размеров контейнера размеры компонентов, располагаемых по краям, остаются прежними, а изменяются лишь размеры центральной области. При вводе компонента на панели указываются константы `CENTER` (Центр), `NORTH` (Север), `SOUTH` (Юг), `EAST` (Восток) или `WEST` (Запад), определенные в классе `BorderLayout`. Занимать все места на панели совсем не обязательно. Если не указано никакого значения, то по умолчанию принимается `CENTER`, т.е. расположение по центру.



НА ЗАМЕТКУ! Константы в классе `BorderLayout` определены как символьные строки. Например, константа `BorderLayout.SOUTH` представляет собой символьную строку "South". Многие программисты предпочитают пользоваться непосредственно символьными строками, поскольку они короче, например `contentPane.add(component, "South")`. Но если вы случайно сделаете в такой строке опечатку, то компилятор не распознает ее как ошибку.

В отличие от поточной компоновки, при граничной компоновке все компоненты растягиваются, чтобы заполнить свободное пространство. (А при поточной компоновке предпочтительные размеры каждого компонента остаются без изменения.) Это может послужить препятствием к добавлению кнопки, как показано ниже.

```
frame.add(yellowButton, BorderLayout.SOUTH); // Не рекомендуется!
```

На рис. 9.10 показано, что произойдет, если попытаться выполнить приведенную выше строку кода. Размеры кнопки увеличатся, и она заполнит всю нижнюю часть фрейма. Если же попытаться вставить в нижней части фрейма еще одну кнопку, она просто заменит предыдущую кнопку.

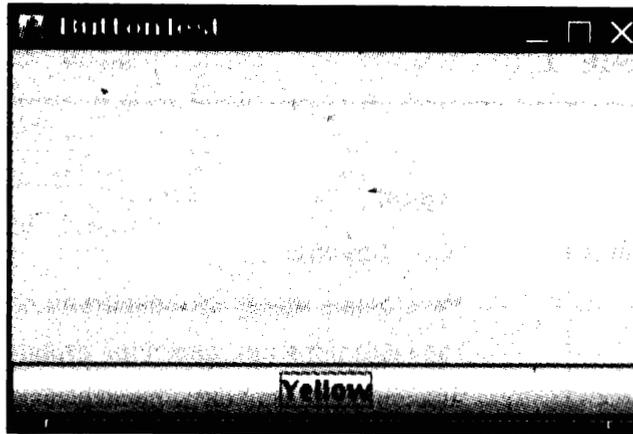


Рис. 9.10. Граничная компоновка одиночной кнопки

В качестве выхода из этого затруднительного положения можно воспользоваться дополнительными панелями. Обратите внимание на пример компоновки, приведенный на рис. 9.11. Все три кнопки в нижней части экрана находятся на одной панели, которая, в свою очередь, располагается в южной области панели содержимого фрейма.

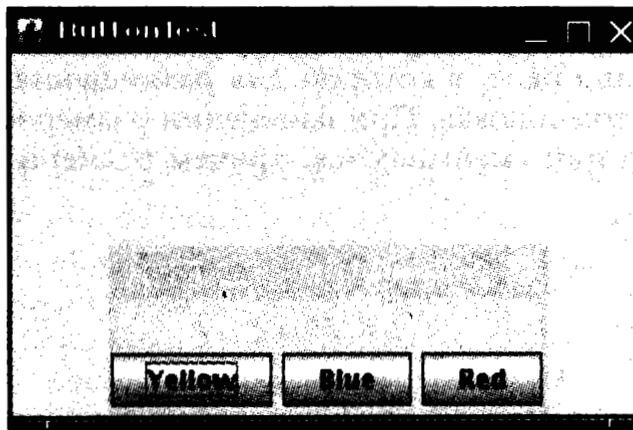


Рис. 9.11. Панель с тремя кнопками, располагаемая в южной области фрейма

Для того чтобы получить такое расположение, сначала создается новый объект типа `JPanel`, в который затем вводятся отдельные кнопки. Как упоминалось ранее, с обычной панелью по умолчанию связывается диспетчер поточной компоновки типа `FlowLayout`. В данном случае он вполне подходит. С помощью метода `add()` на панели размещаются отдельные кнопки, как было показано ранее. Расположение и размеры кнопок определяются диспетчером типа `FlowLayout`. Это означает, что кнопки будут выровнены по центру панели, а их размеры не будут увеличены для заполнения всего свободного пространства. И наконец, панель с тремя кнопками располагается в нижней части панели содержимого фрейма, как показано в приведенном ниже фрагменте кода. Граничная компоновка растягивает панель с тремя кнопками, чтобы она заняла всю нижнюю (южную) область фрейма.

```

JPanel panel = new JPanel();
panel.add(yellowButton);
panel.add(blueButton);
panel.add(redButton);
frame.add(panel, BorderLayout.SOUTH);

```

java.awt.BorderLayout 1.0

- BorderLayout()
- BorderLayout(int hgap, int vgap)

Конструирует новый объект типа BorderLayout.

Параметры:	<i>hgap</i>	Расстояние между компонентами по горизонтали в пикселях (при отрицательных значениях компоненты перекрываются)
	<i>vgap</i>	Расстояние между компонентами по вертикали в пикселях (при отрицательных значениях компоненты перекрываются)

Сеточная компоновка

При сеточной компоновке компоненты располагаются по рядам и столбцам, как в таблице. Но в этом случае размеры всех компонентов оказываются одинаковыми. На рис. 9.12 показано окно, в котором для размещения кнопок калькулятора применяется сеточная компоновка. При изменении размеров окна кнопки автоматически увеличиваются или уменьшаются, причем размеры всех кнопок остаются одинаковыми.



Рис. 9.12. Сеточная компоновка кнопок калькулятора

Требуемое количество рядов и столбцов указывается в конструкторе объекта типа GridLayout следующим образом:

```
panel.setLayout(new GridLayout(4, 4));
```

Компоненты вводятся построчно: сначала в первую ячейку первого ряда, затем во вторую ячейку первого ряда и так далее, как показано ниже.

В листинге 9.1 приведен исходный код программы, имитирующей работу калькулятора. Это обычный калькулятор. В нем не применяется так называемая *обратная*

польская запись, нашедшая необъяснимое распространение в учебной литературе по Java. После ввода во фрейм компонента в этой программе вызывается метод `pack()`, который сохраняет предпочтительные размеры компонентов и на их основании рассчитывает ширину и высоту фрейма.

На практике лишь в немногих программах применяется такая жесткая компоновка ГПИ, как на лицевой панели калькулятора, хотя небольшие сеточные компоновки (обычно из одного ряда или одного столбца) могут применяться для разбиения окна на равные части. Так, если в окне требуется разместить несколько кнопок одинакового размера, их следует ввести на панели с сеточной компоновкой в один ряд. Очевидно, что в конструкторе диспетчера сеточной компоновки нужно задать один ряд, а число столбцов должно равняться количеству кнопок.

Листинг 9.1. Исходный код из файла `calculator/CalculatorPanel.java`

```
1 package calculator;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Панель с кнопками и областью отображения результатов калькуляций
9  */
10 public class CalculatorPanel extends JPanel
11 {
12     private JButton display;
13     private JPanel panel;
14     private double result;
15     private String lastCommand;
16     private boolean start;
17
18     public CalculatorPanel()
19     {
20         setLayout(new BorderLayout());
21         result = 0;
22         lastCommand = "=";
23         start = true;
24         // ввести область отображения результатов калькуляций
25         display = new JButton("0");
26         display.setEnabled(false);
27         add(display, BorderLayout.NORTH);
28
29         ActionListener insert = new InsertAction();
30         ActionListener command = new CommandAction();
31
32         // ввести кнопки сеткой 4 x 4
33         panel = new JPanel();
34         panel.setLayout(new GridLayout(4, 4));
35
36         addButton("7", insert);
37         addButton("8", insert);
38         addButton("9", insert);
39         addButton("/", command);
40
41         addButton("4", insert);
```

```
42     addButton("5", insert);
43     addButton("6", insert);
44     addButton("*", command);
45
46     addButton("1", insert);
47     addButton("2", insert);
48     addButton("3", insert);
49     addButton("-", command);
50
51     addButton("0", insert);
52     addButton(".", insert);
53     addButton("=", command);
54     addButton("+", command);
55
56     add(panel, BorderLayout.CENTER);
57 }
58
59 /**
60  * Вводит кнопку на центральной панели
61  * @param label Метка кнопки
62  * @param listener Приемник действий кнопки
63  */
64 private void addButton(String label, ActionListener listener)
65 {
66     JButton button = new JButton(label);
67     button.addActionListener(listener);
68     panel.add(button);
69 }
70 /**
71  * При обработке событий строка действия кнопки вводится
72  * в конце отображаемого текста
73  */
74 private class InsertAction implements ActionListener
75 {
76     public void actionPerformed(ActionEvent event)
77     {
78         String input = event.getActionCommand();
79         if (start)
80         {
81             display.setText("");
82             start = false;
83         }
84         display.setText(display.getText() + input);
85     }
86 }
87
88 /**
89  * При обработке событий выполняется команда,
90  * указанная в строке действия кнопки
91  */
92 private class CommandAction implements ActionListener
93 {
94     public void actionPerformed(ActionEvent event)
95     {
96         String command = event.getActionCommand();
```

```
97
98     if (start)
99     {
100         if (command.equals("-"))
101         {
102             display.setText(command);
103             start = false;
104         }
105         else lastCommand = command;
106     }
107     else
108     {
109         calculate(Double.parseDouble(display.getText()));
110         lastCommand = command;
111         start = true;
112     }
113 }
114 }
115 /**
116  * Выполняет ожидающую калькуляцию
117  * @param x Значение, накапливаемое с учетом предыдущего результата
118  */
119 public void calculate(double x)
120 {
121     if (lastCommand.equals("+")) result += x;
122     else if (lastCommand.equals("-")) result -= x;
123     else if (lastCommand.equals("*")) result *= x;
124     else if (lastCommand.equals("/")) result /= x;
125     else if (lastCommand.equals("=")) result = x;
126     display.setText("" + result);
127 }
128 }
```

java.awt.GridLayout 1.0

- GridLayout(int rows, int columns)
- GridLayout(int rows, int columns, int hgap, int vgap)

Создают новый объект типа GridLayout с заданным расстоянием между компонентами по горизонтали и по вертикали.

Параметры:	<i>rows</i>	Количество рядов в сетке
	<i>columns</i>	Количество столбцов в сетке
	<i>hgap</i>	Расстояние между компонентами по горизонтали в пикселях (при отрицательных значениях компоненты перекрываются)
	<i>vgap</i>	Расстояние между компонентами по вертикали в пикселях (при отрицательных значениях компоненты перекрываются)

Ввод текста

Теперь можно приступить к рассмотрению компонентов пользовательского интерфейса, входящих в состав библиотеки Swing. Начнем с компонентов, дающих пользователю возможность вводить и править текст. Для этой цели предусмотрены два компонента: текстовое поле типа `JTextField` и текстовая область типа `JTextArea`. В текстовом поле можно ввести только одну текстовую строку, а в текстовой области — несколько строк. Поле типа `JPasswordField` принимает текстовую строку, не отображая ее содержимое.

Все эти три упомянутых выше класса для ввода текста расширяют класс `JTextComponent`. Создать объект этого класса нельзя, поскольку он является абстрактным. С другой стороны, как это часто бывает при программировании на Java, при просмотре документации на прикладной интерфейс API оказывается, что существуют методы, которые определены именно в классе `JTextComponent` и лишь наследуются его подклассами. В качестве примера ниже приведены методы, позволяющие установить текст или получить его из текстового поля или области.

`javax.swing.text.JTextComponent` 1.2

- `String getText()`
- `void setText(String text)`
Получают и устанавливают текст в данном текстовом компоненте.
- `boolean isEditable()`
- `void setEditable(boolean b)`
Получают и устанавливают свойство `editable`, определяющее, может ли пользователь редактировать содержимое данного текстового компонента.

Текстовые поля

Обычный способ ввести текстовое поле в окне — разместить его на панели или в другом контейнере аналогично кнопке. Ниже показано, как это делается в коде.

```
JPanel panel = new JPanel();  
JTextField textField = new JTextField("Default input", 20);  
panel.add(textField);
```

В приведенном выше фрагменте кода вводится текстовое поле, инициализируемое текстовой строкой "Default input" (Ввод по умолчанию). Второй параметр конструктора задает длину текстовой строки. В данном случае длина строки равна 20 символам. К сожалению, символы — не очень точная единица измерения. Их ширина зависит от выбранного шрифта. Дело в том, что если ожидается ввод n символов или меньше, то n следует указать в качестве ширины столбца. На практике такая единица измерения не совсем пригодна, и поэтому длину строки приходится завышать на один или два символа. Следует также учесть, что заданное количество символов считается в AWT лишь *предпочтительной* длиной строки. Решив уменьшить или увеличить текстовое поле, диспетчер компоновки изменит длину строки. Длина строки, задаваемая параметром конструктора типа `JTextField`, не является максимально

допустимым количеством символов, которое может ввести пользователь в данном текстовом поле. Пользователь может набирать и более длинные строки, а если набираемый текст выйдет за пределы текстового поля, то произойдет автоматическая прокрутка его содержимого. Но зачастую прокрутка текста плохо воспринимается пользователями, поэтому размеры текстового поля следует задавать с определенным запасом. Если же во время выполнения возникает необходимость изменить эти размеры, следует вызвать метод `setColumns()`.



СОВЕТ. После изменения размеров текстового поля методом `setColumns()` следует вызвать метод `revalidate()` из контейнера, содержащего данный компонент, как показано ниже.

```
textField.setColumns(10);  
panel.revalidate();
```

В методе `revalidate()` заново рассчитываются размеры и взаимное расположение всех компонентов в контейнере. Затем диспетчер компоновки перерисовывает контейнер, изменяя размеры текстового поля.

Метод `revalidate()` относится к классу `JComponent`. Его выполнение не приводит к немедленному изменению размеров компонента, который лишь помечается специальным образом. Такой подход исключает постоянные вычисления при запросе на изменение размеров нескольких компонентов. Если же требуется изменить размеры компонентов в контейнере типа `JFrame`, следует вызвать метод `validate()`, поскольку класс `JFrame` не является производным от класса `JComponent`.

Обычно пользователь программы должен иметь возможность вводить текст или редактировать содержимое текстового поля. Нередко в начале работы программы текстовые поля оказываются пустыми. Для того чтобы создать пустое текстовое поле, достаточно опустить соответствующий параметр в конструкторе класса `JTextField`, как показано ниже.

```
JTextField textField = new JTextField(20);
```

Содержимое текстового поля можно изменить в любой момент, вызвав метод `setText()` из родительского класса `TextComponent` следующим образом:

```
textField.setText("Hello!");
```

Как упоминалось ранее, определить, какой именно текст содержится в текстовом поле, можно с помощью метода `getText()`. Этот метод возвращает именно тот текст, который был набран пользователем. Чтобы отбросить лишние пробелы в начале и в конце текста, следует вызвать метод `trim()` по значению, возвращаемому методом `getText()`, как показано ниже. А для того чтобы задать шрифт, которым выделяется текст, следует вызвать метод `setFont()`.

```
String text = textField.getText().trim();
```

javax.swing.JTextField 1.2

- `JTextField(int cols)`

Создает пустое текстовое поле типа `JTextField` с заданным числом столбцов.

- `JTextField(String text, int cols)`

Создает текстовое поле указанных размеров с первоначальной символьной строкой и заданным числом столбцов.

- `int getColumns()`
- `void setColumns(int cols)`

Получают или устанавливают число столбцов для данного текстового поля.

`javax.swing.JComponent 1.2`

- `void revalidate()`

Обуславливает перерасчет местоположения и размеров компонента.

- `void setFont(Font f)`

Устанавливает шрифт для данного компонента.

`java.awt.Component 1.0`

- `void validate()`

Обуславливает перерасчет местоположения и размеров компонента. Если компонент является контейнером, местоположение и размеры содержащихся в нем компонентов должны быть также пересчитаны заново.

- `Font getFont()`

Получает шрифт данного компонента.

Метки и пометка компонентов

Метки являются компонентами, хранящими текст надписей. Они не имеют обрамления и других видимых элементов (например, границ). Кроме того, они не реагируют на ввод пользователем данных. Метки могут использоваться для обозначения компонентов. Например, в отличие от кнопок, текстовые компоненты не имеют меток, которые позволили бы их различать. Чтобы пометить компонент, не имеющий своего идентификатора, необходимо выполнить следующие действия.

1. Создать компонент типа `JLabel`, содержащий заданный текст.
2. Расположить его достаточно близко к компоненту, чтобы пользователь мог ясно видеть, что данная метка относится именно к этому компоненту.

Конструктор класса `JLabel` позволяет задать текст или пиктограмму, а если требуется, то и выровнять содержимое компонента. Для этой цели служат константы, объявленные в интерфейсе `SwingConstants`. В этом интерфейсе определено несколько полезных констант, в том числе `LEFT`, `RIGHT`, `CENTER`, `NORTH`, `EAST` и т.п. Класс `JLabel` является одним из нескольких классов из библиотеки `Swing`, реализующих

этот интерфейс. В качестве примера ниже показаны два варианта задания метки, текст надписи в которой будет, например, выровнен по левому краю.

```
JLabel label = new JLabel("User name: ", SwingConstants.RIGHT);
```

Или

```
JLabel label = new JLabel("User name: ", JLabel.RIGHT);
```

А с помощью методов `setText()` и `setIcon()` можно задать текст надписи и пиктограмму для метки во время выполнения.



СОВЕТ. В качестве надписей на кнопках, метках и пунктах меню можно использовать как обычный текст, так и текст, размеченный в формате HTML. Тем не менее указывать текст надписей на кнопках в формате HTML не рекомендуется, поскольку он нарушает общий стиль пользовательского интерфейса. Впрочем, для меток такой текст может оказаться довольно эффективным. Для этого текстовую строку надписи на метке достаточно разместить между дескрипторами `<html> . . .</html>` следующим образом:

```
label = new JLabel("<html><b>Required</b> entry:</html>");
```

Следует, однако, иметь в виду, что первый компонент с меткой, набранной текстом в формате HTML, отображается на экране с запаздыванием, поскольку для этого нужно загрузить довольно сложный код интерпретации и воспроизведения содержимого, размеченного в формате HTML.

Метки можно размещать в контейнере подобно любому другому компоненту пользовательского интерфейса. Это означает, что для их размещения применяются те же самые подходы, что и рассмотренные ранее.

javax.swing.JLabel 1.2

- `JLabel(String text)`
- `JLabel(Icon icon)`
- `JLabel(String text, int align)`
- `JLabel(String text, Icon icon, int align)`

Создают метку с текстом и пиктограммой. Пиктограмма располагается левее текста.

Параметры:

<i>text</i>	Текст метки
<i>icon</i>	Пиктограмма метки
<i>align</i>	Одна из констант, определенных в интерфейсе <code>SwingConstants</code> : <code>LEFT</code> (по умолчанию), <code>CENTER</code> или <code>RIGHT</code>

- `String getText()`
- `void setText(String text)`
Получают или устанавливают текст данной метки.
- `Icon getIcon()`
- `void setIcon(Icon icon)`
Получают или устанавливают пиктограмму данной метки.

Поля для ввода пароля

Поля для ввода пароля представляют собой особый вид текстовых полей. Чтобы скрыть пароль от посторонних наблюдателей, его символы не отображаются на экране. Вместо этого каждый символ в пароле заменяется *эхо-символом*, обычно звездочкой (*). В библиотеке Swing предусмотрен класс `JPasswordField`, реализующий такое текстовое поле.

Поле для ввода пароля служит еще одним примером, наглядно демонстрирующим преимущества шаблона “модель–представление–контроллер”. В целях хранения данных в поле для ввода пароля применяется та же самая модель, что и для обычного текстового поля, но представление этого поля изменено, заменяя все символы пароля эхо-символами.

`javax.swing.JPasswordField 1.2`

- `JPasswordField(String text, int columns)`
Создает новое поле для ввода пароля.
- `void setEchoChar(char echo)`
Задает эхо-символ, который может зависеть от визуального стиля интерфейса. Если задано нулевое значение, выбирается эхо-символ по умолчанию.
- `char[] getPassword()`
Возвращает текст, содержащийся в поле для ввода пароля. Для обеспечения большей безопасности возвращаемый массив после использования следует перезаписать. Пароль возвращается как массив символов, а не как объект типа `String`. Причина такого решения заключается в том, что символьная строка может оставаться в виртуальной машине до тех пор, пока она не будет уничтожена механизмом “сборки мусора”.

Текстовые области

Иногда возникает потребность ввести несколько строк. Как указывалось ранее, для этого применяется компонент типа `JTextArea`. Поместив данный компонент в свою программу, разработчик предоставляет пользователю возможность вводить сколько угодно текста, разделяя его строки нажатием клавиши `<Enter>`. Каждая строка заканчивается символом `'\n'`, как это предусмотрено в Java. Пример текстовой области в действии показан на рис. 9.13.

В конструкторе компонента типа `JTextArea` указывается количество строк и их длина, как в следующем примере кода:

```
textArea = new JTextArea(8, 40); // 8 строк по 40 столбцов в каждой
```

Параметр `columns`, задающий количество столбцов (а по существу, символов) в строке, действует так же, как и для текстового поля; его значение рекомендуется не много зависить. Пользователь не ограничен количеством вводимых строк и их длиной. Если длина строки или число строк выйдет за пределы заданных параметров, текст будет прокручиваться в окне. Для изменения длины строк можно вызвать метод `setColumns()`, а для изменения их количества — метод `setRows()`. Эти параметры задают лишь рекомендуемые размеры, а диспетчер компоновки может самостоятельно увеличивать или уменьшать размеры текстовой области.



Рис. 9.13. Текстовая область вместе с другими текстовыми компонентами

Если пользователь введет больше текста, чем умещается в текстовой области, то остальной текст просто отсекается. Этого можно избежать, установив автоматический перенос строки следующим образом:

```
textArea.setLineWrap(true); // в длинных строках выполняется перенос
```

Автоматический перенос строки проявляется лишь визуально. Текст, хранящийся в документе, не изменяется — в него не вставляются символы '\n'.

Панели прокрутки

В библиотеке Swing текстовая область не снабжается полосами прокрутки. Если они требуются, текстовую область следует ввести на *панели прокрутки*, как показано ниже.

```
textArea = new JTextArea(8, 40);  
JScrollPane scrollPane = new JScrollPane(textArea);
```

Теперь панель прокрутки управляет представлением текстовой области. Полосы прокрутки появляются автоматически, когда текст выходит за пределы отведенной для него области, и исчезают, когда оставшаяся часть текста удаляется. Сама прокрутка обеспечивается панелью прокрутки, а программа не обязана обрабатывать события, связанные с прокруткой.

Это универсальный механизм, который пригоден для любого компонента, а не только для текстовых областей. Чтобы ввести полосы прокрутки в компонент, достаточно разместить его на панели прокрутки.

В программе из листинга 9.2 демонстрируются различные текстовые компоненты. Эта программа отображает текстовое поле, поле для ввода пароля и текстовую область с полосами прокрутки. Текстовое поле и поле для ввода пароля снабжены метками. Чтобы ввести предложение в конце текста, следует щелкнуть на кнопке Insert (Вставить).



НА ЗАМЕТКУ! Компонент типа `JTextArea` позволяет отображать только простой текст без форматирования и выделения специальными шрифтами. Для отображения отформатированного текста (например, в виде HTML-разметки) можно воспользоваться классом `JEditorPane`, подробнее рассматриваемым во втором томе данной книги.

Листинг 9.2. Исходный код из файла `text/TextComponentFrame.java`

```
1 package text;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Фрейм с образцами текстовых компонентов
9  */
10 public class TextComponentFrame extends JFrame
11 {
12     public static final int TEXTAREA_ROWS = 8;
13     public static final int TEXTAREA_COLUMNS = 20;
14
15     public TextComponentFrame()
16     {
17         final JTextField textField = new JTextField();
18         final JPasswordField passwordField = new JPasswordField();
19
20         JPanel northPanel = new JPanel();
21         northPanel.setLayout(new GridLayout(2, 2));
22         northPanel.add(new JLabel("User name: ", SwingConstants.RIGHT));
23         northPanel.add(textField);
24         northPanel.add(new JLabel("Password: ", SwingConstants.RIGHT));
25         northPanel.add(passwordField);
26
27         add(northPanel, BorderLayout.NORTH);
28
29         final JTextArea textArea =
30             new JTextArea(TEXTAREA_ROWS, TEXTAREA_COLUMNS);
31         JScrollPane scrollPane = new JScrollPane(textArea);
32
33         add(scrollPane, BorderLayout.CENTER);
34
35         // ввести кнопку для заполнения текстовой области текстом
36
37         JPanel southPanel = new JPanel();
38
39         JButton insertButton = new JButton("Insert");
40         southPanel.add(insertButton);
41         insertButton.addActionListener(new ActionListener()
42         {
43             public void actionPerformed(ActionEvent event)
44             {
45                 textArea.append("User name: " + textField.getText() +
46                     " Password: " + new String(passwordField.getPassword()) + "\n");
47             }
48         });
49     }
50 }
```

```
50     add(southPanel, BorderLayout.SOUTH);
51     pack();
52 }
53 }
```

javax.swing.JTextArea 1.2

- `JTextArea()`
- `JTextArea(int rows, int cols)`
- `JTextArea(String text, int rows, int cols)`
Создают новую текстовую область.
- `void setColumns(int cols)`
Задаёт предпочтительное число столбцов, определяющее длину строк в текстовой области.
- `void setRows(int rows)`
Задаёт предпочтительное число строк в текстовой области.
- `void append(String newText)`
Добавляет заданный текст в конце содержимого текстовой области.
- `void setLineWrap(boolean wrap)`
Включает и отключает режим автоматического переноса строк.
- `void setWrapStyleWord(boolean word)`
Если параметр *word* имеет логическое значение `true`, перенос в длинных строках выполняется по границам слов, в противном случае границы слов во внимание не принимаются.
- `void setTabSize(int c)`
Устанавливает позиции табуляции через каждые *c* символов. Следует иметь в виду, что символы табуляции не преобразуются в пробелы и лишь выравнивают текст по следующей позиции табуляции.

javax.swing.JScrollPane 1.2

- `JScrollPane(Component c)`
Создает панель прокрутки, которая отображает содержимое указанного компонента. Полоса прокрутки появляется лишь в том случае, если компонент крупнее представления.

Компоненты для выбора разных вариантов

Итак мы рассмотрели, как принимать текстовые данные, вводимые пользователем. Но во многих случаях предпочтительнее ограничить действия пользователя выбором среди конечного набора вариантов. Эти варианты могут быть представлены кнопками или списком выбираемых элементов. (Как правило, такой подход освобождает от необходимости отслеживать ошибки ввода.) В этом разделе описывается порядок программирования таких компонентов пользовательского интерфейса, как флажки, кнопки-переключатели, списки и регулируемые ползунки.

Флажки

Если данные сводятся к двухзначной логике вроде положительного или отрицательного ответа, то для их ввода можно воспользоваться таким компонентом, как флажок. Чтобы установить флажок, достаточно щелкнуть кнопкой мыши на этом компоненте, а для того чтобы сбросить флажок — щелкнуть на нем еще раз. Установить или сбросить флажок можно также с помощью клавиши пробела, нажав ее в тот момент, когда на данном компоненте находится фокус ввода.

На рис. 9.14 показано простое окно прикладной программы с двумя флажками, один из которых включает и отключает курсивное, а другой — полужирное начертание шрифта. Обратите внимание на то, что первый флажок обладает фокусом ввода. Об этом свидетельствует прямоугольная рамка вокруг его метки. Всякий раз, когда пользователь щелкает на флажке, содержимое окна обновляется с учетом нового начертания шрифта.

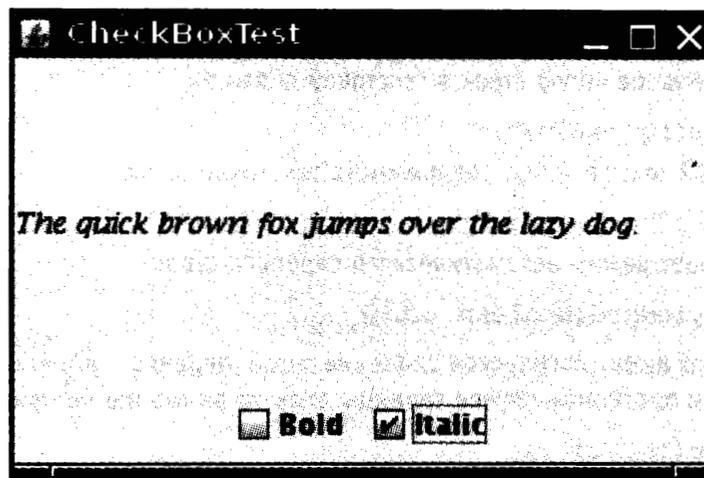


Рис. 9.14. Флажки

Флажки сопровождаются метками, указывающими их назначение. Текст метки задается в конструкторе следующим образом:

```
bold = new JCheckBox("Bold");
```

Для установки и сброса флажка вызывается метод `setSelected()`, как показано ниже.

```
bold.setSelected(true);
```

Метод `isSelected()` позволяет определить текущее состояние каждого флажка. Если он возвращает логическое значение `false`, значит, флажок сброшен, а если логическое значение `true` — флажок установлен.

Щелкая на флажке, пользователь инициирует определенные события. Как всегда, с данным компонентом можно связать объект приемника событий. В рассматриваемом здесь примере программы для обоих флажков предусмотрен один и тот же приемник действий:

```
ActionListener listener = . . .  
bold.addActionListener(listener);  
italic.addActionListener(listener);
```

В приведенном ниже методе `actionPerformed()` обработки событий запрашивается текущее состояние флажков `bold` и `italic`, а затем устанавливается начертание

шрифта, которым должен отображаться обычный текст: **полужирный**, *курсив* или **полужирный курсив**.

```
public void actionPerformed(ActionEvent event)
{
    int mode = 0;
    if (bold.isSelected()) mode += Font.BOLD;
    if (italic.isSelected()) mode += Font.ITALIC;
    label.setFont(new Font("Serif", mode, FONTSIZE));
}
```

В листинге 9.3 приведен весь исходный код программы, демонстрирующей обращение с флажками при построении ГПИ.

Листинг 9.3. Исходный код из файла `checkBox/CheckBoxTest.java`

```
package checkBox;
2
3 import java.awt.*;
4 import javax.swing.*;
5 /**
6  * @version 1.33 2007-06-12
7  * @author Cay Horstmann
8  */
9 public class CheckBoxTest
10 {
11     public static void main(String[] args)
12     {
13         EventQueue.invokeLater(new Runnable()
14         {
15             public void run()
16             {
17                 JFrame frame = new CheckBoxFrame();
18                 frame.setTitle("CheckBoxTest");
19                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20                 frame.setVisible(true);
21             }
22         });
23     }
24 }
```

javax.swing.JCheckBox 1.2

- `JCheckBox(String label)`
- `JCheckBox(String label, Icon icon)`
Создают флажок, который исходно сброшен.
- `JCheckBox(String label, boolean state)`
Создает флажок с указанной меткой и заданным исходным состоянием.
- `boolean isSelected ()`
- `void setSelected(boolean state)`
Получают или устанавливают новое состояние флажка.

Кнопки-переключатели

В предыдущем примере пользователь мог установить оба флажка, один из них или ни одного. Но зачастую требуется выбрать только один из предлагаемых вариантов. Если пользователь установит другой флажок, то предыдущий флажок будет сброшен. Такую группу флажков часто называют *группой кнопок-переключателей*, поскольку они напоминают переключатели диапазонов на радиоприемниках — при нажатии одной из таких кнопок ранее нажатая кнопка возвращается в исходное состояние. На рис. 9.15 приведен типичный пример окна прикладной программы с группой кнопок-переключателей. Пользователь может выбрать размер шрифта — Small (Малый), Medium (Средний), Large (Крупный) и Extra large (Очень крупный). Разумеется, выбрать можно лишь один размер шрифта.



Рис. 9.15. Группа кнопок-переключателей

Библиотека Swing позволяет легко реализовать группы кнопок-переключателей. Для этого нужно создать по одному объекту типа `ButtonGroup` на каждую группу. Затем в группу кнопок-переключателей необходимо ввести объекты типа `JRadioButton`. Объект типа `ButtonGroup` служит для того, чтобы отключать выбранную ранее кнопку-переключатель, если пользователь щелкнет на новой кнопке. Ниже показано, каким образом все это воплощается в коде.

```
ButtonGroup group = new ButtonGroup();

JRadioButton smallButton = new JRadioButton("Small", false);
group.add(smallButton);

JRadioButton mediumButton = new JRadioButton("Medium", true);
group.add(mediumButton);
. . .
```

Второй параметр конструктора принимает логическое значение `true`, если изначально кнопка-переключатель должна быть включена, и логическое значение `false`, если она должна быть выключена. Следует иметь в виду, что объект типа `ButtonGroup` управляет лишь поведением кнопок-переключателей. Если нужно объединить несколько групп кнопок-переключателей, их следует разместить в контейнере, например в объекте типа `JPanel`.

Сравнив рис. 9.14 и 9.15, обратите внимание на то, что кнопки-переключатели отличаются по внешнему виду от флажков. Флажки изображаются в виде квадратов, причем внутри установленных флажков указывается галочка, в то время как

кнопки-переключатели имеют круглую форму: включенные — с точкой внутри, а выключенные — без обозначений.

Механизм уведомления о наступлении событий от кнопок-переключателей точно такой же, как и для любых других видов кнопок. Если пользователь выберет кнопку-переключатель, соответствующий объект инициирует событие. В рассматриваемом здесь примере программы установлен приемник событий, задающий конкретный размер шрифта, как показано ниже.

```
ActionListener listener = new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            // размер шрифта указывается в качестве последнего
            // параметра метода addRadioButton()
            label.setFont(new Font("Serif", Font.PLAIN, size));
        }
    };
```

Сравните этот приемник событий с приемником событий от флажка. Каждой кнопке-переключателю соответствует свой объект приемника событий. И каждому приемнику событий точно известно, что нужно делать — установить конкретный размер шрифта. Совсем иначе дело обстоит с флажками. Оба флажка в рассмотренном ранее примере были связаны с одним и тем же приемником событий. Он вызывал метод, определяющий текущее состояние обоих флажков.

Можно ли применить такой же подход для кнопок-переключателей? С этой целью можно было бы задать один приемник событий, устанавливающий конкретный размер шрифта, как показано ниже. Но все же предпочтительнее использовать отдельные объекты приемники событий, поскольку они более тесно связывают размер шрифта с конкретной кнопкой-переключателем.

```
if (smallButton.isSelected()) size = 8;
else if (mediumButton.isSelected()) size = 12;
. . .
```



НА ЗАМЕТКУ! В группе может быть выбрана только одна кнопка-переключатель. Хорошо бы заранее знать, какая именно, не проверяя каждую кнопку-переключатель в группе. Объект типа `ButtonGroup` управляет всеми кнопками-переключателями, и поэтому было бы удобно, если бы он предоставлял ссылку на выбранную кнопку-переключатель. В самом деле, в классе `ButtonGroup` имеется метод `getSelection()`, но он не возвращает ссылку на выбранную кнопку-переключатель. Вместо этого он возвращает ссылку типа `ButtonModel` на модель, связанную с этой кнопкой-переключателем. К сожалению, все методы из интерфейса `ButtonModel` не представляют собой ничего ценного в этом отношении.

Интерфейс `ButtonModel` наследует от интерфейса `ItemSelectable` метод `getSelectedObjects()`, возвращающий совершенно бесполезную пустую ссылку типа `null`. Метод `getActionCommand()` выглядит предпочтительнее, поскольку позволяет определить текстовую строку с командой действия, а по существу, с текстовой меткой кнопки-переключателя. Но команда в модели этой кнопки-переключателя снова представляет собой пустую ссылку типа `null`. И только в том случае, если явно задать команды действий для каждой кнопки-переключателя с помощью метода `setActionCommand()`, в модели установятся значения, соответствующие каждой команде действия. А в дальнейшем команду действия для включенной кнопки-переключателя можно будет определить, сделав вызов `buttonGroup.getSelection().getActionCommand()`.

В листинге 9.4 представлен весь исходный код программы, в которой размер шрифта устанавливается с помощью кнопок-переключателей.

Листинг 9.4. Исходный код из файла `radioButton/RadioButtonFrame.java`

```
1 package radioButton;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Фрейм с образцами текстовых меток и кнопок-переключателей
9  * для выбора размеров шрифта
10 */
11 public class RadioButtonFrame extends JFrame
12 {
13     private JPanel buttonPanel;
14     private ButtonGroup group;
15     private JLabel label;
16     private static final int DEFAULT_SIZE = 36;
17
18     public RadioButtonFrame()
19     {
20         // ввести образец текстовой метки
21         label = new JLabel("The quick brown fox jumps over the lazy dog.");
22         label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
23         add(label, BorderLayout.CENTER);
24         // ввести кнопки-переключатели
25         buttonPanel = new JPanel();
26         group = new ButtonGroup();
27
28         addRadioButton("Small", 8);
29         addRadioButton("Medium", 12);
30         addRadioButton("Large", 18);
31         addRadioButton("Extra large", 36);
32
33         add(buttonPanel, BorderLayout.SOUTH);
34         pack();
35     }
36
37     /**
38      * Ввести кнопку-переключатель, устанавливающую размер шрифта
39      * для выделения образца текста
40      * @param name Строка надписи на кнопке
41      * @param size Размер шрифта, устанавливаемый данной кнопкой
42      */
43     public void addRadioButton(String name, final int size)
44     {
45         boolean selected = size == DEFAULT_SIZE;
46         JRadioButton button = new JRadioButton(name, selected);
47         group.add(button);
48         buttonPanel.add(button);
49
50         // этот приемник событий устанавливает размер шрифта для метки
51         ActionListener listener = new ActionListener()
52         {
```

```
53.     public void actionPerformed(ActionEvent event)
54.     {
55.         // размер шрифта указывается в качестве последнего
56.         // параметра метода addRadioButton()
57.         label.setFont(new Font("Serif", Font.PLAIN, size));
58.     }
59. };
58
59     button.addActionListener(listener);
60 }
61 }
```

javax.swing.JRadioButton 1.2

- `JRadioButton(String label, Icon icon)`
Создает кнопку-переключатель, которая исходно не выбрана.
- `JRadioButton(String label, boolean state)`
Создает кнопку-переключатель с заданной меткой и в указанном исходном состоянии.

javax.swing.ButtonGroup 1.2

- `void add(AbstractButton b)`
Вводит кнопку-переключатель в группу.
- `ButtonModel getSelection()`
Возвращает модель выбранной кнопки.

javax.swing.ButtonModel 1.2

- `String getActionCommand()`
Возвращает команду для модели данной кнопки.

javax.swing.AbstractButton 1.2

- `void setActionCommand(String s)`
Задает команду для данной кнопки и ее модели.

Рамки

Если в одном окне расположено несколько групп кнопок-переключателей, их нужно каким-то образом различать. Для этого в библиотеке Swing предусмотрен набор *рамки*. Рамку можно задать для каждого компонента, расширяющего класс `JComponent`. Обычно в рамку заключается панель, заполняя ее элементами

пользовательского интерфейса, например, кнопками-переключателями. Выбор рамок невелик, и все они задаются с помощью одинаковых действий, описываемых ниже.

1. Вызовите статический метод из класса `BorderFactory`, создающий рамку в одном из следующих стилей (рис. 9.16):

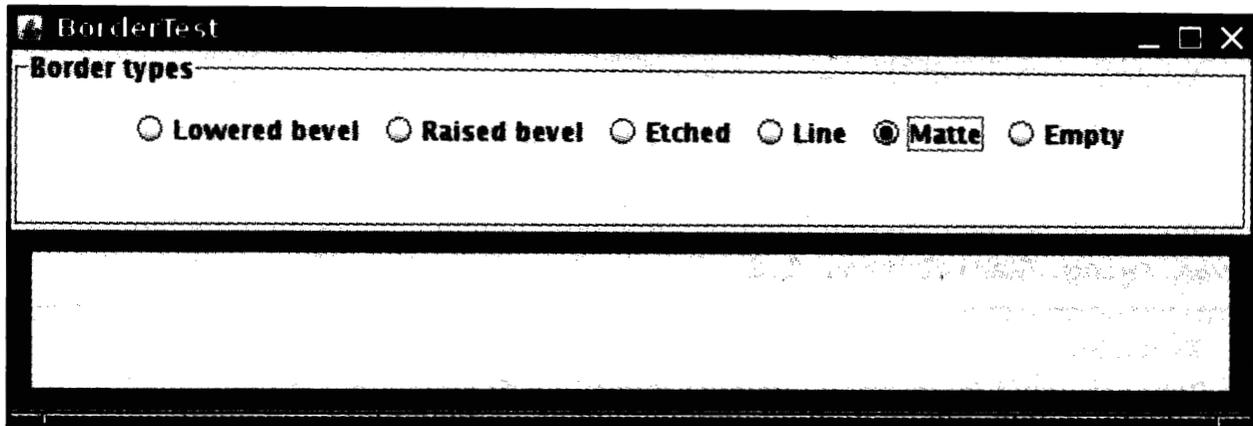


Рис. 9.16. Опробование различных видов рамок

- Lowered bevel (Утопленная фаска)
 - Raised bevel (Приподнятая фаска)
 - Etched (Гравировка)
 - Line (Линия)
 - Matte (Кайма)
 - Empty (Пустая — создается пустое пространство, окружающее компонент)
2. По желанию снабдите рамку с заголовком, сделав вызов `BorderFactory.createTitledBorder()`.
 3. Если требуется, объедините несколько рамок в одну, сделав вызов `BorderFactory.createCompoundBorder()`.
 4. Добавьте полученную в итоге рамку с помощью метода `setBorder()` из класса `JComponent`.

В приведенном ниже фрагменте кода на панели вводится рамка в стиле гравировки с указанным заголовком.

```
Border etched = BorderFactory.createEtchedBorder();
Border titled = BorderFactory.createTitledBorder(etched, "A Title");
panel.setBorder(titled);
```

Запустите на выполнение программу, исходный код которой приведен в листинге 9.5, чтобы посмотреть, как выглядят рамки, оформленные в разных стилях. У различных рамок имеются разные возможности для задания ширины и цвета. Подробнее об этом — в документации на прикладной интерфейс API. Истинные любители пользоваться рамками оценят по достоинству возможность сглаживать и скруглять углы рамок, предоставляемую в классах `SoftBevelBorder` и `LineBorder`. Такие рамки можно создать только с помощью конструкторов этих классов — для них не предусмотрены соответствующие методы в классе `BorderFactory`.

Листинг 9.5. Исходный код из файла border/BorderFrame.java

```
1 package border;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import javax.swing.border.*;
7
8 /**
9  * Фрейм с кнопками-переключателями для выбора стиля рамки
10 */
11 public class BorderFrame extends JFrame
12 {
13     private JPanel demoPanel;
14     private JPanel buttonPanel;
15     private ButtonGroup group;
16
17     public BorderFrame()
18     {
19         demoPanel = new JPanel();
20         buttonPanel = new JPanel();
21         group = new ButtonGroup();
22
23         addRadioButton("Lowered bevel",
24             BorderFactory.createLoweredBevelBorder());
25         addRadioButton("Raised bevel",
26             BorderFactory.createRaisedBevelBorder());
27         addRadioButton("Etched", BorderFactory.createEtchedBorder());
28         addRadioButton("Line", BorderFactory.createLineBorder(Color.BLUE));
29         addRadioButton("Matte", BorderFactory.createMatteBorder(
30             10, 10, 10, 10, Color.BLUE));
31         addRadioButton("Empty", BorderFactory.createEmptyBorder());
32
33         Border etched = BorderFactory.createEtchedBorder();
34         Border titled =
35             BorderFactory.createTitledBorder(etched, "Border types");
36         buttonPanel.setBorder(titled);
37
38         setLayout(new GridLayout(2, 1));
39         add(buttonPanel);
40         add(demoPanel);
41         pack();
42     }
43
44     public void addRadioButton(String buttonName, final Border b)
45     {
46         JRadioButton button = new JRadioButton(buttonName);
47         button.addActionListener(new ActionListener()
48             {
49             public void actionPerformed(ActionEvent event)
50             {
51                 demoPanel.setBorder(b);
52             }
53             });
54         group.add(button);
55         buttonPanel.add(button);
56     }
57 }
```

javax.swing.BorderFactory 1.2

- `static Border createLineBorder(Color color)`
- `static Border createLineBorder(Color color, int thickness)`
Создают рамку в стиле простой линии.
- `static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Color color)`
- `static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Icon tileIcon)`
Создают широкую рамку, заполняемую цветом или рисунком из повторяющихся пиктограмм.
- `static Border createEmptyBorder()`
- `static Border createEmptyBorder(int top, int left, int bottom, int right)`
Создают пустую рамку.
- `static Border createEtchedBorder()`
- `static Border createEtchedBorder(Color highlight, Color shadow)`
- `static Border createEtchedBorder(int type)`
- `static Border createEtchedBorder(int type, Color highlight, Color shadow)`
Создают простую рамку в стиле линии с трехмерным эффектом.
Параметры: *highlight, shadow* Цвета для трехмерного эффекта
type Стиль рамки, определяемый одной из констант `EtchedBorder.RAISED`, `EtchedBorder.LOWERED`
- `static Border createBevelBorder(int type)`
- `static Border createBevelBorder(int type, Color highlight, Color shadow)`
- `static Border createLoweredBevelBorder()`
- `static Border createRaisedBevelBorder()`
Создают рамку с эффектом утопленной или приподнятой поверхности.
Параметры: *highlight, shadow* Цвета для трехмерного эффекта
type Стиль рамки, определяемый одной из констант `EtchedBorder.RAISED`, `EtchedBorder.LOWERED`
- `static TitledBorder createTitledBorder(String title)`
- `static TitledBorder createTitledBorder(Border border)`
- `static TitledBorder createTitledBorder(Border border, String title)`
- `static TitledBorder createTitledBorder(Border border, String title, int justification, int position)`
- `static TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font)`

- `static TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font, Color color)`

Создают рамку с заданными свойствами и снабженную заголовком.

Параметры:	<code>title</code>	Символьная строка заголовка
	<code>border</code>	Рамка для оформления заголовка
	<code>justification</code>	Выравнивание заголовка рамки, определяемое одной из констант: LEFT, CENTER, RIGHT, LEADING, TRAILING или DEFAULT_JUSTIFICATION (по левому краю) из класса TitledBorder
	<code>font</code>	Шрифт заголовка
	<code>color</code>	Цвет заголовка

- `static CompoundBorder createCompoundBorder(Border outsideBorder, Border insideBorder)`

Объединяет две рамки в одну новую.

javax.swing.border.SoftBevelBorder 1.2

- `SoftBevelBorder(int type)`
- `SoftBevelBorder(int type, Color highlight, Color shadow)`

Создают скошенную рамку со сглаженными углами.

Параметры:	<code>highlight, shadow</code>	Цвета для трехмерного эффекта
	<code>type</code>	Стиль рамки, определяемый одной из констант: EtchedBorder.RAISED, EtchedBorder.LOWERED

javax.swing.border.LineBorder 1.2

- `public LineBorder(Color color, int thickness, boolean roundedCorners)`

Создает рамку в стиле линии заданной толщины и цвета. Если параметр `roundedCorners` принимает логическое значение `true`, рамка имеет скругленные углы.

javax.swing.JComponent 1.2

- `void setBorder(Border border)`
- Задаёт рамку для данного компонента.

Комбинированные списки

Если вариантов для выбора слишком много, то кнопки-переключатели для этой цели не подойдут, поскольку для них не хватит места на экране. В таком случае следует использовать *раскрывающийся список*. Если пользователь щелкнет на этом компоненте, раскроется список, откуда он может выбрать один из элементов (рис. 9.17).

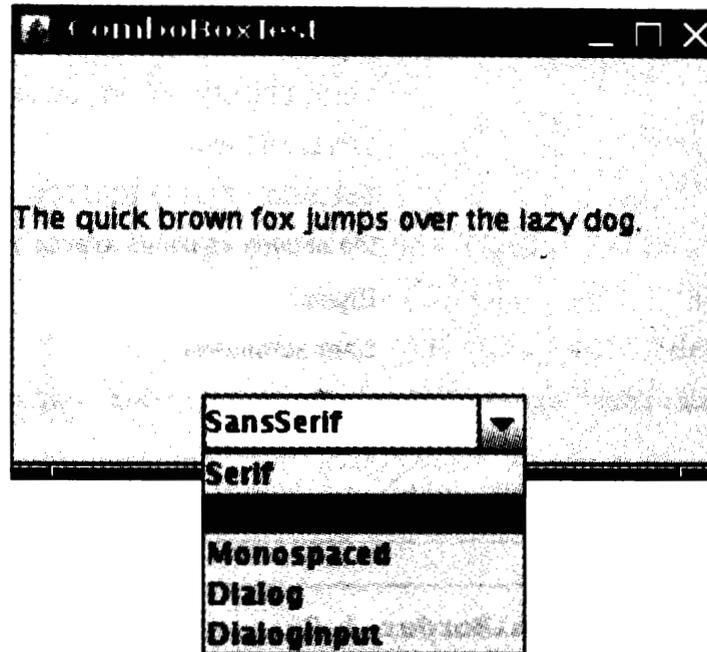


Рис. 9.17. Раскрывающийся список

Если раскрывающийся список является *редактируемым*, то выбранный из него элемент можно поправить так же, как в обычном тестовом поле. Таким образом, редактируемый раскрывающийся список объединяет в себе удобства текстового поля и возможность выбора из predetermined ряда вариантов, и в этом случае такой список называется *комбинированным*. Компоненты комбинированных списков создаются средствами класса `JComboBox`. Начиная с версии Java SE 7, класс `JComboBox` является обобщенным. Например, комбинированный список типа `JComboBox<String>` состоит из строковых объектов типа `String`, а комбинированный список типа `JComboBox<Integer>` — из целочисленных значений.

Для того чтобы сделать раскрывающийся список редактируемым, т.е. комбинированным, достаточно вызвать метод `setEditable()`. Следует, однако, иметь в виду, что изменения вносятся только в текущий элемент списка. Перечень вариантов выбора все равно остается прежним.

Выбранный вариант в исходном или отредактированном виде можно получить с помощью метода `getSelectedItem()`. Но для комбинированного списка этот элемент может быть любого типа в зависимости от редактора, принимающего пользовательские правки и сохраняющего результат в соответствующем объекте. (Подробнее о редакторах речь пойдет в главе 6 второго тома данной книги.) Если же список является раскрывающимся и не допускает редактирование своих элементов, то для получения выбранного варианта нужного типа лучше сделать следующий вызов:

```
combo.getItemAt (combo.getSelectedIndex ())
```

В рассматриваемом здесь примере программы у пользователя имеется возможность выбрать стиль шрифта из предварительно заданного списка

(Serif — с засечками, SunsSerif — без засечек, Monospaced — моноширинный и т.д.). Кроме того, пользователь может ввести в список новый стиль шрифта, добавив в список соответствующий элемент с помощью метода `addItem()`. В данной программе метод `addItem()` вызывается только в конструкторе, как показано ниже, но в случае необходимости к нему можно обратиться из любой части программы.

```
JComboBox<String> faceCombo = new JComboBox<>();
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
. . .
```

Этот метод добавляет символьную строку в конец списка. Если же требуется вставить символьную строку в любом другом месте списка, нужно вызвать метод `insertItemAt()` следующим образом:

```
faceCombo.insertItemAt("Monospaced", 0); // ввести элемент в начале списка
```

В список можно вводить элементы любого типа, а для отображения вызывается метод `toString()`. Если во время выполнения возникает потребность удалить элемент из списка, для этой цели вызывается метод `removeItem()` или `removeItemAt()`, в зависимости от того, что указать: сам удаляемый элемент или его местоположение в списке, как показано ниже. А для удаления сразу всех элементов из списка предусмотрен метод `removeAllItems()`.

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // удалить первый элемент из списка
```



СОВЕТ. Если в комбинированный список требуется включить большое количество объектов, применять для этой цели метод `addItem()` не следует, чтобы не снижать производительность программы. Вместо этого лучше сконструировать объект типа `DefaultComboBoxModel`, заполнить его элементами составляемого списка, вызывая метод `addElement()`, а затем обратиться к методу `setModel()` из класса `JComboBox`.

Когда пользователь выбирает нужный вариант из комбинированного списка, этот компонент инициирует событие. Для того чтобы определить выбранный из списка вариант, следует вызвать метод `getSource()` с данным событием в качестве параметра. Этот метод возвращает ссылку на список, являющийся источником события. Затем следует вызвать метод `getSelectedItem()`, возвращающий выбранный из списка вариант. Значение, возвращаемое этим методом, необходимо привести к соответствующему типу (как правило, к типу `String`). Но если возвращаемое значение передается в качестве параметра методу `getItemAt()`, то приведение типов не требуется, как выделено ниже полужирным.

```
public void actionPerformed(ActionEvent event)
{
    label.setFont(new Font(
        faceCombo.getItemAt(faceCombo.getSelectedIndex()),
        Font.PLAIN,
        DEFAULT_SIZE));
}
```

Весь исходный код программы, демонстрирующей применение комбинированного списка в пользовательском интерфейсе, приведен в листинге 9.6.



НА ЗАМЕТКУ! Если требуется показать обычный список вместо раскрывающегося, т.е. чтобы элементы списка постоянно отображались на экране, воспользуйтесь компонентом типа `JList`, который будет подробнее рассмотрен во втором томе данной книги.

Листинг 9.6. Исходный код из файла `comboBox/ComboBoxFrame.java`

```
1 package comboBox;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Фрейм с образцами текстовой метки и комбинированного списка
9  * для выбора начертаний шрифта
10 */
11 public class ComboBoxFrame extends JFrame
12 {
13     private JComboBox<String> faceCombo;
14     private JLabel label;
15     private static final int DEFAULT_SIZE = 24;
16
17     public ComboBoxFrame()
18     {
19         // добавить образец текстовой метки
20         label = new JLabel("The quick brown fox jumps over the lazy dog.");
21         label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
22         add(label, BorderLayout.CENTER);
23
24         // составить комбинированный список и ввести
25         // в него названия начертаний шрифта
26         faceCombo = new JComboBox<>();
27         faceCombo.addItem("Serif");
28         faceCombo.addItem("SansSerif");
29         faceCombo.addItem("Monospaced");
30         faceCombo.addItem("Dialog");
31         faceCombo.addItem("DialogInput");
32
33         // приемник событий от комбинированного списка изменяет на
34         // выбранное начертание шрифта, которым набрана тестовая метка
35         faceCombo.addActionListener(new ActionListener()
36         {
37             public void actionPerformed(ActionEvent event)
38             {
39                 label.setFont(new Font(
40                     faceCombo.getItemAt(faceCombo.getSelectedIndex()),
41                     Font.PLAIN, DEFAULT_SIZE));
42             }
43         });
44
45         // ввести комбинированный список на панели в нижней части фрейма
46         JPanel comboPanel = new JPanel();
47         comboPanel.add(faceCombo);
48         add(comboPanel, BorderLayout.SOUTH);
49         pack();
50     }
51 }
```

javax.swing.JComboBox 1.2

- `boolean isEditable()`
- `void setEditable(boolean b)`
Получают или устанавливают свойство `editable` данного комбинированного списка.
- `void addItem(Object item)`
Вводит новый элемент в список.
- `void insertItemAt(Object item, int index)`
Вводит элемент в список по указанному индексу.
- `void removeItem(Object item)`
Удаляет заданный элемент из списка.
- `void removeItemAt(int index)`
Удаляет из списка элемент по указанному индексу.
- `void removeAllItems()`
Удаляет из списка все элементы.
- `Object getSelectedItem()`
Возвращает выбранный элемент списка.

Регулируемые ползунки

Комбинированные списки дают пользователю возможность делать выбор из дискретного ряда вариантов. А регулируемые ползунки позволяют выбрать конкретное значение в заданных пределах, например, любое число в пределах от 1 до 100. Чаще всего регулируемые ползунки создаются следующим образом:

```
JSlider slider = new JSlider(min, max, initialValue);
```

Если опустить минимальное, максимальное и начальное значения, то по умолчанию выбираются значения 0, 100 и 50 соответственно. А если регулируемый ползунок должен располагаться вертикально, то для этой цели служит конструктор

```
JSlider slider = new JSlider(SwingConstants.VERTICAL, min, max, initialValue);
```

Каждый такой конструктор создает простой ползунок. В качестве примера можно привести ползунок в верхней части окна, представленного на рис. 9.18. Далее будут рассмотрены более сложные разновидности регулируемых ползунков.

Когда пользователь перемещает ползунок, выбираемое значение в данном компоненте изменяется в пределах от минимального до максимального. При этом все приемники событий от регулируемого ползунка получают событие типа `ChangeEvent`. Для того чтобы получать уведомления об изменении выбираемого значения при перемещении ползунка, следует сначала создать объект класса, реализующего интерфейс `ChangeListener`, а затем вызвать метод `addChangeListener()`. В этом интерфейсе объявлен лишь один метод `stateChanged()`. При его выполнении извлекается значение, на котором установлен ползунок, как показано ниже.

```

public void stateChanged(ChangeEvent event)
{
    JSlider slider = (JSlider) event.getSource();
    int value = slider.getValue();
    . . .
}

```

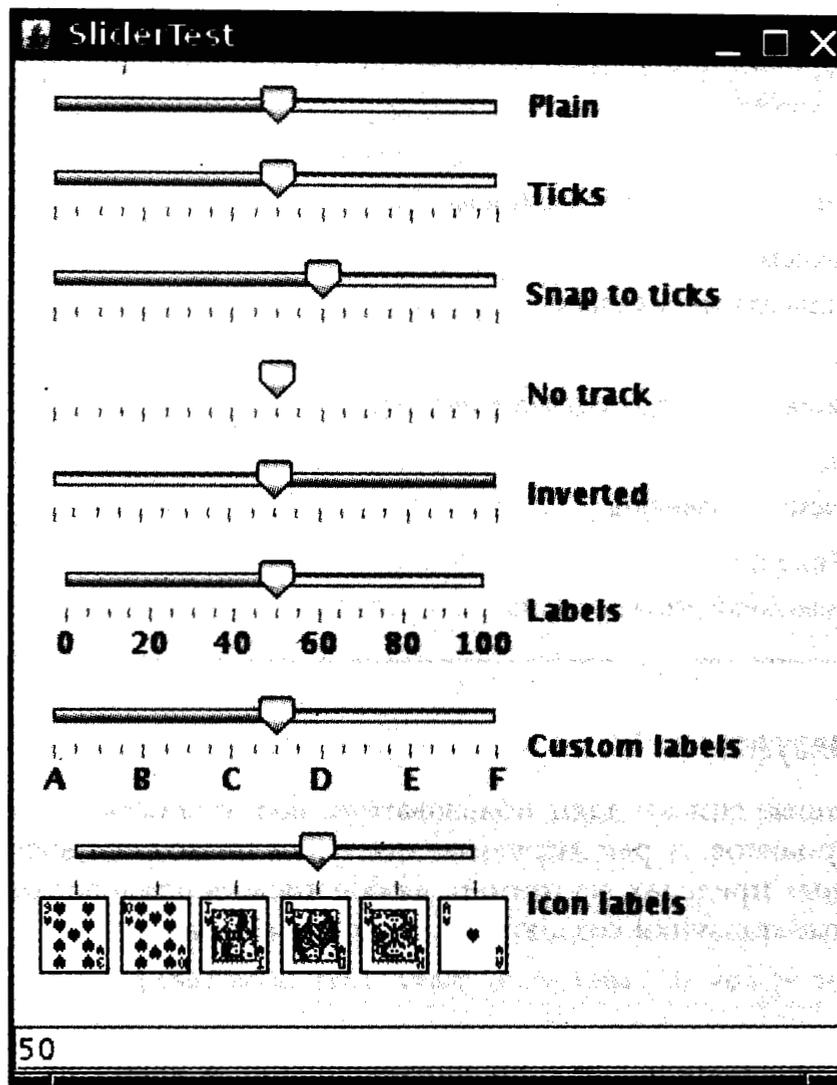


Рис. 9.18. Регулируемые ползунки

Регулируемый ползунок можно дополнить *отметками*, как на шкале. Так, в программе, рассматриваемой здесь в качестве примера, для второго ползунка задаются следующие установки:

```

slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);

```

Регулируемый ползунок снабжается основными отметками, следующими через каждые 20 единиц измерения, а также вспомогательными отметками, следующими через каждые 5 единиц измерения. Сами единицы измерения связываются со значениями, на которых устанавливается ползунок, и не имеют никакого отношения к пикселям на экране. В приведенном выше фрагменте кода лишь устанавливаются отметки регулируемого ползунка. А для того чтобы вывести их на экран, нужно сделать следующий вызов:

```

slider.setPaintTicks(true);

```



ВНИМАНИЕ! В режиме привязки к отметкам регулируемого ползунка данный компонент ведет себя не совсем предсказуемым образом. До тех пор, пока ползунок не установится точно на отметке, приемник изменений получает значения, не соответствующие отметкам. Так, если переместить ползунок щелчком кнопкой мыши, он все равно не установится на следующей отметке в режиме привязки к отметкам.

Сделав следующий вызов, можно обозначить основные отметки регулируемого ползунка:

```
slider.setPaintLabels(true);
```

Так, если регулируемый ползунок перемещается в пределах от 0 до 100 единиц, а промежутки между основными отметками составляет 20 единиц, отметки такого ползунка будут обозначены цифрами 0, 20, 40, 60, 80 и 100.

Кроме цифр, отметки можно, например, обозначить символьными строками или пиктограммами (см. рис. 9.18), хотя сделать это не так-то просто. Сначала нужно заполнить хеш-таблицу с ключами типа `Integer` и значениями типа `Component`, а затем вызвать метод `setLabelTable()`. Соответствующие компоненты располагаются под обозначаемыми отметками ползунка. Для этой цели обычно служат объекты типа `JLabel`. Ниже показано, как обозначить отметки регулируемого ползунка буквами **A, B, C, D, E** и **F**.

```
Hashtable<Integer, Component> labelTable =  
    new Hashtable<Integer, Component>();  
labelTable.put(0, new JLabel("A"));  
labelTable.put(20, new JLabel("B"));  
.  
.  
.  
labelTable.put(100, new JLabel("F"));  
slider.setLabelTable(labelTable);
```

Подробнее хеш-таблицы рассматриваются в главе 13. В листинге 9.7 приведен пример программы, демонстрирующий построение регулируемого ползунка с отметками, обозначаемыми пиктограммами.



СОВЕТ. Если отметки и их обозначения не выводятся на экран, проверьте, вызываются ли методы `setPaintTicks(true)` и `setPaintLabels(true)`.

У четвертого регулируемого ползунка на рис. 9.18 отсутствует полоса перемещения. Подавить отображение полосы, по которой двигается ползунок, можно, сделав следующий вызов:

```
slider.setPaintTrack(false);
```

Для пятого регулируемого ползунка на этом же рисунке направление движения изменено с помощью приведенного ниже метода.

```
slider.setInverted(true);
```

Регулируемые ползунки, создаваемые в рассматриваемом здесь примере программы, демонстрируют различные визуальные эффекты. Для каждого ползунка установлен приемник событий, отображающий значение, на котором в текущий момент установлен данный ползунок, в текстовом поле, расположенном в нижней части окна.

Листинг 9.7. Исходный код из файла slider/SliderFrame.java

```
1 package slider;
2
3 import java.awt.*;
4 import java.util.*;
5 import javax.swing.*;
6 import javax.swing.event.*;
7
8 /**
9  * Фрейм с несколькими ползунками и текстовым полем для показа
10 * значений, на которых по очереди устанавливаются ползунки
11 */
12 public class SliderFrame extends JFrame
13 {
14     private JPanel sliderPanel;
15     private JTextField textField;
16     private ChangeListener listener;
17
18     public SliderFrame()
19     {
20         sliderPanel = new JPanel();
21         sliderPanel.setLayout(new GridBagLayout());
22
23         // общий приемник событий для всех ползунков
24         listener = new ChangeListener()
25         {
26             public void stateChanged(ChangeEvent event)
27             {
28                 // обновить текстовое поле, если выбранный ползунок
29                 // установится на отметке с другим значением
30                 JSlider source = (JSlider) event.getSource();
31                 textField.setText("" + source.getValue());
32             }
33         };
34         // ввести простой ползунок
35         JSlider slider = new JSlider();
36         addSlider(slider, "Plain");
37
38         // ввести ползунок с основными и неосновными отметками
39         slider = new JSlider();
40         slider.setPaintTicks(true);
41         slider.setMajorTickSpacing(20);
42         slider.setMinorTickSpacing(5);
43         addSlider(slider, "Ticks");
44
45         // ввести ползунок, привязываемый к отметкам
46         slider = new JSlider();
47         slider.setPaintTicks(true);
48         slider.setSnapToTicks(true);
49         slider.setMajorTickSpacing(20);
50         slider.setMinorTickSpacing(5);
51         addSlider(slider, "Snap to ticks");
52
53         // ввести ползунок без отметок
54         slider = new JSlider();
55         slider.setPaintTicks(true);
56         slider.setMajorTickSpacing(20);
```

```
57 slider.setMinorTickSpacing(5);
58 slider.setPaintTrack(false);
59 addSlider(slider, "No track");
60
61 // ввести обращенный ползунок
62 slider = new JSlider();
63 slider.setPaintTicks(true);
64 slider.setMajorTickSpacing(20);
65 slider.setMinorTickSpacing(5);
66 slider.setInverted(true);
67 addSlider(slider, "Inverted");
68
69 // ввести ползунок с числовыми обозначениями отметок
70 slider = new JSlider();
71 slider.setPaintTicks(true);
72 slider.setPaintLabels(true);
73 slider.setMajorTickSpacing(20);
74 slider.setMinorTickSpacing(5);
75 addSlider(slider, "Labels");
76
77 // ввести ползунок с буквенными обозначениями отметок
78 slider = new JSlider();
79 slider.setPaintLabels(true);
80 slider.setPaintTicks(true);
81 slider.setMajorTickSpacing(20);
82 slider.setMinorTickSpacing(5);
83
84 Dictionary<Integer, Component> labelTable = new Hashtable<>();
85 labelTable.put(0, new JLabel("A"));
86 labelTable.put(20, new JLabel("B"));
86 labelTable.put(40, new JLabel("C"));
87 labelTable.put(60, new JLabel("D"));
88 labelTable.put(80, new JLabel("E"));
89 labelTable.put(100, new JLabel("F"));
90
91 slider.setLabelTable(labelTable);
92 addSlider(slider, "Custom labels");
93
94 // ввести ползунок с пиктограммными обозначениями отметок
95 slider = new JSlider();
96 slider.setPaintTicks(true);
97 slider.setPaintLabels(true);
98 slider.setSnapToTicks(true);
99 slider.setMajorTickSpacing(20);
100 slider.setMinorTickSpacing(20);
101
102 labelTable = new Hashtable<Integer, Component>();
103
104 // ввести изображения игральных карт
105 labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
106 labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
107 labelTable.put(40, new JLabel(new ImageIcon("jack.gif")));
108 labelTable.put(60, new JLabel(new ImageIcon("queen.gif")));
109 labelTable.put(80, new JLabel(new ImageIcon("king.gif")));
110 labelTable.put(100, new JLabel(new ImageIcon("ace.gif")));
111
112 slider.setLabelTable(labelTable);
113 addSlider(slider, "Icon labels");
```

```

114
115 // ввести текстовое поле для показа значения, на котором
116 // установлен выбранный в настоящий момент ползунок
117 textField = new JTextField();
118 add/sliderPanel, BorderLayout.CENTER);
119 add(textField, BorderLayout.SOUTH);
120 pack();
121 }
122
123 /**
124  * Вводит ползунки на панели и привязывает к ним приемник событий
125  * @param s Ползунок
126  * @param description Описание ползунка
127  */
128 public void addSlider(JSlider s, String description)
129 {
130     s.addChangeListener(listener);
131     JPanel panel = new JPanel();
132     panel.add(s);
133     panel.add(new JLabel(description));
134     panel.setAlignmentX(Component.LEFT_ALIGNMENT);
135     GridBagConstraints gbc = new GridBagConstraints();
136     gbc.gridy = sliderPanel.getComponentCount();
137     gbc.anchor = GridBagConstraints.WEST;
138     sliderPanel.add(panel, gbc);
139 }
140 }

```

javax.swing.JSlider 1.2

- JSlider()
- JSlider(int *direction*)
- JSlider(int *min*, int *max*)
- JSlider(int *min*, int *max*, int *initialValue*)
- JSlider(int *direction*, int *min*, int *max*, int *initialValue*)

Создают горизонтальный регулируемый ползунок с заданным направлением перемещения, минимальным и максимальным значениями.

Параметры:

<i>direction</i>	Одна из констант <code>SwingConstants.HORIZONTAL</code> или <code>swingConstants.VERTICAL</code> . По умолчанию выбирается константа <code>SwingConstants.HORIZONTAL</code> , задающая перемещение ползунка по горизонтали
<i>min, max</i>	Минимальное и максимальное значения для установки ползунка. По умолчанию эти значения равны 0 и 100 соответственно
<i>initialValue</i>	Начальное значение для установки ползунка. По умолчанию это значение равно 50 .

- `void setPaintTicks(boolean b)`
 Если параметр *b* принимает логическое значение `true`, то отображаются отметки, на которых устанавливается ползунок.
- `void setMajorTickSpacing(int units)`
- `void setMinorTickSpacing(int units)`
 Устанавливают разные единицы измерения для основных и неосновных отметок.
- `void setPaintLabels(boolean b)`
 Если параметр *b* принимает логическое значение `true`, то отображаются обозначения меток.
- `void setLabelTable(Dictionary table)`
 Устанавливает компоненты для обозначения отметок. Каждая пара "ключ-значение" представлена в таблице в следующей форме:
- `new Integer(значение) / компонент.`
- `void setSnapToTicks(boolean b)`
 Если параметр *b* принимает логическое значение `true`, то ползунок устанавливается на ближайшей отметке после каждого перемещения.
- `void setPaintTrack(boolean b)`
 Если значение параметра *b* принимает логическое значение `true`, то отображается полоса, по которой перемещается ползунок.

Меню

В начале этой главы были рассмотрены наиболее употребительные компоненты пользовательского интерфейса, которые можно расположить в окне, в том числе разнообразие кнопок, текстовые поля и комбинированные списки. В библиотеке Swing предусмотрены также ниспадающие меню, хорошо известные всем, кому когда-нибудь приходилось пользоваться программами с ГПИ.

Строка меню в верхней части окна содержит названия ниспадающих меню. Щелкая на таком имени кнопкой мыши, пользователь открывает меню, состоящее из пунктов и подменю. Если пользователь щелкнет на пункте меню, все меню закроются и программе будет отправлено соответствующее уведомление. На рис. 9.19 показано типичное меню, состоящее из пунктов и подменю.

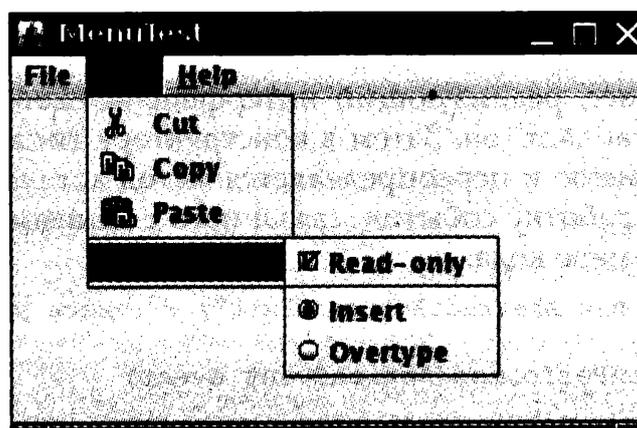


Рис. 9.19. Меню, состоящее из пунктов и подменю

Создание меню

Создать меню совсем не трудно. Для этого сначала создается строка меню следующим образом:

```
JMenuBar menuBar = new JMenuBar();
```

Строка меню — это обычный компонент, который можно расположить где угодно. Как правило, она располагается в верхней части фрейма с помощью метода `setJMenuBar()`, как показано ниже.

```
frame.setJMenuBar(menuBar);
```

Для каждого меню создается свой объект следующим образом:

```
menuBar.add(editMenu);
```

Меню верхнего уровня размещаются в строке меню, как показано ниже.

```
menuBar.add(editMenu);
```

Затем в объект меню вводятся пункты, разделители и подменю:

```
JMenuItem pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
JMenu optionsMenu = . . .; // подменю
editMenu.add(optionsMenu);
```

Разделители показаны на рис. 9.19. Они отображаются под пунктами меню Paste (Вставка) и Read-only (Только для чтения). Когда пользователь выбирает меню, инициируется событие действия. Следовательно, для каждого пункта меню следует определить обработчик, как показано ниже.

```
ActionListener listener = . . .;
pasteItem.addActionListener(listener);
```

Существует удобный метод `JMenu.add(String s)`, позволяющий добавлять новый пункт в конце меню, например, так, как показано ниже.

```
editMenu.add("Paste");
```

Этот метод возвращает созданный пункт меню, для которого можно легко задать обработчик:

```
JMenuItem pasteItem = editMenu.add("Paste");
pasteItem.addActionListener(listener);
```

Очень часто пункты меню связываются с командами, которые могут активизировать другие элементы пользовательского интерфейса, например кнопки. Как упоминалось в главе 8, команды задаются с помощью объектов типа `Action`. Сначала следует определить класс, реализующий интерфейс `Action`. Обычно такой класс расширяет класс `AbstractAction`. Затем в конструкторе типа `AbstractAction` указывается метка пункта меню и переопределяется метод `actionPerformed()`, что позволяет реализовать обработку события, связанного с данным пунктом меню, как в приведенном ниже примере кода.

```
Action exitAction = new AbstractAction("Exit") // здесь указывается пункт меню
{
    public void actionPerformed(ActionEvent event)
    {
        здесь следует код выполняемого действия
    }
}
```

```
        System.exit(0);
    }
};
```

Затем объект типа Action вводится в меню следующим образом:

```
JMenuItem exitItem = fileMenu.add(exitAction);
```

В итоге новый пункт вводится в меню по имени действия. Объект этого действия становится обработчиком. Такой прием позволяет заменить следующие строки кода:

```
JMenuItem exitItem = new JMenuItem(exitAction);
fileMenu.add(exitItem);
```

javax.swing.JMenu 1.2

- `JMenu(String label)`
Создает меню с указанной меткой.
- `JMenuItem add(JMenuItem item)`
Добавляет пункт (или целое меню).
- `JMenuItem add(String label)`
Добавляет пункт в меню с указанной меткой и возвращает этот пункт меню.
- `JMenuItem add(Action a)`
Добавляет пункт и связанное с ним действие и возвращает этот пункт.
- `void addSeparator()`
Добавляет в меню разделитель.
- `JMenuItem insert(JMenuItem menu, int index)`
Добавляет новый пункт меню (или подменю) по указанному индексу.
- `JMenuItem insert(Action a, int index)`
Добавляет новый пункт меню и связанный с ним объект типа Action по указанному индексу.
- `void insertSeparator(int index)`
Добавляет в меню разделитель по указанному индексу.
Параметры: `index` Место для ввода разделителя
- `void remove(int index)`
- `void remove(JMenuItem item)`
Удаляют указанный пункт меню.

javax.swing.JMenuItem 1.2

- `JMenuItem(String label)`
Создает пункт меню с указанной меткой.
- `JMenuItem(Action a) 1.3`
Создает пункт меню для указанного действия.

javax.swing.AbstractButton 1.2

- void setAction(Action a) 1.3
Устанавливает действие для данной кнопки или пункта меню.

javax.swing.JFrame 1.2

- void setJMenuBar(JMenuBar menubar)
Устанавливает строку меню в данном фрейме.

Пиктограммы в пунктах меню

Пункты меню очень похожи на кнопки. В действительности класс JMenuItem расширяет класс AbstractButton. Как и кнопки, меню могут иметь текстовую метку, пиктограмму или и то и другое. Пиктограмму можно, с одной стороны, указать в конструкторе JMenuItem(String, Icon) или JMenuItem(Icon), а с другой стороны, задать с помощью метода setIcon(), унаследованного классом JMenuItem от класса AbstractButton. Ниже приведен соответствующий пример.

```
JMenuItem cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));
```

На рис. 9.19 показано меню с пиктограммами. По умолчанию названия пунктов меню располагаются справа от пиктограмм. Если же требуется, чтобы пиктограммы находились справа от названий пунктов меню, воспользуйтесь методом setHorizontalTextPosition(), унаследованным в классе JMenuItem от класса AbstractButton. Например, в приведенной ниже строке кода текст пункта меню размещается слева от пиктограммы.

```
cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
```

Пиктограмму можно также связать с объектом типа Action следующим образом:

```
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
```

Если пункт меню создается независимо от действия, представленного объектом типа Action, то значением поля Action.NAME становится название пункта меню, а значением поля Action.SMALL_ICON — его пиктограмма. Кроме того, пиктограмму можно задать в конструкторе класса AbstractAction, как показано ниже.

```
cutAction = new
  AbstractAction("Cut", new ImageIcon("cut.gif"))
  {
    public void actionPerformed(ActionEvent event)
    {
      здесь следует код выполняемого действия
    }
  };
```

javax.swing.JMenuItem 1.2

- JMenuItem(String label, Icon icon)
Создает пункт меню с указанными меткой и пиктограммой.

javax.swing.AbstractButton 1.2

- void setHorizontalTextPosition(int pos)
- **Задаёт взаимное расположение текста надписи и пиктограммы.**
Параметры: pos Константа SwingConstants.RIGHT (текст справа от пиктограммы)
или SwingConstants.LEFT (текст слева от пиктограммы)

javax.swing.AbstractAction 1.2

- AbstractAction(String name, Icon smallIcon)
Создает объект типа AbstractAction с указанным именем и пиктограммой.

Пункты меню с флажками и кнопками-переключателями

Пункты меню могут также содержать *флажки* или *кнопки-переключатели* (см. рис. 9.19). Когда пользователь щелкает кнопкой мыши на пункте меню, флажок автоматически устанавливается или сбрасывается, а состояние кнопки-переключателя изменяется в соответствии с выбранным пунктом.

Помимо внешнего вида таких флажков и кнопок-переключателей, они мало чем отличаются от обычных пунктов меню. Ниже в качестве примера показано, каким образом создается пункт меню с флажком.

```
JCheckBoxMenuItem readonlyItem = new JCheckBoxMenuItem("Read-only");  
optionsMenu.add(readonlyItem);
```

Пункты меню с кнопками-переключателями действуют точно так же, как и обычные кнопки-переключатели. Для этого в меню следует добавить группу кнопок-переключателей. Когда выбирается одна из таких кнопок, все остальные автоматически отключаются. Ниже приведен пример создания пунктов меню с кнопками-переключателями.

```
ButtonGroup group = new ButtonGroup();  
JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Insert");  
insertItem.setSelected(true);  
JRadioButtonMenuItem overtypeItem = new JRadioButtonMenuItem("Overtypе");  
group.add(insertItem);  
group.add(overtypeItem);  
optionsMenu.add(insertItem);  
optionsMenu.add(overtypeItem);
```

В этих пунктах меню совсем не обязательно определять, когда именно пользователь сделал выбор. Вместо этого для проверки текущего состояния пункта меню

достаточно вызвать метод `isSelected()`. (Разумеется, это означает, что в какой-то переменной экземпляра придется хранить ссылку на данный пункт меню.) Кроме того, задать состояние пункта меню можно с помощью метода `setSelected()`.

javax.swing.JCheckBoxMenuItem 1.2

- `JCheckBoxMenuItem(String label)`
Создает пункт меню с флажком и заданной меткой.
- `JCheckBoxMenuItem(String label, boolean state)`
Создает пункт меню с флажком и заданными меткой и состоянием (если параметр `state` принимает логическое значение `true`, то пункт считается выбранным).

javax.swing.JRadioButtonMenuItem 1.2

- `JRadioButtonMenuItem(String label)`
Создает пункт меню с кнопкой-переключателем и заданной меткой.
- `JRadioButtonMenuItem(String label, boolean state)`
Создает пункт меню с кнопкой-переключателем и заданными меткой и состоянием (если параметр `state` принимает логическое значение `true`, то пункт считается выбранным).

javax.swing.AbstractButton 1.2

- `boolean isSelected()`
Возвращает состояние пункта меню.
- `void setSelected(boolean state)`
Устанавливает состояние пункта меню (если параметр `state` принимает логическое значение `true`, то пункт считается выбранным).

Всплывающие меню

Всплывающие, или контекстные, меню не связаны со строкой меню, а появляются в произвольно выбранном месте на экране (рис. 9.20).

Всплывающее меню создается точно так же, как и обычное, за исключением того, что у него отсутствует заголовок. Ниже приведен типичный пример создания всплывающего меню в коде.

```
JPopupMenu popup = new JPopupMenu();
```

Пункты меню добавляются как обычно:

```
JMenuItem item = new JMenuItem("Cut");  
item.addActionListener(listener);  
popup.add(item);
```

В отличие от строки меню, которая всегда находится в верхней части фрейма, всплывающее меню следует явным образом выводить на экран с помощью метода

`show()`. При вызове этого метода задается родительский компонент и расположение всплывающего меню в его системе координат:

```
popup.show(panel, x, y);
```

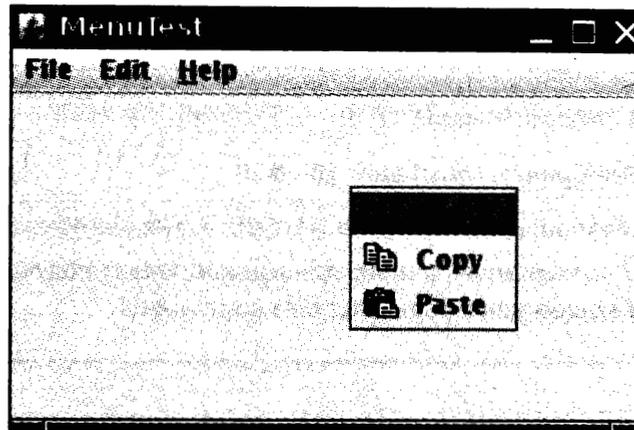


Рис. 9.20. Всплывающее меню

Обычно всплывающее меню отображается на экране, когда пользователь щелкает специально предназначенной для этого кнопкой — так называемым *триггером всплывающего меню*. В Windows и Linux это обычно правая кнопка мыши. Для всплывания меню после щелчка кнопкой мыши вызывается следующий метод:

```
component.setComponentPopupMenu(popup);
```

Нередко один компонент приходится размещать внутри другого компонента, с которым связано всплывающее меню. Для того чтобы производный компонент наследовал меню родительского компонента, нужно сделать следующий вызов:

```
child.setInheritsPopupMenu(true);
```

javax.swing.JPopupMenu 1.2

- `void show(Component c, int x, int y)`

Отображает всплывающее меню.

Параметры: `c` Компонент, посредством которого появляется всплывающее меню
`x, y` Координаты (в системе компонента `c`) левого верхнего угла всплывающего меню

- `boolean isPopupTrigger(MouseEvent event) 1.3`

Возвращает логическое значение `true`, если событие инициировано триггером всплывающего меню (как правило, нажатием правой кнопки мыши).

java.awt.event.MouseEvent 1.1

- `boolean isPopupTrigger()`

Возвращает логическое значение `true`, если данное событие инициировано триггером всплывающего меню (как правило, нажатием правой кнопки мыши).

javax.swing.JComponent 1.2

- `JPopupMenu getComponentPopupMenu()` 5.0
- `void setComponentPopupMenu(JPopupMenu popup)` 5.0
Устанавливают или возвращают всплывающее меню для данного компонента.
- `boolean getInheritsPopupMenu()` 5.0
- `void setInheritsPopupMenu(boolean b)` 5.0
Устанавливают или возвращают свойство `inheritsPopupMenu`. Если данное свойство установлено, а вместо всплывающего меню данный компонент получает пустое значение `null`, то вызывается всплывающее меню родительского компонента.

Клавиши быстрого доступа и оперативные клавиши

Опытному пользователю удобно выбирать пункты меню с помощью *клавиш быстрого доступа*. Связать пункт меню с клавишей быстрого доступа можно, задав эту клавишу в конструкторе пункта меню следующим образом:

```
JMenuItem aboutItem = new JMenuItem("About", 'A');
```

Буква в названии пункта меню, соответствующая клавише быстрого доступа, выделяется подчеркиванием (рис. 9.21). Так, если клавиша быстрого доступа задана с помощью приведенного выше выражения, то метка отобразится как `About`, т.е. с подчеркнутой буквой `A`. Теперь для выбора данного пункта меню пользователю достаточно нажать клавишу `<A>`. (Если буква, соответствующая назначенной клавише, не входит в название пункта меню, она не отображается на экране, но при ее нажатии этот пункт все равно будет выбран. Естественно, что польза от таких “невидимых” клавиш быстрого доступа сомнительна.)

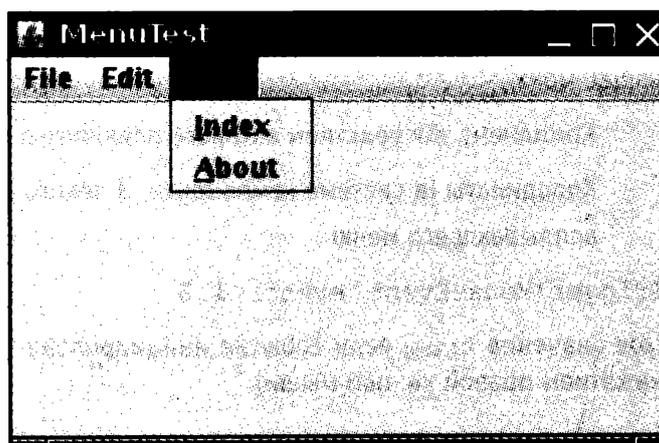


Рис. 9.21. Пункты меню, для которых назначены клавиши быстрого доступа

Задавая клавишу быстрого доступа, не всегда целесообразно выделять первое вхождение соответствующей буквы в названии пункта меню. Так, если для пункта `Save As` (Сохранить) назначена клавиша `<A>`, то гораздо уместнее выделить подчеркиванием букву `A` в слове `As` (`Save As`). Для того чтобы указать подчеркиваемый символ, следует вызвать метод `setDisplayMnemonicIndex()`. А имея в своем распоряжении

объект типа Action, клавишу быстрого доступа можно назначить, указав нужное значение в поле Action.MNEMONIC_KEY следующим образом:

```
cutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
```

Букву, соответствующую клавише быстрого доступа, следует задавать только в конструкторе пункта меню (но не в конструкторе всего меню). Чтобы связать какую-нибудь клавишу с меню в целом, следует вызвать метод setMnemonic():

```
JMenu helpMenu = new JMenu("Help");
helpMenu.setMnemonic('H');
```

Теперь, чтобы сделать выбор из строки меню, достаточно нажать клавишу <Alt> вместе с клавишей назначенной буквы. Например, чтобы выбрать меню Help (Справка), следует нажать комбинацию клавиш <Alt+H>.

Клавиши быстрого доступа позволяют выбрать пункт в уже открытом меню. С другой стороны, *оперативные клавиши* позволяют выбрать пункт, не открывая меню. Например, во многих прикладных программах предусмотрены комбинации клавиш <Ctrl+O> и <Ctrl+S> для пунктов Open (Открыть) и Save (Сохранить) меню File (Файл). Для связывания оперативных клавиш с пунктом меню служит метод setAccelerator(). В качестве параметра он получает объект типа KeyStroke. Например, в приведенном ниже вызове комбинация оперативных клавиш <Ctrl+O> назначается для пункта меню openItem.

```
openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
```

При нажатии оперативных клавиш автоматически выбирается соответствующий пункт меню и инициируется событие таким же образом, как и при выборе пункта меню обычным способом. Оперативные клавиши можно связывать только с пунктами меню, но не с меню в целом. Они не открывают меню, а только инициируют событие, связанное с указанным пунктом меню.

В принципе оперативные клавиши связываются с пунктами меню таким же образом, как и с остальными компонентами из библиотеки Swing. (О том, как это делается, см. в главе 8.) Но если оперативные клавиши назначены для пункта меню, то соответствующая комбинация клавиш автоматически отображается в меню (рис. 9.22).

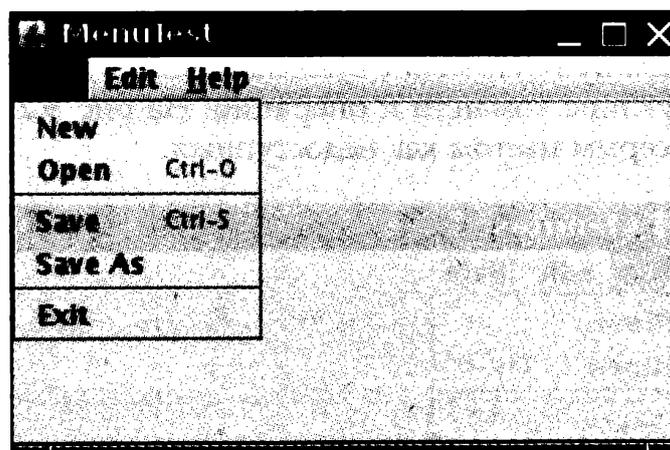


Рис. 9.22. Отображение оперативных клавиш в пунктах меню



НА ЗАМЕТКУ! При нажатии комбинации клавиш <Alt+F4> в Windows закрывается текущее окно. Но эти оперативные клавиши не имеют никакого отношения к Java. Они определены в данной операционной системе и всегда инициируют событие типа WindowClosing для активного окна, независимо от того, имеется ли в меню пункт Close.

javax.swing.JMenuItem 1.2

- JMenuItem(*String label*, *int mnemonic*)
- Создает пункт меню с указанной меткой и клавишей быстрого доступа.

Параметры:	<i>label</i>	Метка пункта меню
	<i>mnemonic</i>	Символ, мнемонически обозначающий клавишу быстрого доступа к пункту меню. В метке пункта меню он подчеркивается
- void setAccelerator(*KeyStroke k*)

Задает оперативную клавишу *k* для данного пункта меню. Соответствующая клавиша отображается в меню рядом с меткой данного пункта.

javax.swing.AbstractButton 1.2

- void setMnemonic(*int mnemonic*)
 Задает символ, мнемонически обозначающий клавишу быстрого доступа к кнопке. В метке кнопки этот символ подчеркивается.
- void setDisplayedMnemonicIndex(*int index*) 1.4
 Задает расположение подчеркиваемого символа. Вызывается в том случае, если выделять первое вхождение символа, мнемонически обозначающего клавишу быстрого доступа, нецелесообразно.

Разрешение и запрет доступа к пунктам меню

Иногда некоторые пункты меню должны выбираться лишь в определенном контексте. Так, если документ открыт лишь для чтения, то пункт меню Save не имеет смысла. Разумеется, этот пункт можно удалить методом `JMenu.remove()`, но пользователя может удивить постоянно изменяющееся меню. Поэтому лучше всего запретить доступ к некоторым пунктам меню, временно лишив пользователя возможности выполнять соответствующие команды и операции. На рис. 9.23 запрещенный пункт меню выделен светло-серым цветом как недоступный.

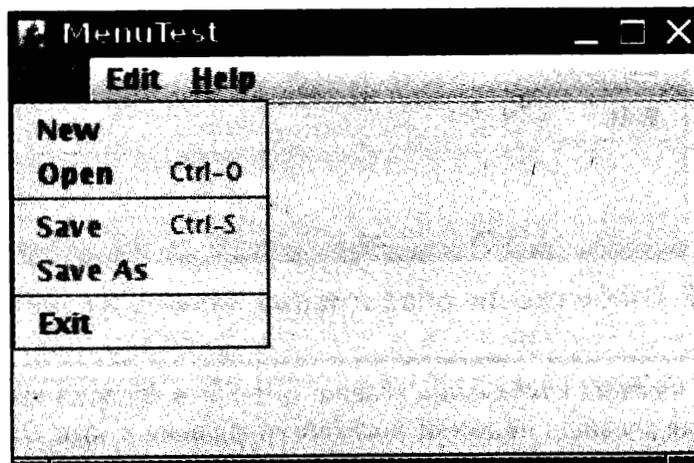


Рис. 9.23. Пункты меню, запрещенные для доступа

Для того чтобы разрешить или запретить доступ к пунктам меню, вызывается метод `setEnabled()`:

```
saveItem.setEnabled(false);
```

Существуют две методики разрешения и запрета доступа к пунктам меню. При всяком изменении состояния программы можно вызывать метод `setEnabled()` для соответствующего пункта меню. Например, открыв документ только для чтения, можно сделать недоступными пункты меню `Save` и `Save As`. С другой стороны, можно делать недоступными пункты меню непосредственно перед их отображением. Для этого нужно зарегистрировать обработчик событий, связанный с выбором меню. В состав пакета `javax.swing.event` входит интерфейс `MenuListener`, в котором объявлены следующие три метода:

```
void menuSelected(MenuEvent event)
void menuDeselected(MenuEvent event)
void menuCanceled(MenuEvent event)
```

Метод `menuSelected()` вызывается до отображения меню. Это самый подходящий момент для того, чтобы разрешить или запретить доступ к пунктам меню. В приведенном ниже фрагменте кода показано, как пункты меню `Save` и `Save As` делаются доступными и недоступными в зависимости от состояния флажка `Read Only` (Только для чтения).

```
public void menuSelected(MenuEvent event)
{
    saveAction.setEnabled(!readonlyItem.isSelected());
    saveAsAction.setEnabled(!readonlyItem.isSelected());
}
```



ВНИМАНИЕ! Запрещать доступ к пунктам меню непосредственно перед их отображением вполне благоразумно, но такая методика не подходит для пунктов меню, имеющих назначенные для них оперативные клавиши. При нажатии оперативной клавиши меню вообще не открывается, поэтому доступ к выбираемому пункту меню не будет запрещен, а следовательно, будет инициировано выполнение соответствующей команды.

javax.swing.JMenuItem 1.2

- `void setEnabled(boolean b)`
Разрешает и запрещает доступ к пункту меню.

javax.swing.event.MenuListener 1.2

- `void menuSelected(MenuEvent e)`
Вызывается, когда меню уже выбрано, но еще не открыто.
- `void menuDeselected(MenuEvent e)`
Вызывается, когда меню уже закрыто.
- `void menuCanceled(MenuEvent e)`
Вызывается, когда обращение к меню отменено; если, например, пользователь щелкнет кнопкой мыши за пределами меню.

В листинге 9.8 приведен исходный код программы, где формируется ряд меню. На примере этой программы демонстрируются все особенности меню, описанные в этом разделе: вложенные меню, недоступные пункты меню, флажки и кнопки-переключатели в пунктах меню, а также клавиши быстрого доступа и оперативные клавиши выбора пунктов меню.

Листинг 9.8. Исходный код из файла menu/MenuFrame.java

```
1 package menu;
2
3 import java.awt.event.*;
4 import javax.swing.*;
5
6 /**
7  * Фрейм с образцом строки меню
8  */
9 public class MenuFrame extends JFrame
10 {
11     private static final int DEFAULT_WIDTH = 300;
12     private static final int DEFAULT_HEIGHT = 200;
13     private Action saveAction;
14     private Action saveAsAction;
15     private JCheckBoxMenuItem readonlyItem;
16     private JPopupMenu popup;
17
18     /**
19      * Обработчик действий, выводящий имя действия в поток System.out
20      */
21     class TestAction extends AbstractAction
22     {
23         public TestAction(String name)
24         {
25             super(name);
26         }
27
28         public void actionPerformed(ActionEvent event)
29         {
30             System.out.println(getValue(Action.NAME) + " selected.");
31         }
32     }
33
34     public MenuFrame()
35     {
36         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
37
38         JMenu fileMenu = new JMenu("File");
39         fileMenu.add(new TestAction("New"));
40
41         // продемонстрировать применение оперативных клавиш
42         JMenuItem openItem = fileMenu.add(new TestAction("Open"));
43         openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
44
45         fileMenu.addSeparator();
46         saveAction = new TestAction("Save");
47         JMenuItem saveItem = fileMenu.add(saveAction);
48         saveItem.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));
49
50         saveAsAction = new TestAction("Save As");
```

```
51 fileMenu.add(saveAsAction);
52 fileMenu.addSeparator();
53
54 fileMenu.add(new AbstractAction("Exit")
55 {
56     public void actionPerformed(ActionEvent event)
57     {
58         System.exit(0);
59     }
60 });
61
62 // продемонстрировать применение флажков и кнопок-переключателей
63 readonlyItem = new JCheckBoxMenuItem("Read-only");
64 readonlyItem.addActionListener(new ActionListener()
65 {
66     public void actionPerformed(ActionEvent event)
67     {
68         boolean saveOk = !readonlyItem.isSelected();
69         saveAction.setEnabled(saveOk);
70         saveAsAction.setEnabled(saveOk);
71     }
72 });
73
74 ButtonGroup group = new ButtonGroup();
75
76 JRadioButtonMenuItem insertItem =
77     new JRadioButtonMenuItem("Insert");
78 insertItem.setSelected(true);
79 JRadioButtonMenuItem overtypeItem =
80     new JRadioButtonMenuItem("Overtypе");
81
82 group.add(insertItem);
83 group.add(overtypеItem);
84
85 // продемонстрировать применение пиктограмм
86 Action cutAction = new TestAction("Cut");
87 cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
88 Action copyAction = new TestAction("Copy");
89 copyAction.putValue(Action.SMALL_ICON, new ImageIcon("copy.gif"));
90 Action pasteAction = new TestAction("Paste");
91 pasteAction.putValue(Action.SMALL_ICON, new ImageIcon("paste.gif"));
92
93 JMenu editMenu = new JMenu("Edit");
94 editMenu.add(cutAction);
95 editMenu.add(copyAction);
96 editMenu.add(pasteAction);
97
98 // продемонстрировать применение вложенных меню
99 JMenu optionMenu = new JMenu("Options");
100
101 optionMenu.add(readonlyItem);
102 optionMenu.addSeparator();
103 optionMenu.add(insertItem);
104 optionMenu.add(overtypеItem);
105
106 editMenu.addSeparator();
107 editMenu.add(optionMenu);
108
109 // продемонстрировать применение клавиш быстрого доступа
```

```
110 JMenu helpMenu = new JMenu("Help");
111 helpMenu.setMnemonic('H');
112
113 JMenuItem indexItem = new JMenuItem("Index");
114 indexItem.setMnemonic('I');
115 helpMenu.add(indexItem);
116
117 // назначить клавишу быстрого доступа, используя объект действия
118 Action aboutAction = new TestAction("About");
119 aboutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
120 helpMenu.add(aboutAction);
121
122 // ввести все меню верхнего уровня в строку меню
123 JMenuBar menuBar = new JMenuBar();
124 setJMenuBar(menuBar);
125
126 menuBar.add(fileMenu);
127 menuBar.add(editMenu);
128 menuBar.add(helpMenu);
129
130 // продемонстрировать применение всплывающих меню
131 JPopupMenu popup = new JPopupMenu();
132 popup.add(cutAction);
133 popup.add(copyAction);
134 popup.add(pasteAction);
135
136 JPanel panel = new JPanel();
137 panel.setComponentPopupMenu(popup);
138 add(panel);
139
140 // следующая строка кода служит для того, чтобы
141 // обойти программную ошибку 4966109
142 panel.addMouseListener(new MouseAdapter() {});
143 }
144 }
145 }
```

Панели инструментов

Панель инструментов представляет собой ряд кнопок, обеспечивающих быстрый доступ к наиболее часто используемым командам (рис. 9.24).

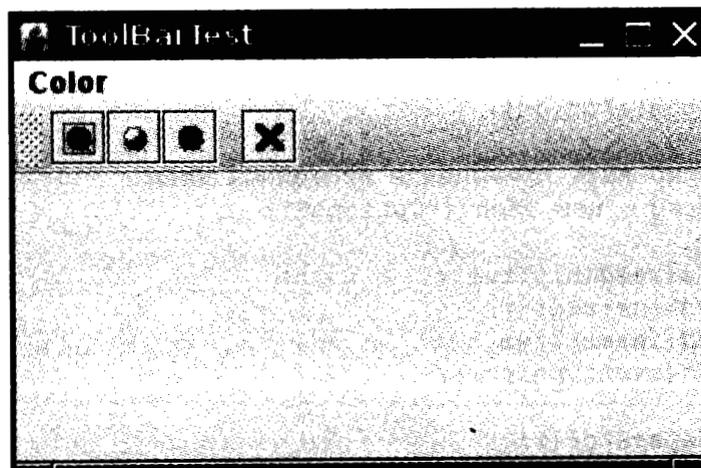


Рис. 9.24. Панель инструментов

Панель инструментов отличается от остальных элементов пользовательского интерфейса тем, что ее можно перетаскивать на любую из четырех сторон фрейма (рис. 9.25). При отпускании кнопки мыши панель инструментов фиксируется на новом месте (рис. 9.26).

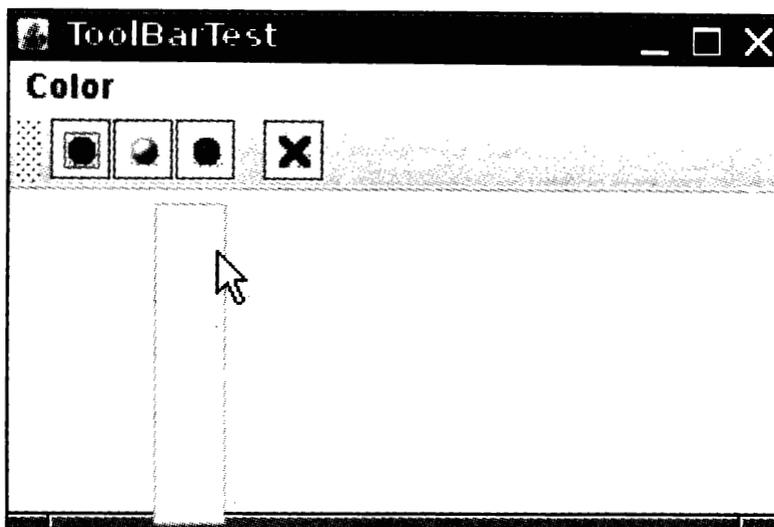


Рис. 9.25. Перетаскивание панели инструментов

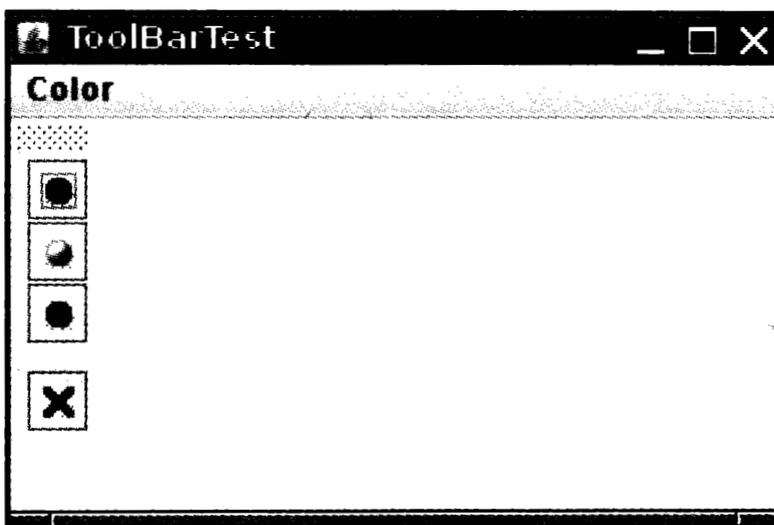


Рис. 9.26. Новое местоположение панели инструментов после перетаскивания



НА ЗАМЕТКУ! Перетаскивание панели инструментов допускается только в том случае, если она размещается в контейнере диспетчером граничной компоновки или любым другим диспетчером, поддерживающим расположение компонентов в северной, южной, восточной и западной областях фрейма.

Панель инструментов может быть даже обособленной от фрейма. Такая панель содержится в своем собственном фрейме (рис. 9.27). Если пользователь закрывает фрейм, содержащий обособленную панель инструментов, она перемещается в исходный фрейм.

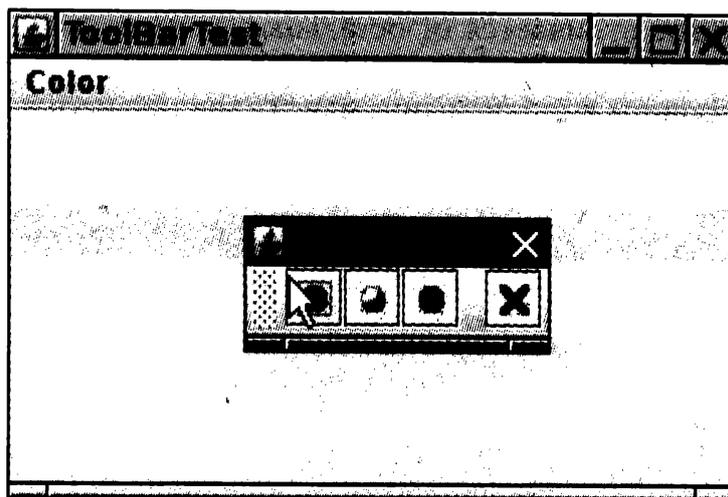


Рис. 9.27. Обособленная панель инструментов

Панель инструментов легко запрограммировать. Ниже приведен пример создания панели и добавления к ней компонента.

```
JToolBar bar = new JToolBar();
bar.add(blueButton);
```

В классе `JToolBar` имеются также методы, предназначенные для добавления объекта типа `Action`. Для этого достаточно заполнить панель инструментов объектами типа `Action`, как показано ниже. Пиктограмма, соответствующая такому объекту, отображается на панели инструментов.

```
bar.add(blueAction);
```

Группы кнопок можно отделять друг от друга с помощью разделителей следующим образом:

```
bar.addSeparator();
```

Например, на панели инструментов, показанной на рис. 9.24, имеется разделитель третьей кнопки от четвертой. Обычно панель инструментов размещается в контейнере, как показано ниже.

```
add(bar, BorderLayout.NORTH);
```

Имеется также возможность указать заголовок панели, который появится, когда панель будет обособлена от фрейма. Ниже показано, каким образом заголовок панели указывается в коде.

```
bar = new JToolBar(titleString);
```

По умолчанию панель инструментов располагается горизонтально. А для того чтобы расположить ее вертикально, достаточно написать одну из следующих двух строк кода:

```
bar = new JToolBar(SwingConstants.VERTICAL)
```

или

```
bar = new JToolBar(titleString, SwingConstants.VERTICAL)
```

Чаще всего на панели инструментов располагаются кнопки. Но никаких ограничений на вид компонентов, которые можно размещать на панели инструментов, не существует. Например, на панели инструментов можно расположить комбинированный список.

Всплывающие подсказки

У панелей инструментов имеется следующий существенный недостаток: по внешнему виду кнопки трудно догадаться о ее назначении. В качестве выхода из этого затруднительного положения были изобретены *всплывающие подсказки*, которые появляются на экране, когда курсор наводится на кнопку. Текст всплывающей подсказки отображается внутри закрашенного прямоугольника (рис. 9.28). Когда же курсор отводится от кнопки, подсказка исчезает.

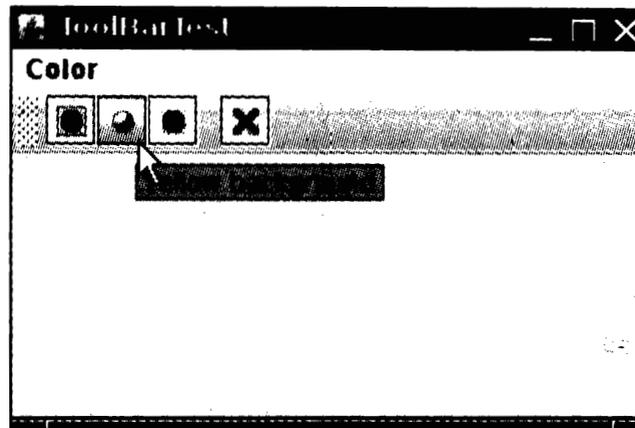


Рис. 9.28. Всплывающая подсказка

В библиотеке Swing допускается вводить всплывающие подсказки в любой объект типа `JComponent`, просто вызывая метод `setToolTipText()`:

```
exitButton.setToolTipText("Exit");
```

С другой стороны, если воспользоваться объектами типа `Action`, то всплывающая подсказка связывается с ключом `SHORT_DESCRIPTION` следующим образом:

```
exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");
```

В листинге 9.9 приведен исходный код программы, демонстрирующей, каким образом одни и те же объекты типа `Action` можно вводить в меню и на панели инструментов. Обратите внимание на то, что имена соответствующих действий появляются на экране как в названиях пунктов меню, так и во всплывающих подсказках к элементам панели инструментов.

Листинг 9.9. Исходный код из файла `toolBar/ToolBarTest.java`

```
1 package toolBar;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * @version 1.13 2007-06-12
8  * @author Cay Horstmann
9  */
10 public class ToolBarTest
11 {
12     public static void main(String[] args)
13     {
14         EventQueue.invokeLater(new Runnable()
15         {
```

```
16     public void run()
17     {
18         ToolBarFrame frame = new ToolBarFrame();
19         frame.setTitle("ToolBarTest");
20         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21         frame.setVisible(true);
22     }
23 });
24 }
25 }
```

javax.swing.JToolBar 1.2

- `JToolBar()`
- `JToolBar(String titleString)`
- `JToolBar(int orientation)`
- `JToolBar(String titleString, int orientation)`
Создают панель инструментов с заданной строкой заголовка и ориентацией. Параметр *orientation* может принимать значения констант `SwingConstants.HORIZONTAL` (по умолчанию) и `SwingConstants.VERTICAL`.
- `JButton add(Action a)`
Создает новую кнопку на панели инструментов с именем, кратким описанием, пиктограммой и обратным вызовом действия. Кнопка вводится в конце панели инструментов.
- `void addSeparator()`
Вводит разделитель в конце панели инструментов.

javax.swing.JComponent 1.2

- `void setToolTipText(String text)`
Задает текст для вывода во всплывающей подсказке, когда курсор мыши наводится на компонент.

Расширенные средства компоновки

В рассматривавшихся до сих пор примерах создания пользовательского интерфейса использовались только диспетчеры граничной, поточной и сеточной компоновки. Но для решения более сложных задач компоновки ГПИ этого явно недостаточно. В этом разделе будут подробно рассмотрены расширенные средства компоновки ГПИ.

У разработчиков приложений на платформе Windows может возникнуть следующий вопрос: зачем в Java столько внимания уделяется диспетчерам компоновки? Ведь во многих ИСР достаточно перетащить компоненты в диалоговое окно и выровнять их, используя соответствующие средства редактирования. Работая над крупным проектом, можно вообще не беспокоиться о компоновке элементов ГПИ, поскольку все связанные с этим хлопоты возьмет на себя квалифицированный разработчик пользовательского интерфейса.

Такому подходу присущ следующий существенный недостаток: если размеры компонентов пользовательского интерфейса изменяются, их приходится перекомпоновывать вручную. А почему размеры компонентов могут измениться? Это происходит в двух случаях. Во-первых, пользователь может выбрать более крупный шрифт для надписей на кнопках и других элементах диалогового окна. Если вы попытаетесь сделать это в Windows, то сами убедитесь, что многие приложения крайне неудачно справляются с подобной задачей. Кнопки не увеличиваются, а более крупный шрифт “втискивается” в те же самые размеры. Аналогичное затруднение может возникнуть при переводе пользовательского интерфейса на иностранный язык. Например, слово “Cancel” на немецкий язык переводится как “Abbrechen”. Если кнопка разработана таким образом, что на ней умещается только надпись “Cancel”, то немецкий перевод надписи на той же кнопке будет усечен.

Почему же кнопки, созданные на платформе Windows, не увеличиваются, чтобы вмещать надписи и метки? Это происходит потому, что разработчики интерфейса не предусмотрели никаких команд, задающих направление, в котором должны увеличиваться размеры кнопок. После того как элементы пользовательского интерфейса будут перемещены в диалоговое окно и выровнены, редактор диалогового окна просто забывает координаты и размеры всех компонентов. Таким образом, теряются сведения, позволяющие судить, почему компоненты были расположены именно таким образом, а не иначе.

Диспетчеры компоновки в Java справляются с расположением компонентов ГПИ много лучше. По существу, компоновка сводится к созданию инструкций, описывающих отношения между компонентами. Это особенно важно для работы с библиотекой AWT, где используются собственные элементы пользовательского интерфейса конкретной платформы. Размеры кнопки или раскрывающегося списка зависят от платформы и могут изменяться в широких пределах, а разработчикам приложений или апплетов *заранее* неизвестно, на какой именно платформе будет отображаться их пользовательский интерфейс. Такая степень изменчивости до некоторой степени исключается в библиотеке Swing. Если приложение принудительно принимает определенный визуальный стиль вроде Metal, оно будет выглядеть одинаково на всех платформах. Но если требуется предоставить пользователю возможность самому выбирать визуальный стиль ГПИ, то в таком случае остается только рассчитывать на гибкость диспетчеров компоновки в расположении компонентов ГПИ.

Начиная с версии Java 1.0, в состав библиотеки AWT входят средства *сеточно-контейнерной компоновки* для расположения компонентов по рядам и столбцам. Размеры ряда и столбца допускают гибкую установку, а компоненты могут занимать несколько рядов и столбцов. Такой диспетчер компоновки действует очень гибко, но в то же время он довольно сложен, причем настолько, что само словосочетание “сеточно-контейнерная компоновка” способно вызвать невольный трепет у программирующих на Java.

Безуспешные попытки разработать диспетчер компоновки, который избавил бы программистов от тирании сеточно-контейнерной компоновки, навели разработчиков библиотеки Swing на мысль о *блочной компоновке*. Как поясняется в документации на JDK, класс BoxLayout диспетчера блочной компоновки “осуществляет вложение многих панелей с горизонтальными и вертикальными размерами в разных сочетаниях, достигая такого же результата, как и класс GridLayout диспетчера сеточной компоновки, но без сложностей, присущих последней”. Но поскольку каждый компонуемый блок располагается независимо от других блоков, то блочная компоновка не подходит для упорядочения соседних компонентов как по вертикали, так и по горизонтали.

В версии Java SE 1.4 была предпринята еще одна попытка найти замену сеточно-контейнерной компоновке так называемой *пружинной компоновкой*. В соответствии с этой разновидностью компоновки для соединения отдельных компонентов в контейнере используются воображаемые пружины. При изменении размеров контейнера эти пружины растягиваются и сжимаются, регулируя таким образом расположение компонентов. На практике такой подход к компоновке оказался слишком трудоемким и запутанным, поэтому пружинная компоновка быстро канула в небытие.

В 2005 году разработчики NetBeans изобрели технологию Matisse, которая сочетает в себе инструмент и диспетчер компоновки (теперь она называется Swing GUI Builder). Разработчики пользовательского интерфейса применяют инструмент Swing GUI Builder, чтобы разместить компоненты в контейнере и указать те компоненты, по которым они должны быть выровнены. А инструмент переводит замысел разработчика в инструкции для диспетчера *групповой компоновки*. Это намного удобнее, чем написание кода управления компоновкой вручную. Диспетчер групповой компоновки вошел в состав Java SE. Даже если вы не пользуетесь NetBeans в качестве ИСР, то вам все равно стоит рассмотреть возможность применения доступного в этой среде построителя ГПИ. В этом случае вы можете разрабатывать ГПИ в NetBeans, вставляя полученный в итоге код в избранную вами ИСР.

В последующих разделах речь пойдет о диспетчере сеточно-контейнерной компоновки, потому что он применяется довольно широко и все еще представляет собой самый простой механизм генерирования кода компоновки для прежних версий Java. Попутно будет представлена методика, благодаря которой применение этого диспетчера компоновки может стать сравнительно безболезненным в типичных случаях.

Затем будут рассмотрены инструмент Swing GUI Builder и диспетчер групповой компоновки. Вам придется как следует разобраться в принципе действия диспетчера групповой компоновки, чтобы самим убедиться, что он генерирует корректные инструкции, когда вы располагаете компоненты ГПИ визуально. И в завершение темы диспетчеров компоновки будет показано, как вообще обойтись без них, располагая компоненты ГПИ вручную, и как написать свой собственный диспетчер компоновки.

Диспетчер сеточно-контейнерной компоновки

Диспетчер сеточно-контейнерной компоновки — предшественник всех остальных диспетчеров компоновки. Его можно рассматривать как диспетчер сеточной компоновки без ограничений, т.е. при сеточно-контейнерной компоновке ряды и столбцы могут иметь переменные размеры. Для того чтобы расположить крупный компонент, который не уместится в одной ячейке, несколько смежных ячеек можно соединить вместе. (Многие редакторы текста и HTML-документов предоставляют такие же возможности для построения таблиц: заполнение начинается с обычной сетки, а при необходимости некоторые ее ячейки соединяются вместе.) Компоненты совсем не обязательно должны заполнять всю ячейку, поэтому можно задать выравнивание внутри ячеек.

Рассмотрим в качестве примера диалоговое окно селектора шрифтов, приведенное на рис. 9.29. Оно состоит из следующих компонентов.

- Два комбинированных списка для выбора начертания и размера шрифта.
- Метки для обоих комбинированных списков.

- Два флажка для выбора полужирного и наклонного начертания шрифта.
- Текстовая область для отображения образца текстовой строки.

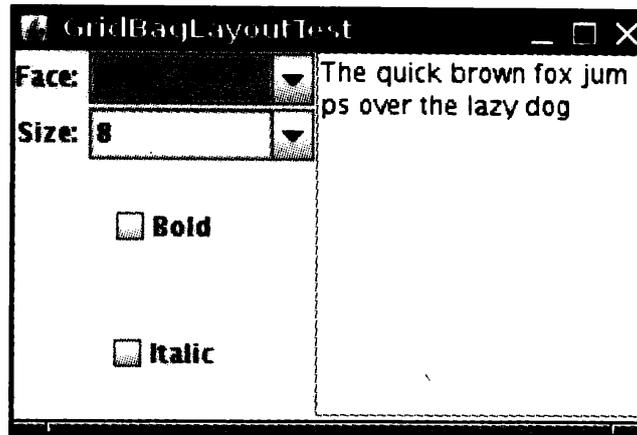


Рис. 9.29. Диалоговое окно селектора шрифтов

А теперь разделим диалоговое окно как контейнер на ячейки в соответствии с рис. 9.30. (Ряды и столбцы совсем не обязательно должны иметь одинаковые размеры.) Как видите, каждый флажок занимает два столбца, а текстовая область — четыре ряда.

Для описания компоновки, которое было бы понятно диспетчеру сеточно-контейнерной компоновки, выполните следующие действия.

1. Создайте объект типа `GridBagLayout`. Конструктору не обязательно знать, из какого числа рядов и столбцов состоит сетка. Он попытается впоследствии сам уточнить эти параметры по сведениям, полученным от вас.

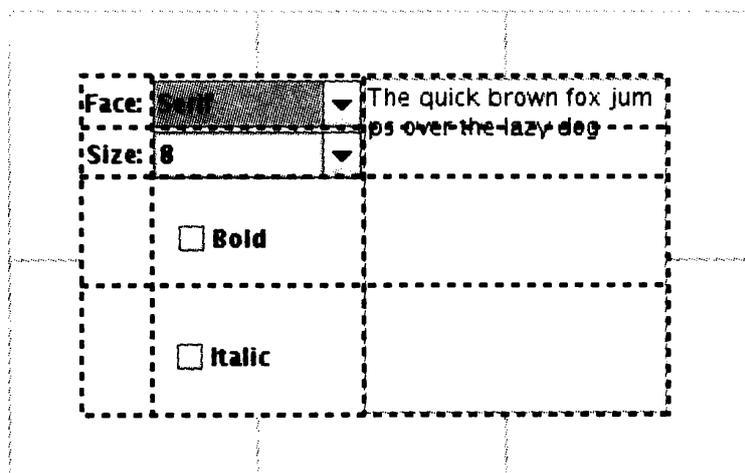


Рис. 9.30. Сетка разбиения диалогового окна, используемая для компоновки

2. Установите объект типа `GridLayout` в качестве диспетчера компоновки для данного компонента.
3. Создайте для каждого компонента объект типа `GridBagConstraints`. Установите соответствующие значения в полях этого объекта, чтобы задать расположение компонентов в сеточном контейнере.
4. Введите каждый компонент с ограничениями, сделав следующий вызов: `add(Component, constraints);`

Ниже приведен пример кода, в котором реализуются описанные выше действия. (Ограничения будут подробнее рассмотрены далее, а до тех пор не следует особенно беспокоиться об их назначении.)

```
GridBagLayout layout = new GridBagLayout();
panel.setLayout(layout);
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 100;
constraints.weighty = 100;
constraints.gridx = 0;
constraints.gridy = 2;
constraints.gridwidth = 2;
constraints.gridheight = 1;
panel.add(component, constraints);
```

Самое главное — правильно установить состояние объекта типа `GridBagConstraints`. Поэтому в следующих далее подразделах поясняется, как пользоваться этим объектом.

Параметры `gridx`, `gridy`, `gridwidth` и `gridheight`

Эти параметры определяют место компонента в сетке компоновки. В частности, параметры `gridx` и `gridy` задают столбец и ряд для местоположения левого верхнего угла компонента. А параметры `gridwidth` и `gridheight` определяют число рядов и столбцов, занимаемых данным компонентом.

Координаты сетки отсчитываются от нуля. В частности, выражения `gridx=0` и `gridy=0` обозначают левый верхний угол. Например, местоположение текстовой области в рассматриваемом здесь примере определяется параметрами `gridx=2` и `gridy=0`, поскольку она начинается со столбца под номером 2 (т.е. с третьего столбца). А если текстовая область занимает один столбец и четыре ряда, то `gridwidth=1`, а `gridheight=4`.

Весовые поля

Для каждой области сеточно-контейнерной компоновки следует задать так называемые *весовые поля*, определяемые параметрами `weightx` и `weighty`. Если вес равен нулю, то область всегда сохраняет свои первоначальные размеры. В примере, приведенном на рис. 9.29, для текстовых меток установлено нулевое значение параметра `weightx`. Это позволяет сохранять постоянную ширину меток при изменении размеров окна. С другой стороны, если задать нулевой вес всех областей, контейнер расположит компоненты в центре выделенной для него области, не заполнив до конца все ее пространство.

Затруднение, возникающее при установке параметров весовых полей, состоит в том, что они являются свойствами рядов и столбцов, а не отдельных ячеек. Но их нужно задавать для ячеек, поскольку диспетчер сеточно-контейнерной компоновки не различает отдельные ряды и столбцы. В качестве веса ряда или столбца принимается максимальный вес среди всех содержащихся в них ячеек. Следовательно, если требуется сохранить фиксированными размеры рядов и столбцов, необходимо установить нулевым вес всех компонентов в этих рядах и столбцах.

Однако вес на самом деле не позволяет определить относительные размеры столбцов. Он лишь указывает на ту часть “свободного” пространства, которая должна быть выделена для каждой области, если контейнер превышает рекомендуемые размеры. Чтобы научиться правильно подбирать вес, нужно иметь определенный опыт работы с рассматриваемым здесь диспетчером компоновки. Поэтому для начала рекомендуется поступать следующим образом. Установите вес равным **100**, затем запустите

программу на выполнение и посмотрите, как будет выглядеть пользовательский интерфейс. Для того чтобы выяснить, насколько правильно выравниваются ряды и столбцы, попробуйте изменить размеры окна. Если окажется, что какой-то ряд или столбец не должен увеличиваться, установите нулевой вес всех находящихся в нем компонентов. Можно, конечно, опробовать и другие весовые значения, но овчинка, как правило, выделки не стоит.

Параметры `fill` и `anchor`

Если требуется, чтобы компонент не растягивался и не заполнял все доступное пространство, на него следует наложить ограничение с помощью параметра `fill`. Этот параметр принимает четыре возможных значения: `GridBagConstraints.NONE`, `GridBagConstraints.HORIZONTAL`, `GridBagConstraints.VERTICAL` и `GridBagConstraints.BOTH`.

Если компонент не заполняет все доступное пространство, можно указать область, к которой его следует привязать, установив параметр `anchor`. Этот параметр может принимать следующие значения: `GridBagConstraints.CENTER` (по умолчанию), `GridBagConstraints.NORTH`, `GridBagConstraints.NORTHEAST`, `GridBagConstraints.EAST` и т.д.

Заполнение пустого пространства

Установив соответствующее значение в поле `insets` объекта типа `GridBagConstraints`, можно ввести дополнительное пустое пространство вокруг компонента. А если требуется задать конкретные пределы пустого пространства вокруг компонента, то следует установить соответствующие значения в полях `left`, `top`, `right` и `bottom` объекта типа `Insets`. Этот процесс называется *внешним заполнением*.

Значения, устанавливаемые в полях `ipadx` и `ipady`, определяют *внутреннее заполнение*. Эти значения добавляются к минимальным ширине и высоте компонента, гарантируя тем самым, что компонент не уменьшится до своих минимальных размеров.

Альтернативные способы установки параметров `gridx`, `gridy`, `gridwidth` и `gridheight`

В документации на библиотеку AWT рекомендуется не задавать абсолютные значения параметров `gridx` и `gridy`, а вместо этого использовать константу `GridBagConstraints.RELATIVE`. Затем компоненты нужно расположить, как обычно для сеточно-контейнерной компоновки: последовательными рядами слева направо.

Количество рядов и столбцов, занятых ячейкой, как правило, указывается в полях `gridheight` и `gridwidth`. Исключение из этого правила составляет компонент, занимающий последний ряд или столбец. Для него указывается не числовое значение, а специальная константа `GridBagConstraints.REMAINDER`. Этим диспетчер компоновки уведомляется, что данный компонент является последним в ряду.

Описанная схема компоновки выглядит вполне работоспособной. Но в то же время кажется довольно странным сначала скрывать от диспетчера компоновки сведения о фактическом расположении компонентов, а затем полагаться на то, что он сам угадает правильные параметры. У вас может сложиться впечатление, будто пользоваться диспетчером сеточно-контейнерной компоновки очень сложно. Но, следуя приведенным ниже рекомендациям, вы сможете научиться располагать компоненты ГПИ без особых усилий.

1. Набросайте схему расположения компонентов на бумаге.
2. Подберите такую сетку, чтобы мелкие компоненты уместались в отдельных ячейках, а крупные — в нескольких ячейках.
3. Пометьте ряды и столбцы сетки номерами 0, 1, 2, 3... . После этого вы можете определить значения параметров `gridx`, `gridy`, `gridwidth` и `gridheight`.
4. Выясните для каждого компонента, должен ли он заполнять ячейку по горизонтали или по вертикали? Если не должен, то как его выровнять внутри ячейки? Для этой цели служат параметры `fill` и `anchor`.
5. Установите вес равным 100. Но если требуется, чтобы отдельный ряд или столбец всегда сохранял свои первоначальные размеры, задайте нулевое значение параметров `weightx` или `weighty` всех компонентов, находящихся в данном ряду или столбце.
6. Напишите код. Тщательно проверьте ограничения, накладываемые в классе `GridBagConstraints`. Одно неверное ограничение может нарушить всю компоновку.
7. Скомпилируйте код, запустите его на выполнение и пользуйтесь им в свое удовольствие.

В состав некоторых построителей ГПИ входят инструментальные средства для визуального наложения ограничений. На рис. 9.31 в качестве примера показано диалоговое окно конфигурации в среде NetBeans.

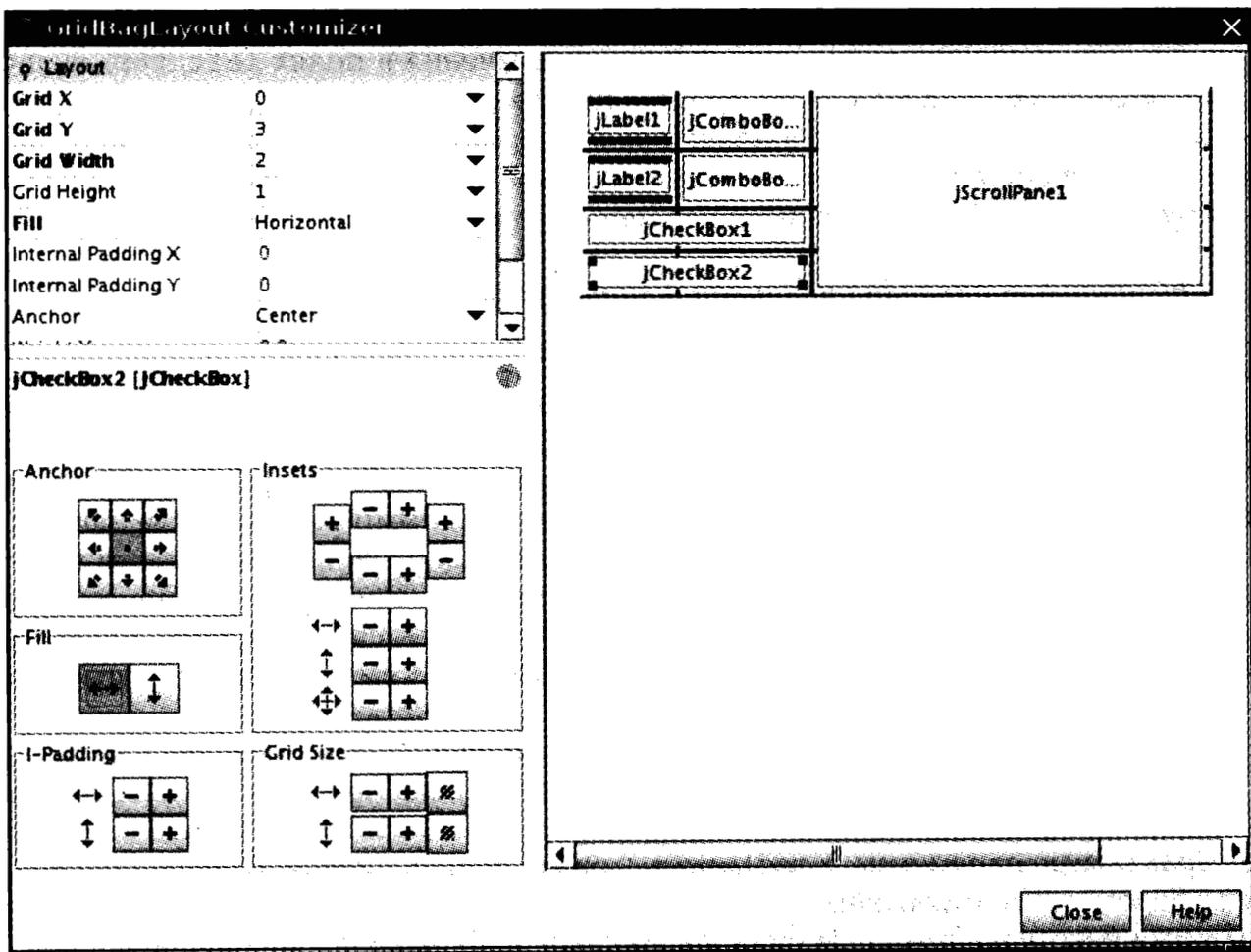


Рис. 9.31. Наложение ограничений в среде NetBeans

Вспомогательный класс на наложения ограничений при сеточно-контейнерной компоновке

Самая трудоемкая стадия сеточно-контейнерной компоновки относится к написанию кода, накладывающего ограничения. Многие программисты создают для этой цели вспомогательные функции или небольшие вспомогательные классы. Код одного из таких классов приведен после примера программы, где демонстрируется создание диалогового окна селектора шрифтов. Ниже перечислены характеристики, которыми должен обладать такой вспомогательный класс.

- Имеет имя GBC (прописные буквы из имени класса GridBagConstraints).
- Расширяет класс GridBagConstraints, поэтому константы можно указывать, используя более короткое имя, например GBC.EAST.
- Объект типа GBC используется при вводе компонента следующим образом:
`add(component, new GBC(1, 2));`
- Предусматривает два конструктора для установки наиболее употребительных параметров `gridx` и `gridy`, `gridx` и `gridy`, `gridwidth` и `gridheight`, как, например:
`add(component, new GBC(1, 2, 1, 4));`
- Предусматривает удобные методы для установки в полях пар значений координат `x` и `y`, как показано ниже.
`add(component, new GBC(1, 2).setWeight(100, 100));`
- Предусматривает методы для установки значений в полях и возврата ссылки `this`, чтобы объединять их в цепочки следующим образом:
`add(component, new GBC(1, 2).setAnchor(GBC.EAST).setWeight(100, 100));`
- Предусматривает метод `setInsets()` для построения объектов типа `Insets`. Так, если требуется создать пустое пространство размером в один пиксель, следует написать приведенную ниже строку кода.
`add(component, new GBC(1, 2).setAnchor(GBC.EAST).setInsets(1));`

В листинге 9.10 представлен исходный код класса для рассматриваемого здесь примера диалогового окна селектора шрифтов, а в листинге 9.11 — исходный код вспомогательного класса GBC. Ниже приведен фрагмент кода для ввода компонентов в сеточный контейнер.

```
add(faceLabel, new GBC(0,0).setAnchor(GBC.EAST));
add(face, new GBC(1,0).setFill(GBC.HORIZONTAL).setWeight(100,0).setInsets(1));
add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
add(size, new GBC(1,1).setFill(GBC.HORIZONTAL).setWeight(100,0).setInsets(1));
add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).setWeight(100, 100));
```

Уяснив принцип наложения ограничений при сеточно-контейнерной компоновке, разобраться в этом коде и отладить его будет несложно.



НА ЗАМЕТКУ! В руководстве, доступном по адресу <http://docs.oracle.com/javase/tutorial/uiswing/layout/gridbag.html>, предлагается использовать один и тот же объект типа `GridBagConstraints` для всех компонентов. На наш взгляд, получающийся в итоге код труден для восприятия, и при его написании легко допустить ошибку. Обратите

внимание на демонстрационный код из документа по адресу <http://docs.oracle.com/javase/tutorial/uiswing/events/containerlistener.html>. Действительно ли разработчик стремился растянуть кнопки по горизонтали, или он забыл снять ограничение `WOTH`, установленное в параметре `fill`?

Листинг 9.10. Исходный код из файла `gridbag/FontFrame.java`

```
1 package gridbag;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.beans.*;
6 import javax.swing.*;
7
8 /**
9  * Фрейм, в котором сеточно-контейнерная компоновка служит для
10 * расположения компонентов, предназначенных для выбора шрифтов
11 */
12 public class FontFrame extends JFrame
13 {
14     publicstatic final int TEXT_ROWS = 10;
15     public static final int TEXT_COLUMNS = 20;
16
17     private JComboBox<String> face;
18     private JComboBox<Integer> size;
19     private JCheckBox bold;
20     private JCheckBox italic;
21     private JTextArea sample;
22
23     public FontFrame()
24     {
25         GridBagLayout layout = new GridBagLayout();
26         setLayout(layout);
27
28         ActionListener listener =
29             EventHandler.create(ActionListener.class, this, "updateSample");
30
31         // сконструировать компоненты
32
33         JLabel faceLabel = new JLabel("Face: ");
34         face = new JComboBox<>(new String[] { "Serif", "SansSerif",
35             "Monospaced", "Dialog", "DialogInput" });
36
37         face.addActionListener(listener);
38
39         JLabel sizeLabel = new JLabel("Size: ");
40
41         size =
42             new JComboBox<>(new Integer[] { 8, 10, 12, 15, 18, 24, 36, 48 });
43
44         size.addActionListener(listener);
45
46         bold = new JCheckBox("Bold");
47         bold.addActionListener(listener);
48
49         italic = new JCheckBox("Italic");
50         italic.addActionListener(listener);
51
52         sample = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
```

```
53 sample.setText("The quick brown fox jumps over the lazy dog");
54 sample.setEditable(false);
55 sample.setLineWrap(true);
56 sample.setBorder(BorderFactory.createEtchedBorder());
57
58 // ввести компоненты в сетку, используя служебный класс GBC
59
60 add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
61 add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).
62     setWeight(100, 0).setInsets(1));
63 add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
64 add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).
65     setWeight(100, 0).setInsets(1));
66 add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).
67     setWeight(100, 100));
68 add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).
69     setWeight(100, 100));
70 add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).
71     setWeight(100, 100));
72 pack();
73 updateSample();
74 }
75
76 public void updateSample()
77 {
78     String fontFace = (String) face.getSelectedItem();
79     int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
80         + (italic.isSelected() ? Font.ITALIC : 0);
81     int fontSize = size.getItemAt(size.getSelectedIndex());
82     Font font = new Font(fontFace, fontStyle, fontSize);
83     sample.setFont(font);
84     sample.repaint();
85 }
86 }
```

Листинг 9.11. Исходный код из файла gridbag/GBC.java

```
1 package gridbag;
2
3 import java.awt.*;
4
5 /**
6  * Этот класс упрощает применение класса GridBagConstraints
7  * @version 1.01 2004-05-06
8  * @author Cay Horstmann
9  */
10 public class GBC extends GridBagConstraints
11 {
12     /**
13      * Строит объект типа GBC, накладывая ограничения с помощью
14      * параметров gridx и gridy, а все остальные ограничения
15      * накладываются на сеточно-контейнерную компоновку по умолчанию
16      * @param gridx Местоположение в сетке по горизонтали
17      * @param gridy Местоположение в сетке по вертикали
18      */
19     public GBC(int gridx, int gridy)
20     {
21         this.gridx = gridx;
22         this.gridy = gridy;
23     }
24 }
```

```
25  /**
26   * Строит объект типа GBC, накладывая ограничения с помощью параметров
27   * gridx, gridy, gridwidth, gridheight, а все остальные ограничения
28   * накладываются на сеточно-контейнерную компоновку по умолчанию
29   * @param gridx Местоположение в сетке по горизонтали
30   * @param gridy Местоположение в сетке по вертикали
31   * @param gridwidth Шаг сетки по горизонтали
32   * @param gridheight Шаг сетки по вертикали
33   */
34  public GBC(int gridx, int gridy, int gridwidth, int gridheight)
35  {
36      this.gridx = gridx;
37      this.gridy = gridy;
38      this.gridwidth = gridwidth;
39      this.gridheight = gridheight;
40  }
41
42  /**
43   * Устанавливает привязку к сетке
44   * @param anchor Степень привязки
45   * @return this Объект для последующего видоизменения
46   */
47  public GBC setAnchor(int anchor)
48  {
49      this.anchor = anchor;
50      return this;
51  }
52
53  /**
54   * Устанавливает направление для заполнения
55   * @param fill Направление заполнения
56   * @return this Объект для последующего видоизменения
57   */
58  public GBC setFill(int fill)
59  {
60      this.fill = fill;
61      return this;
62  }
63
64  /**
65   * Устанавливает веса ячеек
66   * @param weightx Вес ячейки по горизонтали
67   * @param weighty Вес ячейки по вертикали
68   * @return this Объект для последующего видоизменения
69   */
70  public GBC setWeight(double weightx, double weighty)
71  {
72      this.weightx = weightx;
73      this.weighty = weighty;
74      return this;
75  }
76
77  /**
78   * Вводит пробелы вокруг данной ячейки
79   * @param distance Пробел по всем направлениям
80   * @return this Объект для последующего видоизменения
81   */
82  public GBC setInsets(int distance)
83  {
84      this.insets = new Insets(distance, distance, distance, distance);
85      return this;
86  }
```

```

87
88  /**
89   * Вводит пробелы вокруг данной ячейки
90   * @param top Пробел сверху
91   * @param left Пробел слева
92   * @param bottom Пробел снизу
93   * @param right Пробел справа
94   * @return this Объект для последующего видоизменения
95   */
96  public GBC setInsets(int top, int left, int bottom, int right)
97  {
98      this.insets = new Insets(top, left, bottom, right);
99      return this;
100 }
101 /**
102  * Устанавливает внутреннее заполнение
103  * @param ipadx Внутреннее заполнение по горизонтали
104  * @param ipady Внутреннее заполнение по вертикали
105  * @return this Объект для последующего видоизменения
106  */
107  public GBC setIpad(int ipadx, int ipady)
108  {
109      this.ipadx = ipadx;
110      this.ipady = ipady;
111      return this;
112  }
113 }

```

java.awt.GridBagConstraints 1.0

- int gridx, gridy
Задают начальный столбец и ряд для расположения ячейки. По умолчанию принимаются нулевые значения.
- int gridwidth, gridheight
Задают количество столбцов и рядов, занимаемых ячейкой. По умолчанию принимают значение 1.
- double weightx, weighty
Определяют способность ячейки увеличиваться в размерах. По умолчанию принимают нулевое значение.
- int anchor
Задаёт вид выравнивания компонента внутри ячейки. Допускает указывать константы, определяющие абсолютное расположение.

NORTHWEST NORTH NORTHEAST

WEST CENTER EAST

SOUTHWEST SOUTH SOUTHEAST

или константы, определяющие расположение независимо от ориентации:

FIRST_LINE_START LINE_START FIRST_LINE_END

PAGE_START CENTER PAGE_END

LAST_LINE_START LINE_END LAST_LINE_END

Вторая группа констант оказывается удобной в том случае, если приложение локализуется на языки, где символы следуют справа налево или сверху вниз.

java.awt.GridBagConstraints 1.0

- int fill

Задает способ заполнения компонентом ячейки. Допускаются значения NONE, BOTH, HORIZONTAL и VERTICAL. По умолчанию принимается значение NONE.

- int ipadx, ipady

Задает внутреннее заполнение вокруг компонента. По умолчанию принимают нулевое значение.

- Insets insets

Задает внешнее заполнение вокруг границ ячейки. По умолчанию заполнение отсутствует.

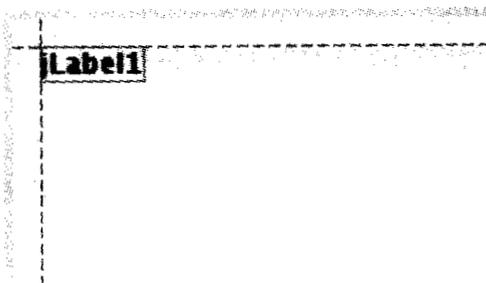
- GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady) **1.2**

Создает объект типа GridBagConstraints, заполняя все его поля указанными значениями. Этот конструктор следует использовать только в программах автоматического построения ГПИ, поскольку получаемый в итоге код труден для восприятия.

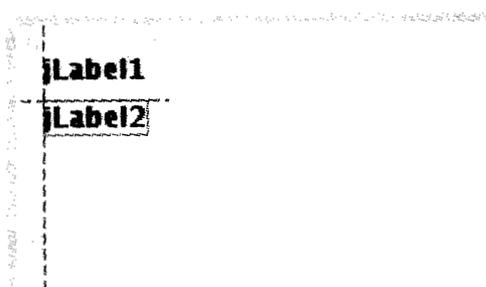
Диспетчер групповой компоновки

Прежде чем приступать к рассмотрению прикладного интерфейса API для класса GroupLayout диспетчера групповой компоновки, следует дать хотя бы краткое описание строителя ГПИ Swing GUI Builder (ранее называвшегося Matisse), входящего в состав среды NetBeans. А полностью ознакомиться с этим инструментальным средством можно по адресу <https://netbeans.org/features/java/swing.html>.

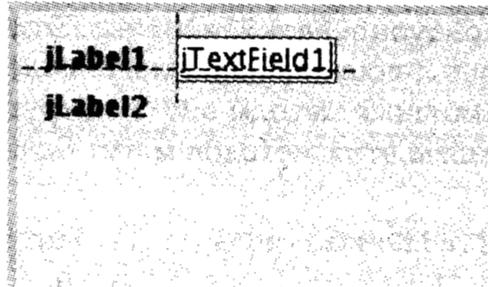
Итак, последовательность действий для компоновки верхней части диалогового окна, приведенного на рис. 9.13, следующая. Сначала создайте новый проект и введите в него новую форму типа JFrame. Перетащите метку в угол, чтобы появились две линии, отделяющие ее от границ контейнера, как показано ниже.



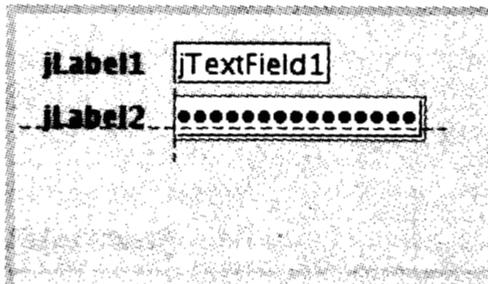
Расположите другую метку ниже первой в следующем ряду:



Перетащите текстовое поле таким образом, чтобы его базовая линия была выровнена по базовой линии первой метки, как показано ниже. Опять же обращайтесь внимание на направляющие.



И, наконец, выровняйте поле пароля по метке слева и расположенному выше полю.



Построитель ГПИ Swing GUI Builder интерпретирует все эти действия в следующий фрагмент кода Java:

```
layout.setHorizontalGroup(
    layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup()
            .addContainerGap()
            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                    .addComponent(jLabel1)
                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                    .addComponent(jTextField1))
                .addGroup(layout.createSequentialGroup()
                    .addComponent(jLabel2)
                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                    .addComponent(jPasswordField1)))
            .addContainerGap(222, Short.MAX_VALUE));
layout.setVerticalGroup(
    layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup()
            .addContainerGap()
            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                .addComponent(jLabel1)
                .addComponent(jTextField1))
            .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                .addComponent(jLabel2)
                .addComponent(jPasswordField1))
            .addContainerGap(244, Short.MAX_VALUE));
```

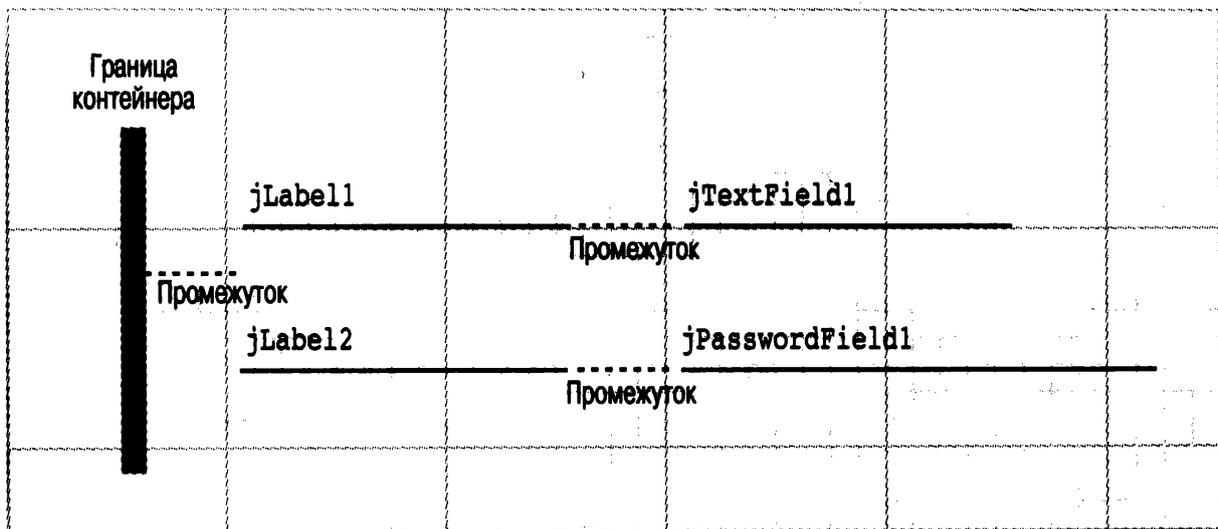
Этот код выглядит пугающе. Правда, писать его не придется. Тем не менее полезно иметь хотя бы элементарное представление о действиях компоновки, чтобы уметь

находить в них ошибки. Проанализируем основную структуру этого кода. Во врезках из документации на прикладной интерфейс API в конце этого раздела поясняется назначение каждого из методов и классов, применяемых в рассматриваемом здесь коде.

Компоненты организуются путем их размещения в объектах типа `GroupLayout`. `SequentialGroup` или `GroupLayout.ParallelGroup`. Эти классы являются производными от класса `GroupLayout.Group`. Группы могут содержать компоненты, пробелы и вложенные группы. Различные методы `add()` из групповых классов возвращают объект группы, поэтому вызовы этих методов могут быть соединены в цепочку следующим образом:

```
group.addComponent(...).addPreferredGap(...).addComponent(...);
```

Как следует из данного примера кода, компоновка групп разделяет вычисление расположения компонентов по горизонтали и по вертикали. Чтобы наглядно представить вычисление расположения компонентов по горизонтали, допустим, что компоненты сокращены до нулевой высоты, как показано ниже.



В данном случае имеются две параллельные последовательности компонентов, описываемые в следующем (немного упрощенном) коде:

```
.addContainerGap()
.addGroup(layout.createParallelGroup()
.addGroup(layout.createSequentialGroup()
.addComponent(jLabel1)
.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
.addComponent(jTextField1))
.addGroup(layout.createSequentialGroup()
.addComponent(jLabel2)
.addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
.addComponent(jPasswordField1)))
```

Но разве это правильно? Ведь если метки имеют разную длину, то текстовое поле и поле пароля не будут выровнены. Нужно как-то сообщить Swing GUI Builder, что эти поля требуется выровнять. Для этого выделите оба поля, щелкните правой кнопкой мыши и выберите команду `Align⇒Left to Column` (Выровнять⇒По левому краю столбца) из контекстного меню. Аналогичным образом выровняйте метки, выбрав на этот раз команду `Align⇒Right to Column` (Выровнять⇒По правому краю столбца), как показано на рис. 9.32.

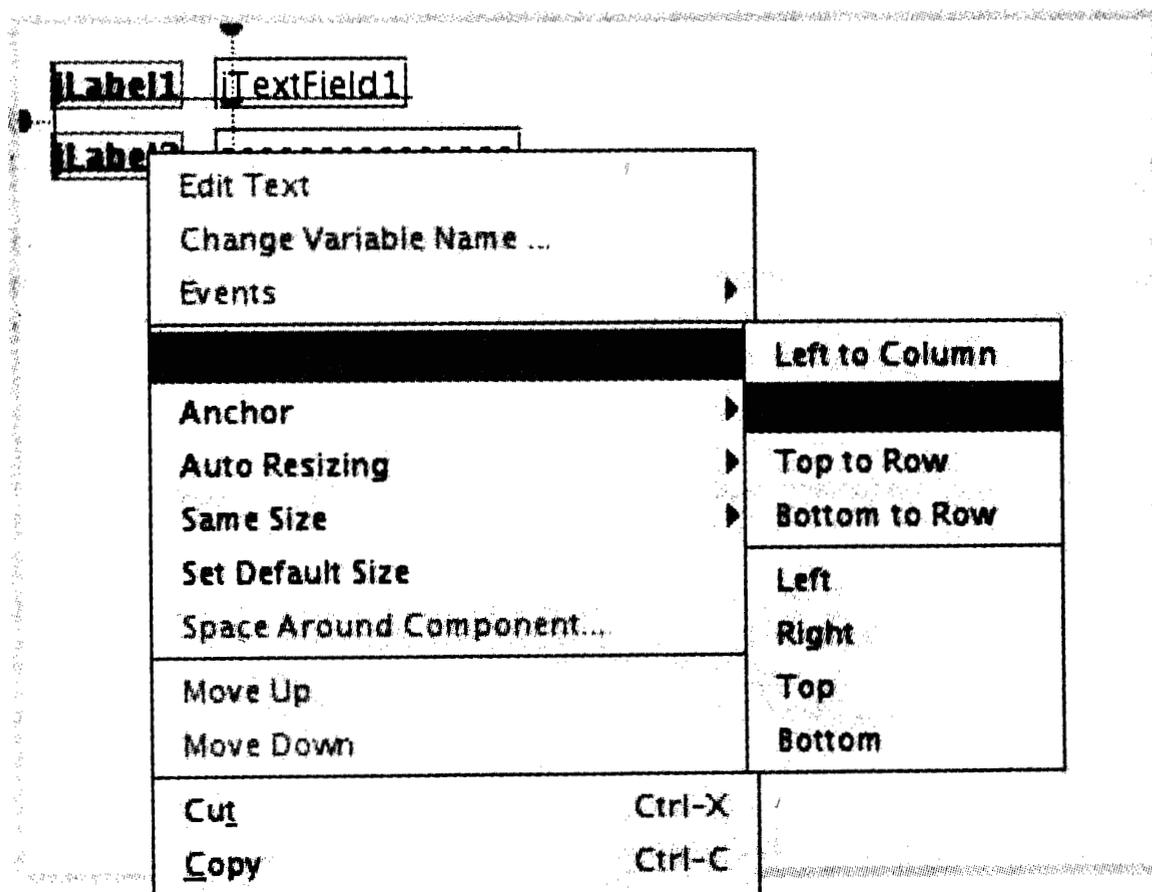
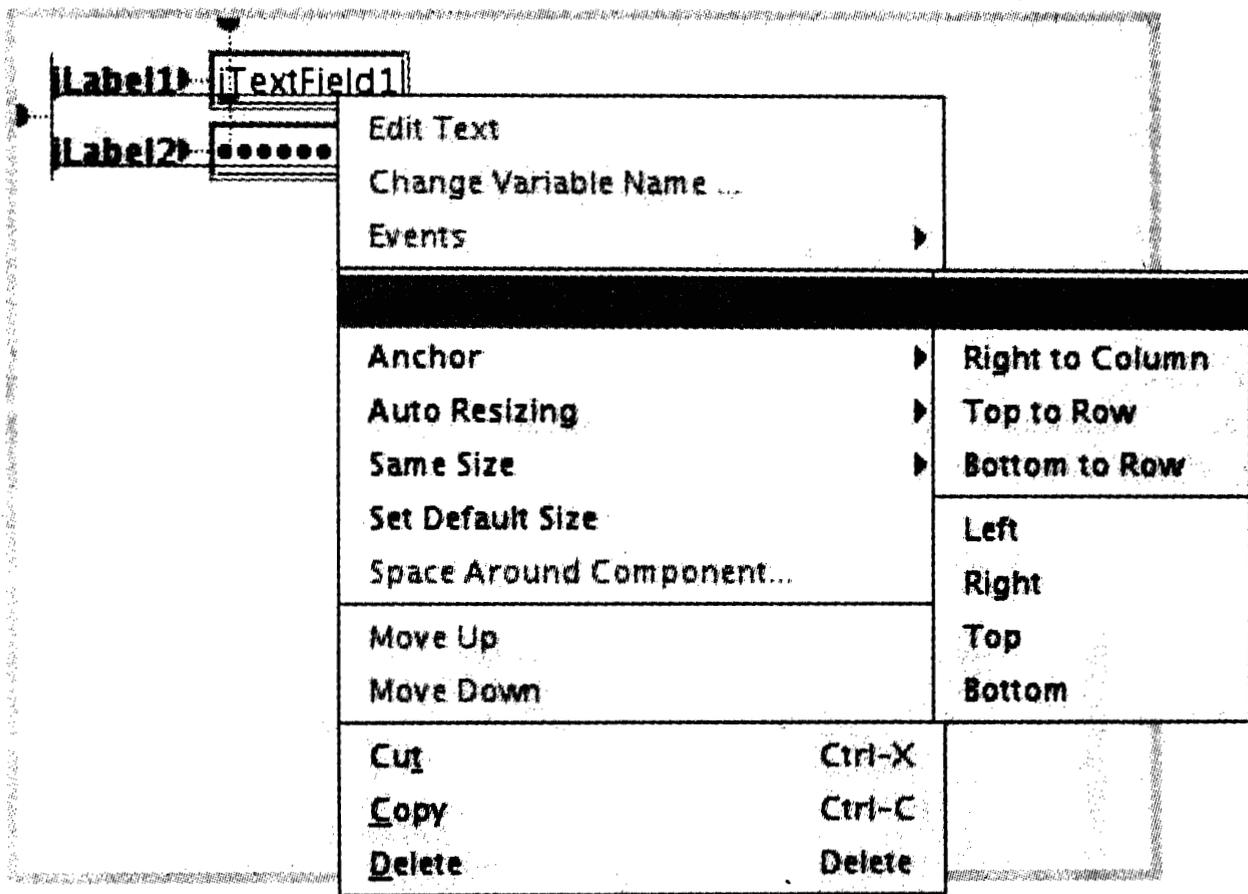
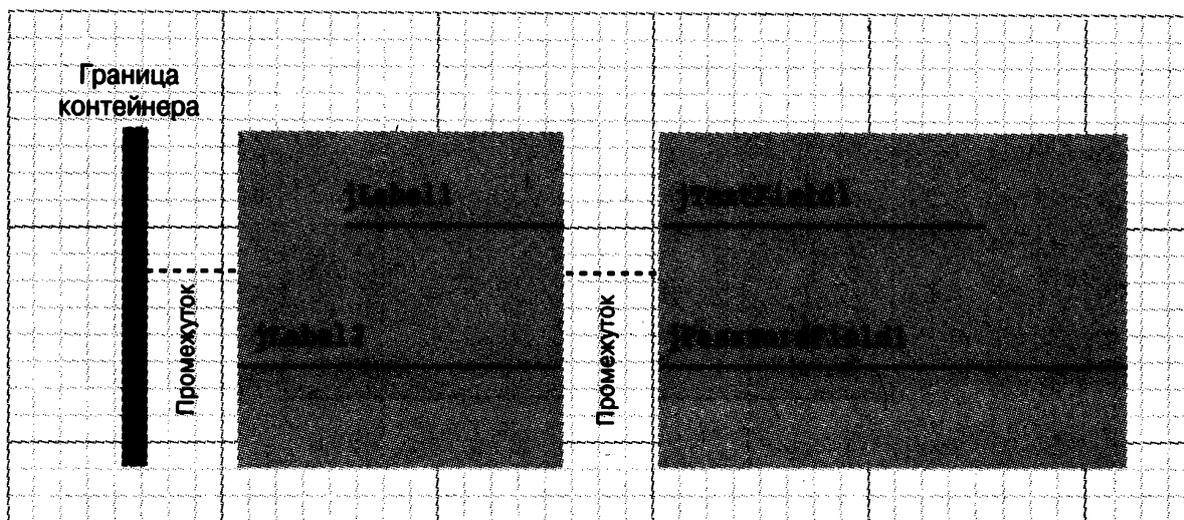


Рис. 9.32. Выравнивание меток и текстовых полей в Swing GUI Builder

В итоге генерируемый код компоновки изменится радикальным образом:

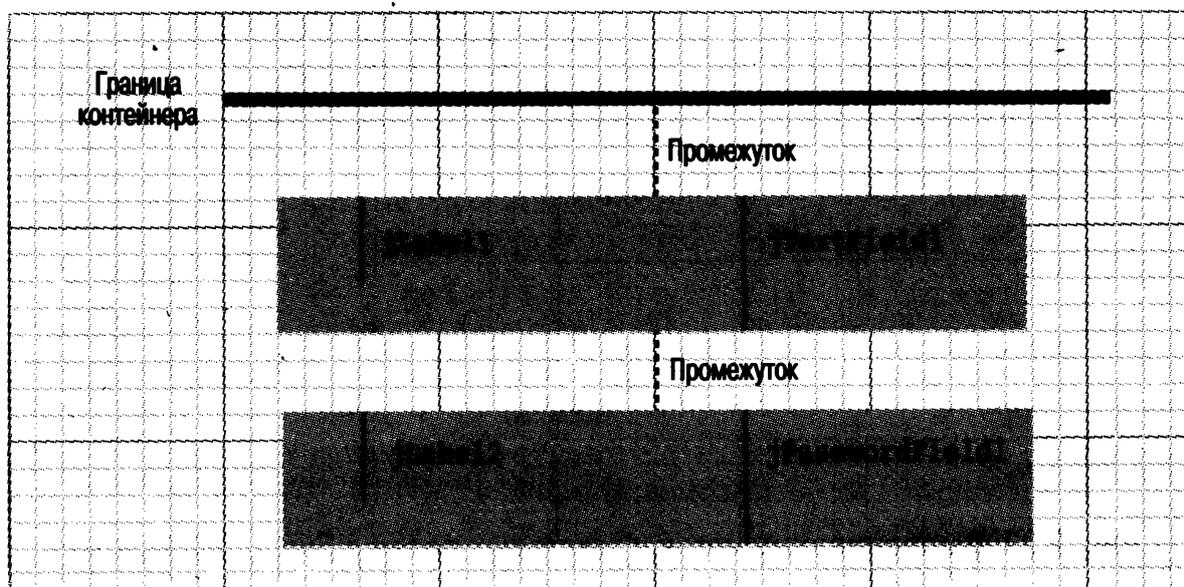
```
.addGroup(layout.createSequentialGroup()
    .addContainerGap()
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addComponent(jLabel1, GroupLayout.Alignment.TRAILING)
        .addComponent(jLabel2, GroupLayout.Alignment.TRAILING)
    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addComponent(jTextField1)
        .addComponent(jPasswordField1))
```

Теперь метки и поля расположены в двух параллельных группах. Первая группа имеет выравнивание TRAILING (т.е. выравнивание по правому краю при расположении текста слева направо), как показано ниже.



Способность Swing GUI Builder превращать инструкции разработчика ГПИ во вложенные группы сродни волшебству, но, как сказал известный писатель-фантаст Артур Кларк, любая достаточно развитая технология неотличима от волшебства.

Для полноты картины рассмотрим вычисление расположения компонентов по вертикали. Теперь компоненты нужно представить так, как будто у них нет ширины. Итак, имеется последовательная группа, состоящая из двух параллельных групп, разделенных пробельными промежутками, как показано ниже.



Соответствующий код компоновки выглядит следующим образом:

```
layout.createSequentialGroup()
    .addContainerGap()
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
        .addComponent(jLabel1)
        .addComponent(jTextField1))
    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
        .addComponent(jLabel2)
        .addComponent(jPasswordField1))
```

Как видите, в данном случае компоненты выравниваются по их базовым линиям. (Базовой называется линия, по которой выравнивается текст компонентов.) Одинаковые размеры ряда компонентов можно установить принудительно. Так, если требуется, чтобы текстовое поле и поле пароля точно совпадали по размеру, выделите оба поля в Swing GUI Builder, щелкните правой кнопкой мыши и выберите команду Same Size ⇒ Same Width (Одинаковый размер ⇒ Одинаковая ширина) из контекстного меню, как показано на рис. 9.33.

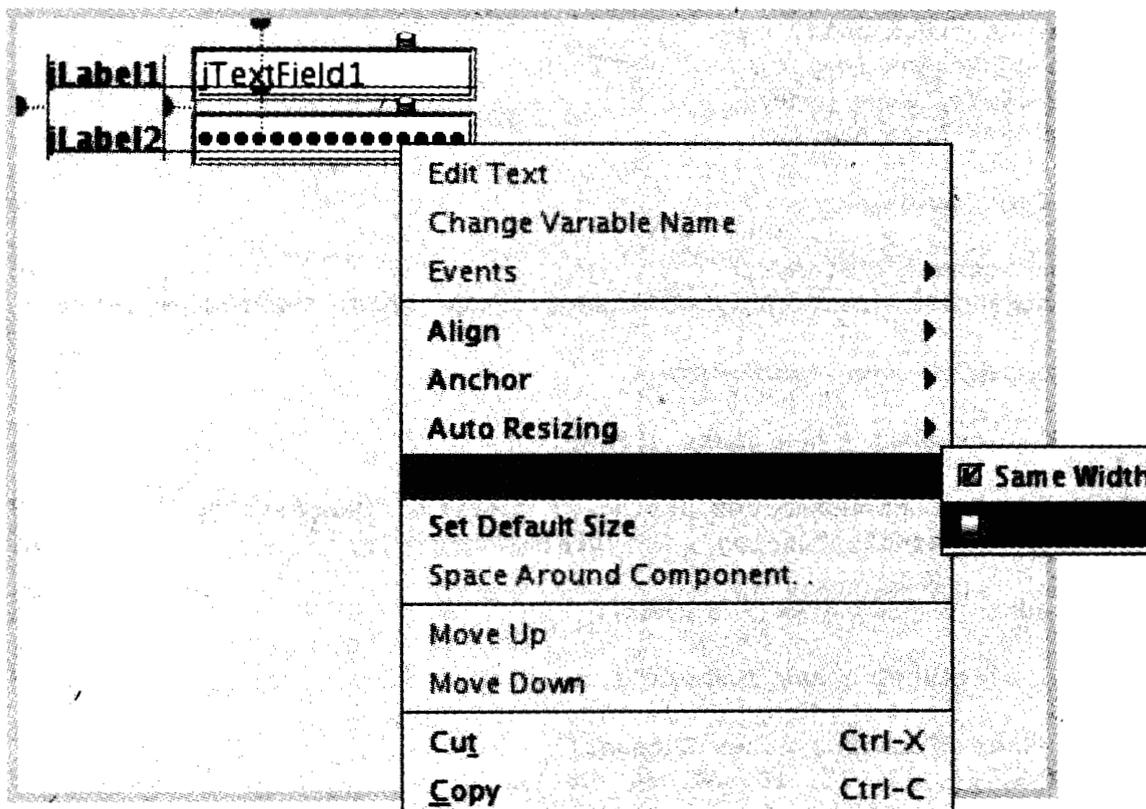


Рис. 9.33. Установка одинаковой ширины у двух компонентов

В итоге Swing GUI Builder добавляет в код компоновки следующий оператор:

```
layout.linkSize(SwingConstants.HORIZONTAL, new Component[]
    {jPasswordField1, jTextField1});
```

В примере кода из листинга 9.12 демонстрируется расположение компонентов селектора шрифтов из предыдущего примера с помощью диспетчера компоновки типа GroupLayout вместо диспетчера компоновки типа GridBagLayout. На первый взгляд этот код не проще, чем в листинге 9.10, но его не пришлось писать вручную. Компоновка была первоначально выполнена в Swing GUI Builder, а затем сгенерированный код был немного отредактирован.

Листинг 9.12. Исходный код из файла `groupLayout/FontFrame.java`

```
1 package groupLayout;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.beans.*;
6 import javax.swing.*;
7
8 /**
9  * Фрейм, в котором групповая компоновка служит для расположения
10  * компонентов, предназначенных для выбора шрифтов
11  */
12 public class FontFrame extends JFrame
13 {
14     public static final int TEXT_ROWS = 10;
15     public static final int TEXT_COLUMNS = 20;
16
17     private JComboBox<String> face;
18     private JComboBox<Integer> size;
19     private JCheckBox bold;
20     private JCheckBox italic;
21     private JScrollPane pane;
22     private JTextArea sample;
23
24     public FontFrame()
25     {
26         ActionListener listener =
27             EventHandler.create(ActionListener.class, this, "updateSample");
28
29         // сконструировать компоненты
30
31         JLabel faceLabel = new JLabel("Face: ");
32
33         face = new JComboBox<>(new String[]{ "Serif", "SansSerif",
34             "Monospaced", "Dialog", 32 "DialogInput" });
35
36         face.addActionListener(listener);
37
38         JLabel sizeLabel = new JLabel("Size: ");
39
40         size = new JComboBox<>(new Integer[]
41             { 8, 10, 12, 15, 18, 24, 36, 48 });
42
43         size.addActionListener(listener);
44
45         bold = new JCheckBox("Bold");
46         bold.addActionListener(listener);
47
48         italic = new JCheckBox("Italic");
49         italic.addActionListener(listener);
50
51         sample = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
52         sample.setText("The quick brown fox jumps over the lazy dog");
53         sample.setEditable(false);
54         sample.setLineWrap(true);
55         sample.setBorder(BorderFactory.createEtchedBorder());
```

```
56
57     pane = new JScrollPane(sample);
58
59     GroupLayout layout = new GroupLayout(getContentPane());
60     setLayout(layout);
61     layout.setHorizontalGroup(
62         layout.createParallelGroup(GroupLayout.Alignment.LEADING)
63             .addGroup(
64                 layout.createSequentialGroup().addContainerGap()
65                     .addGroup(
66                         layout.createParallelGroup(GroupLayout.Alignment.LEADING)
67                             .addGroup(GroupLayout.Alignment.TRAILING,
68                                 layout.createSequentialGroup())
69                             .addGroup(
70                                 layout.createParallelGroup(GroupLayout.Alignment.TRAILING)
71                                     .addComponent(faceLabel).addComponent(sizeLabel))
72                             .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
73                             .addGroup(
74                                 layout.createParallelGroup(
75                                     GroupLayout.Alignment.LEADING, false)
76                                     .addComponent(size).addComponent(face))
77                             .addComponent(italic).addComponent(bold)).addPreferredGap(
78                                 LayoutStyle.ComponentPlacement.RELATED).addComponent(pane)
79                             .addContainerGap()));
80
81     layout.linkSize(SwingConstants.HORIZONTAL,
82         new java.awt.Component[] { face, size });
83
84     layout.setVerticalGroup(
85         layout.createParallelGroup(GroupLayout.Alignment.LEADING)
86             .addGroup(
87                 layout.createSequentialGroup().addContainerGap().addGroup(
88                     layout.createParallelGroup(GroupLayout.Alignment.LEADING)
89                         .addComponent(
90                             pane, GroupLayout.Alignment.TRAILING).addGroup(
91                             layout.createSequentialGroup().addGroup(
92                                 layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
93                                     .addComponent(face).addComponent(faceLabel))
94                                     .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
95                                     .addGroup(
96                                         layout.createParallelGroup(
97                                             GroupLayout.Alignment.BASELINE).addComponent(size)
98                                             .addComponent(sizeLabel)).addPreferredGap(
99                                                 LayoutStyle.ComponentPlacement.RELATED).addComponent(
100                                 italic, GroupLayout.DEFAULT_SIZE,
101                                     GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
102                                 .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
103                                 .addComponent(bold, GroupLayout.DEFAULT_SIZE,
104                                     GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)))
105                             .addContainerGap()));
106     pack();
107 }
108
109 public void updateSample()
110 {
111     String fontFace = (String) face.getSelectedItemAt();
112     int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
```

```

113         + (italic.isSelected() ? Font.ITALIC : 0);
114     int fontSize = size.getItemAt(size.getSelectedIndex());
115     Font font = new Font(fontFace, fontStyle, fontSize);
116     sample.setFont(font);
117     sample.repaint();
118 }
119 }
120 }

```

javax.swing.GroupLayout 6

- `GroupLayout(Container host)`
Конструирует объект типа `GroupLayout` для компоновки компонентов в контейнере `host`. (Следует, однако, иметь в виду, что для объекта `host` все равно придется вызывать метод `setLayout()`.)
 - `void setHorizontalGroup(GroupLayout.Group g)`
 - `void setVerticalGroup(GroupLayout.Group g)`
Устанавливают группу, управляющую компоновкой по горизонтали или по вертикали.
 - `void linkSize(Component... components)`
 - `void linkSize(int axis, Component... component)`
Устанавливают одинаковые размеры указанных компонентов или одинаковый размер только по заданной оси (`SwingConstants.HORIZONTAL` по горизонтали или `SwingConstants.VERTICAL` по вертикали).
 - `GroupLayout.SequentialGroup createSequentialGroup()`
Создает группу, располагающую свои члены последовательно.
 - `GroupLayout.ParallelGroup createParallelGroup()`
 - `GroupLayout.ParallelGroup createParallelGroup(GroupLayout.Alignment align)`
 - `GroupLayout.ParallelGroup createParallelGroup(GroupLayout.Alignment align, boolean resizable)`
Создают группу, располагающую свои члены параллельно.
- | | | |
|-------------------|------------------|---|
| <i>Параметры:</i> | <i>align</i> | Одно из значений: <code>BASELINE</code> , <code>LEADING</code> (по умолчанию), <code>TRAILING</code> или <code>CENTER</code> |
| | <i>resizable</i> | Логическое значение <code>true</code> (по умолчанию), если размеры группы могут изменяться; логическое значение <code>false</code> , если предпочтительные размеры совпадают с минимальными и максимальными |
- `boolean getHonorsVisibility()`
 - `void setHonorsVisibility(boolean b)`
Получают или устанавливают свойство `honorsVisibility`. Когда оно принимает логическое значение `true` (по умолчанию), то невидимые компоненты не подлежат компоновке. А когда оно принимает логическое значение `false`, невидимые компоненты подлежат компоновке как видимые. Это удобно, когда некоторым компонентам требуется временно скрыть, не затрагивая компоновку.

- `boolean getAutoCreateGaps()`
- `void setAutoCreateGaps(boolean b)`
- `boolean getAutoCreateContainerGaps()`
- `void setAutoCreateContainerGaps(boolean b)`

Получают или устанавливают свойства `autoCreateGaps` и `autoCreateContainerGaps`. Когда они принимают логическое значение `true`, между компонентами или на границах контейнера автоматически вводятся отступы. По умолчанию они принимают логическое значение `false`. Устанавливать логическое значение `true` удобно в том случае, если объект типа `GroupLayout` создается вручную.

javax.swing.GroupLayout.Group

- `GroupLayout.Group addComponent(Component c)`
- `GroupLayout.Group addComponent(Component c, int minimumSize, int preferredSize, int maximumSize)`

Добавляют компонент в группу. Параметры, задающие размеры, могут принимать конкретные (неотрицательные) значения или же значения специальных констант `GroupLayout.DEFAULT_SIZE` или `GroupLayout.PREFERRED_SIZE`. Когда указывается константа `DEFAULT_SIZE`, для компонента вызываются методы `getMinimumSize()`, `getPreferredSize()` или `getMaximumSize()`. А если указывается константа `PREFERRED_SIZE`, то для компонента вызывается метод `getPreferredSize()`.

- `GroupLayout.Group addGap(int size)`
- `GroupLayout.Group addGap(int minimumSize, int preferredSize, int maximumSize)`

Вводят пробельный промежуток с жестко или гибко заданным размером.

- `GroupLayout.Group addGroup(GroupLayout.Group g)`

Вводит указанную группу в данную группу.

javax.swing.GroupLayout.ParallelGroup

- `GroupLayout.ParallelGroup addComponent(Component c, GroupLayout.Alignment align)`
- `GroupLayout.ParallelGroup addComponent(Component c, GroupLayout.Alignment align, int minimumSize, int preferredSize, int maximumSize)`
- `GroupLayout.ParallelGroup addGroup(GroupLayout.Group g, GroupLayout.Alignment align)`

Вводят компонент или группу в данную группу, используя заданное выравнивание (одно из значений `BASELINE`, `LEADING`, `TRAILING` или `CENTER`).

javax.swing.GroupLayout.SequentialGroup

- GroupLayout.SequentialGroup addContainerGap()
- GroupLayout.SequentialGroup addContainerGap(int preferredSize, int maximumSize)

Вводят пробельный промежуток, чтобы отделить компонент от края контейнера.

- GroupLayout.SequentialGroup addPreferredGap(LayoutStyle.ComponentPlacement type)

Вводит пробельный промежуток для разделения компонентов. В качестве параметра *type* указывается константа `LayoutStyle.ComponentPlacement.RELATED` или `LayoutStyle.ComponentPlacement.UNRELATED`.

Компоновка без диспетчера

Иногда требуется просто расположить какой-нибудь компонент в нужном месте, не прибегая к помощи диспетчера компоновки. Такой процесс называют *абсолютным расположением*. Если приложение задумывается как платформенно-независимое, такой подход не годится. Но если требуется быстро создать прототип ГПИ, то он вполне пригоден.

Для того чтобы расположить компонент в нужном месте, выполните следующие действия.

1. Укажите пустое значение `null` вместо конкретного диспетчера компоновки.
2. Введите компонент в контейнер.
3. Задайте координаты местоположения и размеры компонента следующим образом:

```
frame.setLayout(null);
JButton ok = new JButton("OK");
frame.add(ok);
ok.setBounds(10, 10, 30, 15);
```

java.awt.Component 1.0

- void setBounds(int x, int y, int width, int height)

Перемещает компонент и изменяет его размеры.

Параметры: *x, y* Координаты левого верхнего угла компонента
width, height Новые размеры компонента

Специальные диспетчеры компоновки

В принципе можно разработать собственный класс `LayoutManager`, управляющий расположением компонентов особым образом. В качестве любопытного примера покажем, как расположить все компоненты в контейнере по кругу (рис. 9.34).

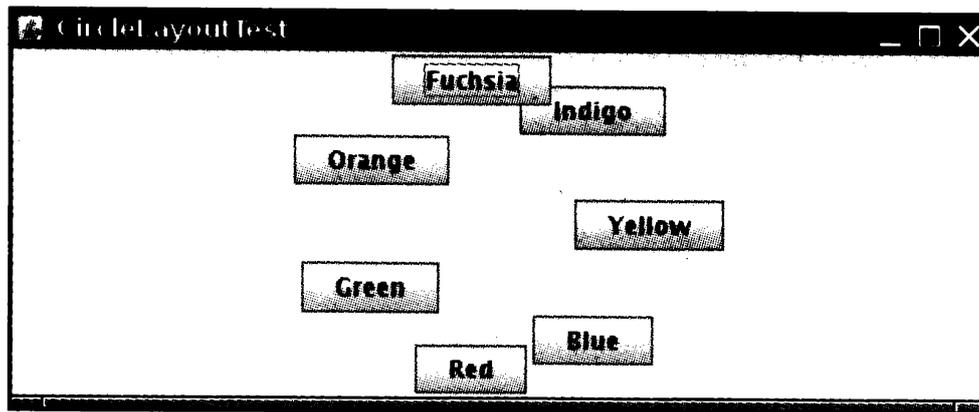


Рис. 9.34. Расположение компонентов по кругу

Класс специального диспетчера компоновки должен реализовывать интерфейс `LayoutManager`. Для этого придется переопределить следующие пять методов:

```
void addLayoutComponent(String s, Component c);  
void removeLayoutComponent(Component c);  
Dimension preferredLayoutSize(Container parent);  
Dimension minimumLayoutSize(Container parent);  
void layoutContainer(Container parent);
```

Первые два метода вызываются при добавлении или удалении компонента. Если дополнительные сведения о компоненте отсутствуют, тело этих методов можно оставить пустым. Следующие два метода вычисляют объем пространства, требующегося при минимальных и рекомендуемых размерах компонентов. Обычно эти величины равны. А пятый метод вызывает метод `setBounds()` для всех компонентов и выполняет основные операции по расположению компонента.



НА ЗАМЕТКУ! В библиотеке AWT имеется второй интерфейс под названием `LayoutManager2`. Он содержит десять, а не пять методов. Главная особенность этого интерфейса состоит в том, что с его помощью можно применять метод `add()` без всяких ограничений. Например, классы `BorderLayout` и `GridBagLayout` реализуют именно интерфейс `LayoutManager2`.

В листинге 9.13 приведен исходный код программы, реализующей специальный диспетчер компоновки `CircleLayout`, который располагает компоненты по кругу внутри контейнера. И хотя это довольно любопытная компоновка, тем не менее она совершенно бесполезна. А в листинге 9.14 приведен исходный код класса фрейма, создаваемого в данном примере программы.

Листинг 9.13. Исходный код из файла `circleLayout/CircleLayout.java`

```
1 package circleLayout;  
2  
3 import java.awt.*;  
4  
5 /**  
6  * Диспетчер компоновки, располагающий компоненты по кругу  
7  */  
8 public class CircleLayout implements LayoutManager  
9 {  
10     private int minWidth = 0;  
11     private int minHeight = 0;
```

```
12 private int preferredWidth = 0;
13 private int preferredHeight = 0;
14 private boolean sizesSet = false;
15 private int maxComponentWidth = 0;
16 private int maxComponentHeight = 0;
17
18 public void addLayoutComponent(String name, Component comp)
19 {
20 }
21 public void removeLayoutComponent(Component comp)
22 {
23 }
24
25 public void setSizes(Container parent)
26 {
27     if (sizesSet) return;
28     int n = parent.getComponentCount();
29
30     preferredWidth = 0;
31     preferredHeight = 0;
32     minWidth = 0;
33     minHeight = 0;
34     maxComponentWidth = 0;
35     maxComponentHeight = 0;
36
37     // вычислить максимальную ширину и высоту компонентов и установить
38     // предпочтительные размеры по сумме размеров компонентов
39
40     for (int i = 0; i < n; i++)
41     {
42         Component c = parent.getComponent(i);
43         if (c.isVisible())
44         {
45             Dimension d = c.getPreferredSize();
46             maxComponentWidth = Math.max(maxComponentWidth, d.width);
47             maxComponentHeight = Math.max(maxComponentHeight, d.height);
48             preferredWidth += d.width;
49             preferredHeight += d.height;
50         }
51     }
52     minWidth = preferredWidth / 2;
53     minHeight = preferredHeight / 2;
54     sizesSet = true;
55 }
56
57 public Dimension preferredLayoutSize(Container parent)
58 {
59     setSizes(parent);
60     Insets insets = parent.getInsets();
61     int width = preferredWidth + insets.left + insets.right;
62     int height = preferredHeight + insets.top + insets.bottom;
63     return new Dimension(width, height);
64 }
65 public Dimension minimumLayoutSize(Container parent)
66 {
67     setSizes(parent);
68     Insets insets = parent.getInsets();
```

```
69     int width = minWidth + insets.left + insets.right;
70     int height = minHeight + insets.top + insets.bottom;
71     return new Dimension(width, height);
72 }
73
74 public void layoutContainer(Container parent)
75 {
76     setSizes(parent);
77
78     // вычислить центр круга
79
80     Insets insets = parent.getInsets();
81     int containerWidth =
82         parent.getSize().width - insets.left - insets.right;
83     int containerHeight =
84         parent.getSize().height - insets.top - insets.bottom;
85
86     int xcenter = insets.left + containerWidth / 2;
87     int ycenter = insets.top + containerHeight / 2;
88
89     // вычислить радиус окружности
90
91     int xradius = (containerWidth - maxComponentWidth) / 2;
92     int yradius = (containerHeight - maxComponentHeight) / 2;
93     int radius = Math.min(xradius, yradius);
94
95     // расположить компоненты по кругу
96
97     int n = parent.getComponentCount();
98     for (int i = 0; i < n; i++)
99     {
100         Component c = parent.getComponent(i);
101         if (c.isVisible())
102         {
103             double angle = 2 * Math.PI * i / n;
104
105             // центральная точка компонента
106             int x = xcenter + (int) (Math.cos(angle) * radius);
107             int y = ycenter + (int) (Math.sin(angle) * radius);
108
109             // переместить компонент, расположив его в центральной точке
110             // с координатами (x, y) и предпочтительными размерами
111             Dimension d = c.getPreferredSize();
112             c.setBounds(x - d.width / 2, y - d.height / 2, d.width, d.height);
113         }
114     }
115 }
```

Листинг 9.14. Исходный код из файла circleLayout/CircleLayoutFrame.java

```
1 package circleLayout;
2
3 import javax.swing.*;
4
5 /**
```

```
6  * Фрейм, в котором демонстрируется расположение кнопок по кругу
7  */
8  public class CircleLayoutFrame extends JFrame
9  {
10     public CircleLayoutFrame()
11     {
12         setLayout(new CircleLayout());
13         add(new JButton("Yellow"));
14         add(new JButton("Blue"));
15         add(new JButton("Red"));
16         add(new JButton("Green"));
17         add(new JButton("Orange"));
18         add(new JButton("Fuchsia"));
19         add(new JButton("Indigo"));
20         pack();
21     }
22 }
```

java.awt.LayoutManager 1.0

- void addLayoutComponent(String name, Component comp)
Вводит компонент в текущую компоновку.
Параметры: name Идентификатор, обозначающий расположение компонента
 comp Вводимый компонент
- void removeLayoutComponent(Component comp)
Удаляет компонент из текущей компоновки.
- Dimension preferredLayoutSize(Container cont)
Возвращает рекомендуемые размеры контейнера, в котором выполняется текущая компоновка.
- void layoutContainer(Container cont)
Располагает компоненты в контейнере.

Порядок обхода компонентов

Если в окне содержится много компонентов, следует подумать о *порядке их обхода*. Когда окно открывается в первый раз, фокус ввода находится на том компоненте, который считается первым при заданном порядке обхода. Каждый раз, когда пользователь нажимает клавишу <Tab>, фокус ввода перемещается к следующему компоненту. (Напомним, что компонентами, обладающими фокусом ввода, можно манипулировать с клавиатуры. Например, кнопку, обладающую фокусом ввода, можно выбрать, нажав клавишу пробела.) Вам лично такая возможность может быть неинтересной, но имеется немало пользователей, которые предпочитают работать именно так. Среди них есть “ненавистники” мыши, а также те, кто просто не в состоянии ею пользоваться, подавая голосом команды для перемещения по элементам ГПИ. Именно по этим причинам вам стоит знать, каким образом в Swing интерпретируется порядок обхода компонентов ГПИ.

Порядок обхода осуществляется очень просто: сначала слева направо, а затем сверху вниз. Например, в диалоговом окне селектора шрифтов обход компонентов осуществляется в следующем порядке (рис. 9.35).

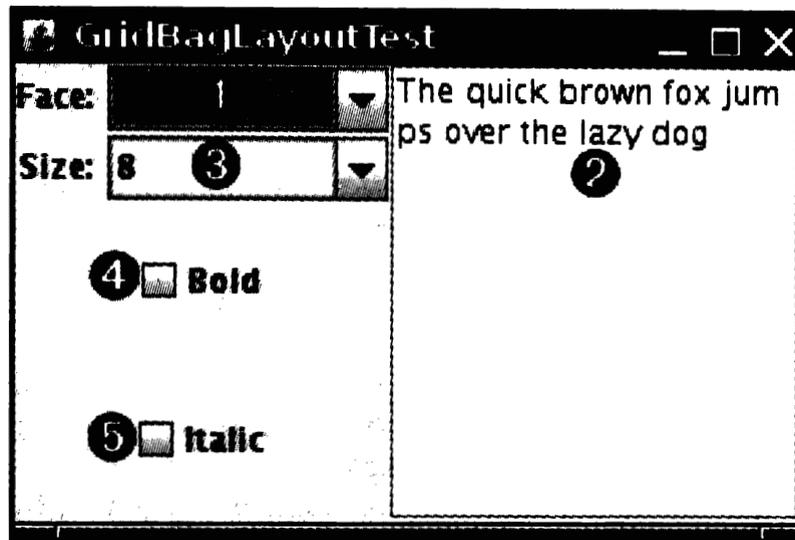


Рис. 9.35. Геометрический порядок обхода компонентов

- ❶ Комбинированный список для выбора начертания шрифта.
- ❷ Текстовая область с образцом текста. (Находясь в этой области, следует нажать комбинацию клавиш <Ctrl+Tab> для перехода к следующему полю, потому что символ табуляции интерпретируется как часть вводимого текста.)
- ❸ Комбинированный список для выбора размера шрифта.
- ❹ Флажок для установки полужирного начертания шрифта.
- ❺ Флажок для установки наклонного начертания шрифта.

Если контейнеры вложены друг в друга, то дело обстоит сложнее. Когда фокус ввода передается вложенному контейнеру, его автоматически получает левый верхний компонент, а затем начинается обход остальных компонентов. В конечном итоге фокус ввода передается компоненту, следующему за контейнером. Это обстоятельство можно с выгодой использовать, помещая группу связанных между собой элементов в другой контейнер, например, на панель.



НА ЗАМЕТКУ! Сделав вызов `component.setFocusable(false)`, можно исключить компонент из обхода с фокусом ввода. Это удобно сделать для рисованных компонентов, не принимающих данные, вводимые с клавиатуры.

Диалоговые окна

До сих пор рассматривались только компоненты пользовательского интерфейса, которые находились в окне фрейма, создаваемого в приложении, что характерно в основном для *апплетов*, выполняемых в окне браузера. Но при разработке приложений возникает потребность в отдельных всплывающих диалоговых окнах, которые должны появляться на экране и обеспечивать обмен данными с пользователем.

Как и в большинстве оконных систем, в AWT различаются *режимные (модальные)* и *безрежимные (немодальные)* диалоговые окна. Режимные диалоговые окна не дают пользователю возможности одновременно работать с другими окнами приложения. Такие окна требуются в том случае, если пользователь должен ввести данные, от которых зависит дальнейшая работа приложения. Например, при вводе файла пользователь должен сначала указать его имя. И только после того, как пользователь закроет режимное диалоговое окно, приложение начнет выполнять операцию ввода файла.

Безрежимное диалоговое окно дает пользователю возможность одновременно вводить данные как в этом окне, так и в других окнах приложения. Примером такого окна служит панель инструментов. Она постоянно находится на своем месте, и пользователь может одновременно взаимодействовать как с ней, так и с другими окнами.

В начале этого раздела будет представлено простейшее режимное диалоговое окно, содержащее единственную строку сообщения. В Swing имеется удобный класс `JOptionPane`, позволяющий выводить на экран режимное диалоговое окно, не прибегая к написанию специального кода для его поддержки. Далее будет показано, как реализовать свое собственное диалоговое окно. И в конце раздела поясняется, каким образом данные передаются из приложения в диалоговое окно и обратно.

Завершается раздел анализом двух стандартных диалоговых окон для обращения с файлами и выбора цвета. Диалоговое окно для работы с файлами устроено достаточно сложно. Для обращения с ним нужно хорошо знать функциональные возможности класса `JFileChooser` из библиотеки Swing. А для выбора цвета из палитры может пригодиться диалоговое окно, создаваемое средствами класса `JColorChooser`.

Диалоговые окна для выбора разных вариантов

В состав библиотеки Swing входит много готовых простых диалоговых окон, которые позволяют вводить данных отдельными фрагментами. Для этой цели в классе `JOptionPane` имеются перечисленные ниже статические методы.

<code>ShowMessageDialog()</code>	Выводит на экран сообщение и ожидает до тех пор, пока пользователь не щелкнет на кнопке OK
<code>ShowConfirmDialog()</code>	Выводит на экран сообщение и ожидает от пользователя подтверждения (щелчок на кнопке OK или Cancel)
<code>ShowOptionDialog()</code>	Выводит на экран сообщение и предоставляет пользователю возможность выбора среди нескольких вариантов
<code>showInputDialog()</code>	Выводит на экран сообщение и поле, в котором пользователь должен ввести данные

На рис. 9.36 показано типичное диалоговое окно для выбора разных вариантов. Как видите, в нем содержатся следующие компоненты:

- пиктограмма;
- сообщение;
- одна или несколько кнопок.

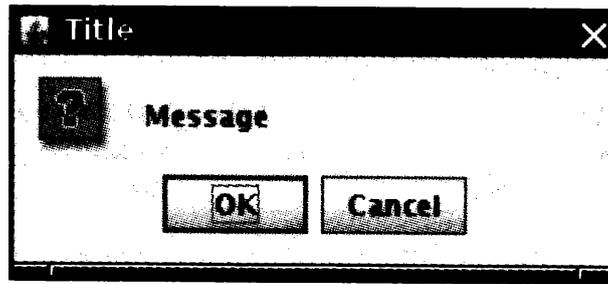


Рис. 9.36. Типичное диалоговое окно для выбора разных вариантов

Диалоговое окно может содержать дополнительный компонент для ввода данных. Этим компонентом может быть текстовое поле, в котором пользователь вводит произвольную символьную строку, или комбинированный список, один из элементов которого пользователь должен выбрать. Компоновка подобных диалоговых окон и выбор пиктограмм для стандартных сообщений зависит от визуального стиля ГПИ.

Пиктограмма в левой части диалогового окна выбирается в зависимости от типа сообщения. Существуют пять типов сообщений:

```
ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE
```

Для сообщения типа `PLAIN_MESSAGE` пиктограмма не предусмотрена. А для каждого типа диалоговых окон существует метод, позволяющий указывать свою собственную пиктограмму.

С каждым типом диалоговых окон можно связать определенное сообщение, которое может быть представлено символьной строкой, пиктограммой, компонентом пользовательского интерфейса или любым другим объектом. Ниже поясняется, каким образом отображаются эти сообщения.

<code>String</code>	Выводит символьную строку
<code>Icon</code>	Отображает пиктограмму
<code>Component</code>	Отображает компонент
<code>Object[]</code>	Выводит все объекты из массива, отображая их один над другим
Любой другой объект	Вызывает метод <code>toString()</code> и выводит получаемую в итоге символьную строку

Все эти возможности демонстрируются далее в примере программы из листинга 9.15. Разумеется, на экран чаще всего выводится символьная строка сообщения. В то же время возможность отображать в диалоговом окне объекты типа `Component` дает немало удобств, поскольку, вызвав метод `paintComponent()`, можно нарисовать все, что угодно.

Внешний вид кнопок, расположенных в нижней части диалогового окна, зависит от его типа, а также от *типа вариантов*. Вызывая метод `showMessageDialog()` или `showInputDialog()`, вы ограничиваетесь только стандартным набором кнопок (OK или OK и Cancel). А вызывая метод `showConfirmDialog()`, можно выбрать один из четырех типов вариантов:

```
DEFAULT_OPTION
YES_NO_OPTION
```

```
YES_NO_CANCEL_OPTION
OK_CANCEL_OPTION
```

С помощью метода `showOptionDialog()` можно указать произвольный набор вариантов, задав массив объектов, соответствующих каждому из них. Элементы этого массива отображаются на экране описанным ниже образом.

<code>String</code>	Создает кнопку, меткой которой служит указанная символьная строка
<code>Icon</code>	Создает кнопку, меткой которой служит указанная пиктограмма
<code>Component</code>	Отображает компонент
Любой другой объект	Вызывает метод <code>toString()</code> и создает кнопку, меткой которой служит получаемая в итоге символьная строка

Статические методы, предназначенные для создания диалоговых окон, возвращают перечисленные ниже значения.

<code>ShowMessageDialog()</code>	Возвращаемое значение отсутствует
<code>showConfirmDialog()</code>	Целое значение, соответствующее выбранному варианту
<code>showOptionDialog()</code>	Целое значение, соответствующее выбранному варианту
<code>showInputDialog()</code>	Символьная строка, введенная или выбранная пользователем

Методы `showConfirmDialog()` и `showOptionDialog()` возвращают целое значение, обозначающее кнопку, на которой щелкнул пользователь. В диалоговом окне для выбора разных вариантов это числовое значение является порядковым номером. Если вместо выбора варианта пользователь закрыл диалоговое окно, возвращается константа `CLOSED_OPTION`. Ниже приведены константы, используемые в качестве возвращаемых значений.

```
OK_OPTION
CANCEL_OPTION
YES_OPTION
NO_OPTION
CLOSED_OPTION
```

Несмотря на обилие упомянутых выше мнемонических обозначений разных вариантов выбора, создать диалоговое окно данного типа совсем не трудно. Для этого выполните следующие действия.

1. Выберите тип диалогового окна (для вывода сообщения, получения подтверждения, выбора разных вариантов или ввода данных).
2. Выберите пиктограмму (с ошибкой, важной информацией, предупреждением, вопросом, свою собственную) или вообще откажитесь от нее.
3. Выберите сообщение (в виде символьной строки, пиктограммы, пользовательского компонента или массива компонентов).
4. Если вы выбрали диалоговое окно для подтверждения выбора, задайте тип вариантов (по умолчанию Yes/No, No/Cancel или OK/Cancel).
5. Если вы создаете диалоговое окно для выбора разных вариантов, задайте варианты выбора (в виде символьных строк, пиктограмм или собственных компонентов), а также вариант, выбираемый по умолчанию.

6. Если вы создаете диалоговое окно для ввода данных, выберите текстовое поле или комбинированный список.
7. Найдите подходящий метод в классе `JOptionPane`.

Допустим, требуется отобразить на экране диалоговое окно, показанное на рис. 9.36. В этом окне выводится сообщение, а пользователю предлагается подтвердить его или отклонить. Следовательно, это диалоговое окно для подтверждения выбора. Пиктограмма, отображаемая в этом окне, относится к разряду вопросов, а сообщение выводится символьной строкой. Тип вариантов выбора обозначается константой `OK_CANCEL_OPTION`. Для создания такого диалогового окна служит следующий фрагмент кода:

```
int selection = JOptionPane.showConfirmDialog(parent,
    "Message", "Title",
    JOptionPane.OK_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE);
if (selection == JOptionPane.OK_OPTION) . . .
```



СОВЕТ. Символьная строка сообщения может содержать символ перевода строки (`'\n'`). В этом случае сообщение выводится в нескольких строках.

В примере программы, исходный код которой вместе с классом фрейма приведен в листинге 9.15, на экран выводится окно с шестью панелями кнопок (рис. 9.37). А в листинге 9.16 приведен исходный код класса для этих панелей. Если щелкнуть на кнопке Show (Показать), появится выбранный тип диалогового окна.

Листинг 9.15. Исходный код из файла `optionDialog/OptionDialogFrame.java`

```
1 package optionDialog;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.geom.*;
6 import java.util.*;
7 import javax.swing.*;
8
9 /**
10  * Фрейм с установками для выбора различных типов диалоговых окон
11  */
12 public class OptionDialogFrame extends JFrame
13 {
14     private ButtonPanel typePanel;
15     private ButtonPanel messagePanel;
16     private ButtonPanel messageTypePanel;
17     private ButtonPanel optionTypePanel;
18     private ButtonPanel optionsPanel;
19     private ButtonPanel inputPanel;
20     private String messageString = "Message";
21     private Icon messageIcon = new ImageIcon("blue-ball.gif");
22     private Object messageObject = new Date();
23     private Component messageComponent = new SampleComponent();
24
25     public OptionDialogFrame()
26     {
```

```

27     JPanel gridPanel = new JPanel();
28     gridPanel.setLayout(new GridLayout(2, 3));
29
30     typePanel =
31         new ButtonPanel("Type", "Message", "Confirm", "Option", "Input");
32     messageTypePanel = new ButtonPanel("Message Type", "ERROR_MESSAGE",
33         "INFORMATION_MESSAGE", "WARNING_MESSAGE", "QUESTION_MESSAGE",
34         "PLAIN_MESSAGE");
35     messagePanel = new ButtonPanel("Message", "String", "Icon",
36         "Component", "Other", "Object[]");
37     optionTypePanel = new ButtonPanel("Confirm", "DEFAULT_OPTION",
38         "YES_NO_OPTION", 35 "YES_NO_CANCEL_OPTION", "OK_CANCEL_OPTION");
39     optionsPanel = new ButtonPanel("Option", "String[]",
40         "Icon[]", "Object[]");
41     inputPanel = new ButtonPanel("Input", "Text field", "Combo box");
42
43     gridPanel.add(typePanel);
44     gridPanel.add(messageTypePanel);
45     gridPanel.add(messagePanel);
46     gridPanel.add(optionTypePanel);
47     gridPanel.add(optionsPanel);
48     gridPanel.add(inputPanel);
49
50     // ввести панель с кнопкой Show
51
52     JPanel showPanel = new JPanel();
53     JButton showButton = new JButton("Show");
54     showButton.addActionListener(new ShowAction());
55     showPanel.add(showButton);
56
57     add(gridPanel, BorderLayout.CENTER);
58     add(showPanel, BorderLayout.SOUTH);
59     pack();
60 }
61
62 /**
63  * Получает выбранное в настоящий момент сообщение
64  * @return Возвращается строка, пиктограмма, компонент или массив
65  *         объектов в зависимости от выбора на панели сообщений
66  */
67 public Object getMessage()
68 {
69     String s = messagePanel.getSelection();
70     if (s.equals("String")) return messageString;
71     else if (s.equals("Icon")) return messageIcon;
72     else if (s.equals("Component")) return messageComponent;
73     else if (s.equals("Object[]")) return new Object[]
74         { messageString, messageIcon, 69 messageComponent, messageObject };
75     else if (s.equals("Other")) return messageObject;
76     else return null;
77 }
78
79 /**
80  * Получает выбранные в настоящий момент варианты
81  * @return Возвращается строка, пиктограмма, компонент или массив
82  *         объектов в зависимости от выбора на панели сообщений
83  */
84 public Object[] getOptions()

```

```
85  {
86    String s = optionsPanel.getSelection();
87    if (s.equals("String[]")) return new String[]
88        { "Yellow", "Blue", "Red" };
89    else if (s.equals("Icon[]")) return new Icon[]
90        { new ImageIcon("yellow-ball.gif"),
91          new ImageIcon("blue-ball.gif"), new ImageIcon("red-ball.gif") };
92    else if (s.equals("Object[]")) return new Object[]
93        { messageString, messageIcon, messageComponent, messageObject };
94    else return null;
95  }
96  /**
97   * Получает выбранное в настоящий момент сообщение или тип вариантов
98   * @param panel Панель Message Type (Тип сообщения)
99   *           или Confirm (Подтверждение)
100  * @return Константа XXX_MESSAGE или XXX_OPTION,
101  *         выбранная из класса JOptionPane
102  */
103  public int getType(ButtonPanel panel)
104  {
105    String s = panel.getSelection();
106    try
107    {
108      return JOptionPane.class.getField(s).getInt(null);
109    }
110    catch (Exception e)
111    {
112      return -1;
113    }
114  }
115
116  /**
117   * Приемник действий для кнопки Show. Отображает диалоговое окно
118   * подтверждения (Confirm), ввода (Input), сообщения (Message) или
119   * выбора варианта (Option) в зависимости от типа диалогового окна,
120   * выбранного на панели Type
121  */
122  private class ShowAction implements ActionListener
123  {
124    public void actionPerformed(ActionEvent event)
125    {
126      if (typePanel.getSelection().equals("Confirm"))
127        JOptionPane.showConfirmDialog(OptionDialogFrame.this,
128          getMessage(), "Title", getType(optionTypePanel),
129          getType(messageTypePanel));
130      else if (typePanel.getSelection().equals("Input"))
131      {
132        if (inputPanel.getSelection().equals("Text field"))
133          JOptionPane.showInputDialog(
134            OptionDialogFrame.this, getMessage(), "Title",
135            getType(messageTypePanel));
136        else JOptionPane.showInputDialog(OptionDialogFrame.this,
137          getMessage(), "Title", getType(messageTypePanel), null,
138          new String[] { "Yellow", "Blue", "Red" }, "Blue");
139      }
140      else if (typePanel.getSelection().equals("Message"))
141        JOptionPane.showMessageDialog(OptionDialogFrame.this,
142          getMessage(), "Title", getType(messageTypePanel));
143    }
144  }
```

```
141     else if (typePanel.getSelection().equals("Option"))
142         JOptionPane.showOptionDialog(OptionDialogFrame.this,
143             getMessage(), "Title", getType(optionTypePanel),
144             getType(messageTypePanel), null, getOptions(), getOptions()[0]);
145     }
146 }
147 }
148 /**
149  * Компонент с окрашенной поверхностью
150  */
151 class SampleComponent extends JComponent
152 {
153     public void paintComponent(Graphics g)
154     {
155         Graphics2D g2 = (Graphics2D) g;
156         Rectangle2D rect =
157             new Rectangle2D.Double(0, 0, getWidth() - 1, getHeight() - 1);
158         g2.setPaint(Color.YELLOW);
159         g2.fill(rect);
160         g2.setPaint(Color.BLUE);
162         g2.draw(rect);
163     }
164
165     public Dimension getPreferredSize()
166     {
167         return new Dimension(10, 10);
168     }
169 }
```

Листинг 9.16. Исходный код из файла optionDialog/ButtonPanel.java

```
1 package optionDialog;
2
3 import javax.swing.*;
4
5 /**
6  * Панель с кнопками-переключателями, заключенная в рамку с заголовком
7  */
8 public class ButtonPanel extends JPanel
9 {
10     private ButtonGroup group;
11
12     /**
13      * Конструирует панель кнопок
14      * @param title Заголовок, отображаемый в рамке
15      * @param options Массив меток для кнопок-переключателей
16      */
17     public ButtonPanel(String title, String... options)
18     {
19         setBorder(BorderFactory.createTitledBorder(BorderFactory.
20             createEtchedBorder(), title));
21         setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
22         group = new ButtonGroup();
23         // создать по одной кнопке-переключателю на каждый вариант выбора
24         for (String option : options)
25         {
26             JRadioButton b = new JRadioButton(option);
```

```

27     b.setActionCommand(option);
28     add(b);
29     group.add(b);
30     b.setSelected(option == options[0]);
31 }
32 }
33
34 /**
35  * Получает выбранный в настоящий момент вариант
36  * @return Метка выбранной в настоящий момент кнопки-переключателя
37  */
38 public String getSelection()
39 {
40     return group.getSelection().getActionCommand();
41 }
42 }

```

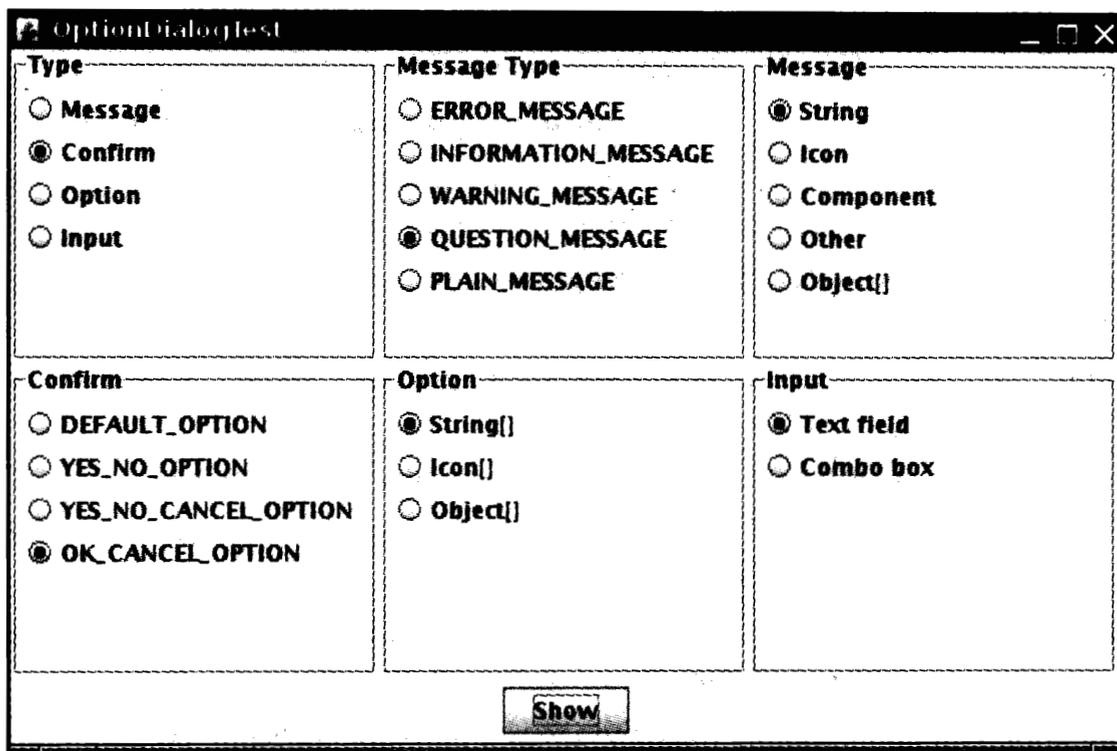


Рис. 9.37. Окно программы OptionDialogTest

javax.swing.JOptionPane 1.2

- static void showMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)
- static void showMessageDialog(Component parent, Object message, String title, int messageType)
- static void showMessageDialog(Component parent, Object message)
- static void showInternalMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)
- static void showInternalMessageDialog(Component parent, Object message, String title, int messageType)

- `static void showInternalMessageDialog(Component parent, Object message)`
Выводят на экран обычное диалоговое окно или внутреннее диалоговое окно для сообщения. (Внутреннее диалоговое окно воспроизводится только в пределах фрейма.)

Параметры:	<i>parent</i>	Родительский компонент (значение этого параметра может быть пустым (null))
	<i>message</i>	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов)
	<i>title</i>	Символьная строка с заголовком диалогового окна
	<i>messageType</i>	Одна из следующих констант: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE или PLAIN_MESSAGE
	<i>icon</i>	Пиктограмма, отображаемая вместо стандартной

- `static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)`
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showConfirmDialog(Component parent, Object message)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showInternalConfirmDialog(Component parent, Object message)`
Отображают обычное или внутреннее диалоговое окно для подтверждения. (Внутреннее диалоговое окно воспроизводится только в пределах фрейма.) Возвращают вариант, выбранный пользователем (одну из следующих констант: OK_OPTION, YES_OPTION или NO_OPTION), или же константу CLOSED_OPTION, если пользователь закрыл диалоговое окно.

Параметры:	<i>parent</i>	Родительский компонент (значение этого параметра может быть пустым (null))
	<i>message</i>	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов)
	<i>title</i>	Символьная строка с заголовком диалогового окна

<i>messageType</i>	Одна из следующих констант: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE или PLAIN_MESSAGE
<i>optionType</i>	Одна из следующих констант: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION или OK_CANCEL_OPTION
<i>icon</i>	Пиктограмма, отображаемая вместо стандартной

- `static int showOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`
- `static int showInternalOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`

Отображают обычное или внутреннее диалоговое окно для выбора разных вариантов. (Внутреннее диалоговое окно воспроизводится только в пределах фрейма.) Возвращают индекс варианта, выбранного пользователем, или же константу CLOSED_OPTION, если пользователь закрыл диалоговое окно.

Параметры:	<i>parent</i>	Родительский компонент (значение этого параметра может быть пустым (null))
	<i>message</i>	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов)
	<i>title</i>	Символьная строка с заголовком диалогового окна
	<i>messageType</i>	Одна из следующих констант: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE или PLAIN_MESSAGE
	<i>optionType</i>	Одна из следующих констант: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION или OK_CANCEL_OPTION
	<i>icon</i>	Пиктограмма, отображаемая вместо стандартной
	<i>default</i>	Вариант, предоставляемый пользователю по умолчанию

- `static Object showInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)`
- `static String showInputDialog(Component parent, Object message, String title, int messageType)`

- `static String showInputDialog(Component parent, Object message)`
- `static String showInputDialog(Object message)`
- `static String showInputDialog(Component parent, Object message, Object default) 1.4`
- `static String showInputDialog(Object message, Object default) 1.4`
- `static Object showInternalInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)`
- `static String showInternalInputDialog(Component parent, Object message, String title, int messageType)`
- `static String showInternalInputDialog(Component parent, Object message)`

Отображают обычное или внутреннее диалоговое окно для ввода. (Внутреннее диалоговое окно воспроизводится только в пределах фрейма.) Возвращают символьную строку, введенную пользователем, или же пустое значение `null`, если пользователь закрыл диалоговое окно.

<i>parent</i>	Родительский компонент (значение этого параметра может быть пустым (<code>null</code>))
<i>message</i>	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов)
<i>title</i>	Символьная строка с заголовком диалогового окна
<i>messageType</i>	Одна из следующих констант: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE или PLAIN_MESSAGE
<i>icon</i>	Пиктограмма, отображаемая вместо стандартной
<i>values</i>	Массив значений, доступных для выбора из комбинированного списка
<i>default</i>	Вариант, предоставляемый пользователю по умолчанию

Создание диалоговых окон

Как упоминалось ранее, для применения predefined диалоговых окон служит класс `JOptionPane`. В этом разделе будет показано, как создать диалоговое окно самостоятельно. На рис. 9.38 показано типичное режимное диалоговое окно, содержащее сообщение. Подобное окно обычно появляется на экране после того, как пользователь выберет пункт меню `About` (О программе).

Для того чтобы реализовать такое окно, необходимо создать подкласс, производный от класса `JDialog`. По существу, этот процесс ничем не отличается от создания главного окна приложения расширением класса `JFrame`. А точнее говоря, для этого нужно сделать следующее.

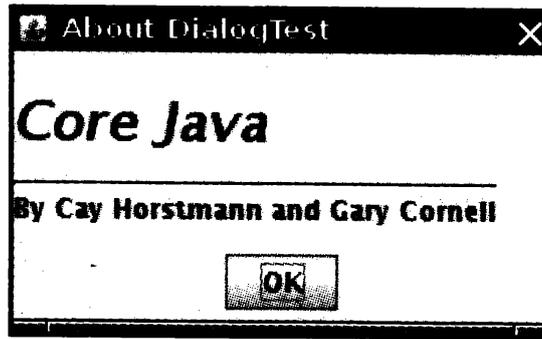


Рис. 9.38. Диалоговое окно About

1. Вызовите конструктор суперкласса `JDialog` в конструкторе вашего диалогового окна.
2. Введите в свое диалоговое окно компоненты пользовательского интерфейса.
3. Введите обработчики событий, наступающих в данном окне.
4. Установите размеры своего диалогового окна.

При вызове конструктора суперкласса следует указать *фрейм-владельца*, заголовок окна и признак *модальности*. Фрейм-владелец управляет местом отображения диалогового окна. Вместо владельца можно указать пустое значение `null`, и тогда диалоговое окно будет принадлежать скрытому фрейму.

Признак модальности означает, должны ли блокироваться другие окна приложения до тех пор, пока отображается данное диалоговое окно. Немодальные (т.е. безрежимные) окна не блокируют другие окна. А модальное (т.е. режимное) диалоговое окно блокирует все остальные окна приложения (за исключением производных от этого диалогового окна). Как правило, немодальные диалоговые окна служат для создания панелей инструментов, к которым постоянно открыт доступ. С другой стороны, модальные диалоговые окна обычно служат для того, чтобы принудить пользователя ввести необходимую информацию, прежде чем продолжить работу с приложением.



НА ЗАМЕТКУ! В версии Java SE 6 внедрены два дополнительных вида модальности. Документно-модальное диалоговое окно блокирует все окна, относящиеся к одному и тому же "документу", а точнее говоря, все окна с тем же самым родительским корневым окном, что и у данного диалогового окна. Это устраняет известные затруднения в справочных системах. В более старых версиях Java пользователи не могли взаимодействовать с окнами справочной системы, когда всплывало модальное диалоговое окно. Инструментально-модальное диалоговое окно блокирует все окна из одного и того же набора инструментов — программы на Java, запускающей несколько приложений вроде механизма апплетов в браузере. Дополнительные сведения на эту тему можно найти по адресу www.oracle.com/technetwork/articles/javase/modality-137604.html.

Ниже приведен фрагмент кода, в котором создается диалоговое окно.

```
public AboutDialog extends JDialog
{
    public AboutDialog(JFrame owner)
    {
        super(owner, "About DialogTest", true);
        add(new JLabel(
            "<html><h1><i>Core Java</i></h1><hr>
```

```

        By Cay Horstmann and Gary Cornell</html>"),
        BorderLayout.CENTER);

    JPanel panel = new JPanel();
    JButton ok = new JButton("OK");

    ok.addActionListener(new
        ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                setVisible(false);
            }
        });
    panel.add(ok);
    add(panel, BorderLayout.SOUTH);
    setSize(250, 150);
}
}

```

Как видите, конструктор диалогового окна вводит в него элементы пользовательского интерфейса, в данном случае метки и кнопку. Кроме того, он вводит обработчик событий от кнопки и задает размеры окна. Для того чтобы отобразить диалоговое окно, нужно создать новый объект типа `JDialog` и вызвать метод `setVisible()` следующим образом:

```

JDialog dialog = new AboutDialog(this);
dialog.setVisible(true);

```

На самом деле в программе, рассматриваемой здесь в качестве примера, диалоговое окно создается только один раз, а затем оно используется повторно всякий раз, когда пользователь щелкает на кнопке `About`, как показано ниже.

```

if (dialog == null) // первый раз
    dialog = new AboutDialog(this);
dialog.setVisible(true);

```

Когда пользователь щелкает на кнопке `OK`, диалоговое окно должно закрываться. Такая реакция на действия пользователя определяется в обработчике событий от кнопки `OK`, как следует из приведенного ниже фрагмента кода.

```

ok.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            setVisible(false);
        }
    });

```

Когда же пользователь закрывает диалоговое окно, щелкая на кнопке `Close`, оно исчезает из виду. Как и в классе `JFrame`, такое поведение можно изменить с помощью метода `setDefaultCloseOperation()`.

В листинге 9.17 приведен исходный код класса фрейма для примера программы, где демонстрируется применение режимного диалогового окна, создаваемого самостоятельно. А в листинге 9.18 представлен исходный код класса для создания этого диалогового окна.

Листинг 9.17. Исходный код из файла dialog/DialogFrame.java

```
1 package dialog;
2
3 import java.awt.event.*;
4 import javax.swing.*;
5
6 /**
7  * Фрейм со строкой меню, при выборе команды File→About из
8  * которого появляется диалоговое окно About
9  */
10 public class DialogFrame extends JFrame
11 {
12     private static final int DEFAULT_WIDTH = 300;
13     private static final int DEFAULT_HEIGHT = 200;
14     private AboutDialog dialog;
15
16     public DialogFrame()
17     {
18         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
19
20         // сконструировать меню File
21
22         JMenuBar menuBar = new JMenuBar();
23         setJMenuBar(menuBar);
24         JMenu fileMenu = new JMenu("File");
25         menuBar.add(fileMenu);
26
27         // ввести в меню пункты About и Exit
28
29         // При выборе пункта меню About открывается
30         // одноименное диалоговое окно
31
32         JMenuItem aboutItem = new JMenuItem("About");
33         aboutItem.addActionListener(new ActionListener()
34         {
35             public void actionPerformed(ActionEvent event)
36             {
37                 if (dialog == null) // первый раз
38                     dialog = new AboutDialog(DialogFrame.this);
39                 dialog.setVisible(true); // показать диалоговое окно
40             }
41         });
42         fileMenu.add(aboutItem);
43
44         // При выборе пункта меню Exit происходит выход из программы
45
46         JMenuItem exitItem = new JMenuItem("Exit");
47         exitItem.addActionListener(new ActionListener()
48         {
49             public void actionPerformed(ActionEvent event)
50             {
51                 System.exit(0);
52             }
53         });
54         fileMenu.add(exitItem);
55     }
56 }
```

Листинг 9.18. Исходный код из файла dialog/AboutDialog.java

```

1 package dialog;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Образец режимного диалогового окна, в котором выводится сообщение
9  * и ожидается до тех пор, пока пользователь не щелкнет на кнопке Ok
10 */
11 public class AboutDialog extends JDialog
12 {
13     public AboutDialog(JFrame owner)
14     {
15         super(owner, "About DialogTest", true);
16
17         // ввести HTML-метку по центру окна
18
19         add(
20             new JLabel(
21                 "<html><h1><i>Core Java</i></h1><hr>
22                 By Cay Horstmann and Gary Cornell</html>"),
23             BorderLayout.CENTER);
24
25         // При выборе кнопки Ok диалоговое окно закрывается
26
27         JButton ok = new JButton("Ok");
28         ok.addActionListener(new ActionListener()
29             {
30             public void actionPerformed(ActionEvent event)
31             {
32                 setVisible(false);
33             }
34             });
35         // ввести кнопку Ok в нижней части окна у южной его границы
36         JPanel panel = new JPanel();
37         panel.add(ok);
38         add(panel, BorderLayout.SOUTH);
39
40         pack();
41     }
42 }

```

javax.swing.JDialog 1.2

- `public JDialog(Frame parent, String title, boolean modal)`

Создает диалоговое окно, которое оказывается невидимым до тех пор, пока не будет показано явным образом.

Параметры:	<code>parent</code>	Фрейм-владелец диалогового окна
	<code>title</code>	Заголовок диалогового окна
	<code>modal</code>	Этот параметр принимает логическое значение <code>true</code> , если диалоговое окно является режимным (т.е. модальным) и блокирует доступ к остальным окнам

Обмен данными

Чаще всего диалоговые окна создаются для того, чтобы получить информацию от пользователя. Выше было показано, насколько просто создаются объекты типа `JDialog`: достаточно указать исходные данные и вызвать метод `setVisible(true)`, чтобы вывести окно на экран. А теперь покажем, как вводить данные в диалоговом окне. Обратите внимание на диалоговое окно, показанное на рис. 9.39. Его можно использовать для получения имени пользователя и пароля при подключении к оперативно доступной службе.

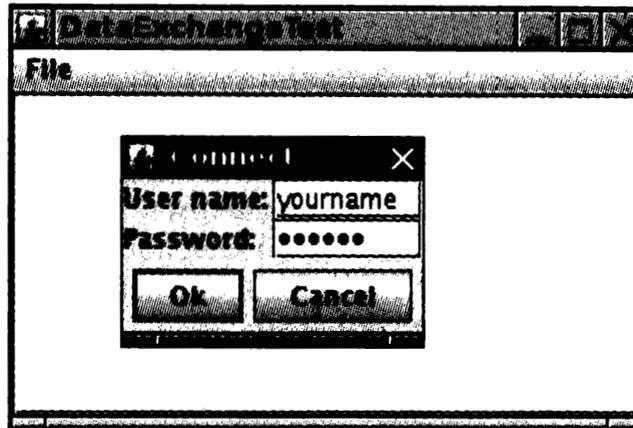


Рис. 9.39. Диалоговое окно для ввода имени пользователя и пароля

Для нормальной работы такого диалогового окна должны быть предусмотрены методы, формирующие данные по умолчанию. Например, в классе `PasswordChooser` имеется метод `setUser()` для ввода исходных значений, задаваемых по умолчанию в следующих полях:

```
public void setUser(User u)
{
    username.setText(u.getName());
}
```

После установки значений по умолчанию, если требуется, окно выводится на экран. Для этого вызывается метод `setVisible(true)`. Далее пользователь должен ввести данные в указанных полях и щелкнуть на кнопке `OK` или `Cancel`. В обработчиках событий от обеих кнопок вызывается метод `setVisible(false)`. При таком вызове выполняются действия, обратные тем, что происходят при вызове метода `setVisible(true)`. С другой стороны, пользователь может просто закрыть диалоговое окно. Если приемник событий в диалоговом окне не установлен, то выполняются обычные операции по закрытию окон. В итоге диалоговое окно становится невидимым, а вызов `setVisible(true)` завершается.

Следует особо подчеркнуть, что при вызове метода `setVisible(true)` выполнение программы приостанавливается до тех пор, пока пользователь не выполнит действие, приводящее к закрытию окна. Такой подход существенно упрощает реализацию режимных диалоговых окон.

В большинстве случаев требуется знать, подтвердил ли пользователь введенные им данные или отказался от их ввода. В программе, рассматриваемой здесь в качестве примера, применяется следующий подход. Перед обращением к диалоговому окну в переменной `ok` устанавливается логическое значение `false`. А логическое

значение `true` присваивается этой переменной только в обработчике событий от кнопки ОК. В этом случае введенные пользователем данные могут быть использованы в программе.



НА ЗАМЕТКУ! Передать данные из безрежимного (немодального) диалогового окна не так-то просто. При отображении такого окна вызов метода `setVisible(true)` не приводит к приостановке выполнения программы. Если пользователь выполнит манипуляции над элементами в безрежимном диалоговом окне и затем щелкнет на кнопке ОК, это окно должно уведомить соответствующий приемник событий в самой программе.

В рассматриваемом здесь примере программы имеются и другие заметные усовершенствования. При создании объекта типа `JDialog` должен быть указан фрейм-владелец. Но зачастую одно и то же диалоговое окно приходится отображать с разными фреймами. Поэтому намного удобнее, если фрейм-владелец задается, когда диалоговое окно *готово* к выводу на экран, а не при создании объекта типа `PasswordChooser`.

Этого можно добиться, если класс `PasswordChooser` будет расширять класс `JPanel`, а не класс `JDialog`. А объект типа `JDialog` можно создать по ходу выполнения метода `showDialog()` следующим образом:

```
public boolean showDialog(Frame owner, String title)
{
    ok = false;
    if (dialog == null || dialog.getOwner() != owner)
    {
        dialog = new JDialog(owner, true);
        dialog.add(this);
        dialog.pack();
    }

    dialog.setTitle(title);
    dialog.setVisible(true);
    return ok;
}
```

Следует заметить, что для большей безопасности значение параметра `owner` должно быть равно `null`. Можно пойти еще дальше. Ведь иногда фрейм-владелец оказывается недоступным. Но его можно вычислить как и любой родительский компонент из параметра `parent` следующим образом:

```
Frame owner;
if (parent instanceof Frame)
    owner = (Frame) parent;
else
    owner = (Frame) SwingUtilities.getAncestorOfClass(Frame.class, parent);
```

Именно такой подход применяется в приведенном ниже примере программы. Этот механизм используется и в классе `JOptionPane`.

Во многих диалоговых окнах имеется кнопка по умолчанию, которая выбирается автоматически, если пользователь нажимает клавишу ввода (в большинстве визуальных стилей ГПИ для этого служит клавиша `<Enter>`). Кнопка по умолчанию выделяется среди остальных компонентов ГПИ, чаще всего контуром. Для установки кнопки по умолчанию на *корневой панели* диалогового окна делается следующий вызов:

```
dialog.getRootPane().setDefaultButton(okButton);
```

Если вы собираетесь разместить элементы диалогового окна на панели, будьте внимательны: устанавливайте кнопку по умолчанию только после оформления панели в виде диалогового окна. Такая панель сама по себе не имеет корневой панели.

В листинге 9.19 приведен исходный код класса фрейма для примера программы, где демонстрируется обмен данными с диалоговым окном. А в листинге 9.20 представлен исходный код класса для этого диалогового окна.

Листинг 9.19. Исходный код из файла `dataExchange/DataExchangeFrame.java`

```
1 package dataExchange;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Фрейм со строкой меню, при выборе команды File→Connect
9  * из которого появляется диалоговое окно для ввода пароля
10 */
11 public class DataExchangeFrame extends JFrame
12 {
13     public static final int TEXT_ROWS = 20;
14     public static final int TEXT_COLUMNS = 40;
15     private PasswordChooser dialog = null;
16     private JTextArea textArea;
17
18     public DataExchangeFrame()
19     {
20         // сконструировать меню File
21         JMenuBar mbar = new JMenuBar();
22         setJMenuBar(mbar);
23         JMenu fileMenu = new JMenu("File");
24         mbar.add(fileMenu);
25
26         // ввести в меню пункты Connect и Exit
27         JMenuItem connectItem = new JMenuItem("Connect");
28         connectItem.addActionListener(new ConnectAction());
29         fileMenu.add(connectItem);
30
31         // При выборе пункта меню Exit происходит выход из программы
32         JMenuItem exitItem = new JMenuItem("Exit");
33         exitItem.addActionListener(new ActionListener()
34         {
35             public void actionPerformed(ActionEvent event)
36             {
37                 System.exit(0);
38             }
39         });
40         fileMenu.add(exitItem);
41
42         textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
43         add(new JScrollPane(textArea), BorderLayout.CENTER);
44         pack();
45     }
46     /**
47     * При выполнении команды Connect появляется
```

```
48  * диалоговое окно для ввода пароля
49  */
50
51  private class ConnectAction implements ActionListener
52  {
53      public void actionPerformed(ActionEvent event)
54      {
55          // При первом обращении конструируется диалоговое окно
56          if (dialog == null) dialog = new PasswordChooser();
57
58          // установить значения по умолчанию
59          dialog.setUser(new User("yourname", null));
60
61          // показать диалоговое окно
62          if (dialog.showDialog(DataExchangeFrame.this, "Connect"))
63          {
64              // Если пользователь подтвердил введенные данные,
65              // извлечь их для последующей обработки
66              User u = dialog.getUser();
67              textArea.append("user name = " + u.getName() + ", password = "
68                  + (new String(u.getPassword())) + "\n");
69          }
70      }
71  }
72 }
```

Листинг 9.20. Исходный код из файла dataExchange/PasswordChooser.java

```
1  package dataExchange;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  /**
8   * Окно для ввода пароля, отображаемое в диалоговом окне
9   */
10 public class PasswordChooser extends JPanel
11 {
12     private JTextField username;
13     private JPasswordField password;
14     private JButton okButton;
15     private boolean ok;
16     private JDialog dialog;
17     public PasswordChooser()
18     {
19         setLayout(new BorderLayout());
20
21         // сконструировать панель с полями для
22         // ввода имени пользователя и пароля
23         JPanel panel = new JPanel();
24         panel.setLayout(new GridLayout(2, 2));
25         panel.add(new JLabel("User name:"));
26         panel.add(username = new JTextField(""));
27         panel.add(new JLabel("Password:"));
28         panel.add(password = new JPasswordField(""));
29         add(panel, BorderLayout.CENTER);
```

```
30
31 // создать кнопки Ok и Cancel для закрытия диалогового окна
32 okButton = new JButton("Ok");
33 okButton.addActionListener(new ActionListener()
34 {
35     public void actionPerformed(ActionEvent event)
36     {
37         ok = true;
38         dialog.setVisible(false);
39     }
40 });
41
42 JButton cancelButton = new JButton("Cancel");
43 cancelButton.addActionListener(new ActionListener()
44 {
45     public void actionPerformed(ActionEvent event)
46     {
47         dialog.setVisible(false);
48     }
49 });
50
51 // ввести кнопки в нижней части окна у южной его границы
52 JPanel buttonPanel = new JPanel();
53 buttonPanel.add(okButton);
54 buttonPanel.add(cancelButton);
55 add(buttonPanel, BorderLayout.SOUTH);
56 }
57
58 /**
59  * Устанавливает диалоговое окно в исходное состояние
60  * @param u Данные о пользователе по умолчанию
61  */
62 public void setUser(User u)
63 {
64     username.setText(u.getName());
65 }
66 /**
67  * Получает данные, введенные в диалоговом окне
68  * @return Возвращает объект типа User, состояние которого
69  *         отражает введенные пользователем данные
70  */
71 public User getUser()
72 {
73     return new User(username.getText(), password.getPassword());
74 }
75
76 /**
77  * Отображает панель для ввода пароля в диалоговом окне
78  * @param parent Компонент из фрейма-владельца или
79  *         пустое значение null
80  * @param title Заголовок диалогового окна
81  */
82 public boolean showDialog(Component parent, String title)
83 {
84     ok = false;
85
86     // обнаружить фрейм-владелец
87     Frame owner = null;
```

```
88     if (parent instanceof Frame) owner = (Frame) parent;
89     else owner =
90         (Frame) SwingUtilities.getAncestorOfClass(Frame.class, parent);
91
92     // создать новое диалоговое окно при первом обращении
93     // или изменении фрейма-владельца
94     if (dialog == null || dialog.getOwner() != owner)
95     {
96         dialog = new JDialog(owner, true);
97         dialog.add(this);
98         dialog.getRootPane().setDefaultButton(okButton);
99         dialog.pack();
100    }
101
102    // установить заголовок и отобразить диалоговое окно
103    dialog.setTitle(title);
104    dialog.setVisible(true);
105    return ok;
106 }
107 }
```

javax.swing.SwingUtilities 1.2

- Container `getAncestorOfClass(Class c, Component comp)`
Возвращает наиболее глубоко вложенный родительский контейнер указанного компонента, принадлежащего заданному классу или одному из его подклассов.

javax.swing.JComponent 1.2

- JRootPane `getRootPane()`
Определяет корневую панель, содержащую данный компонент. Если у компонента отсутствует предшественник с корневой панелью, то возвращает пустое значение `null`.

javax.swing.JRootPane 1.2

- void `setDefaultButton(JButton button)`
Устанавливает кнопку по умолчанию на данной корневой панели. Чтобы запретить доступ к кнопке по умолчанию, этот метод вызывается с пустым значением `null` параметра `button`.

javax.swing.JButton 1.2

- boolean `isDefaultButton()`
Возвращает логическое значение `true`, если это кнопка по умолчанию на своей корневой панели.

Диалоговые окна для обращения с файлами

Во многих приложениях требуется открывать и сохранять файлы. Написать код для построения диалогового окна, позволяющего свободно перемещаться по каталогам файловой системы, не так-то просто. Впрочем, это и не требуется, чтобы не изобретать заново колесо! В библиотеке Swing имеется класс `JFileChooser`, позволяющий отображать диалоговое окно для обращения с файлами, удовлетворяющее потребностям большинства приложений. Это диалоговое окно всегда является режимным. Следует, однако, иметь в виду, что класс `JFileChooser` не расширяет класс `JDialog`. Вместо метода `setVisible(true)` для отображения диалогового окна при открытии файлов вызывается метод `showOpenDialog()`, а при сохранении файлов — метод `showSaveDialog()`. Кнопка, подтверждающая выбор файла, автоматически помечается как `Open` или `Save`. С помощью метода `showDialog()` можно задать свою собственную метку кнопки. На рис. 9.40 показан пример диалогового окна для выбора файлов.

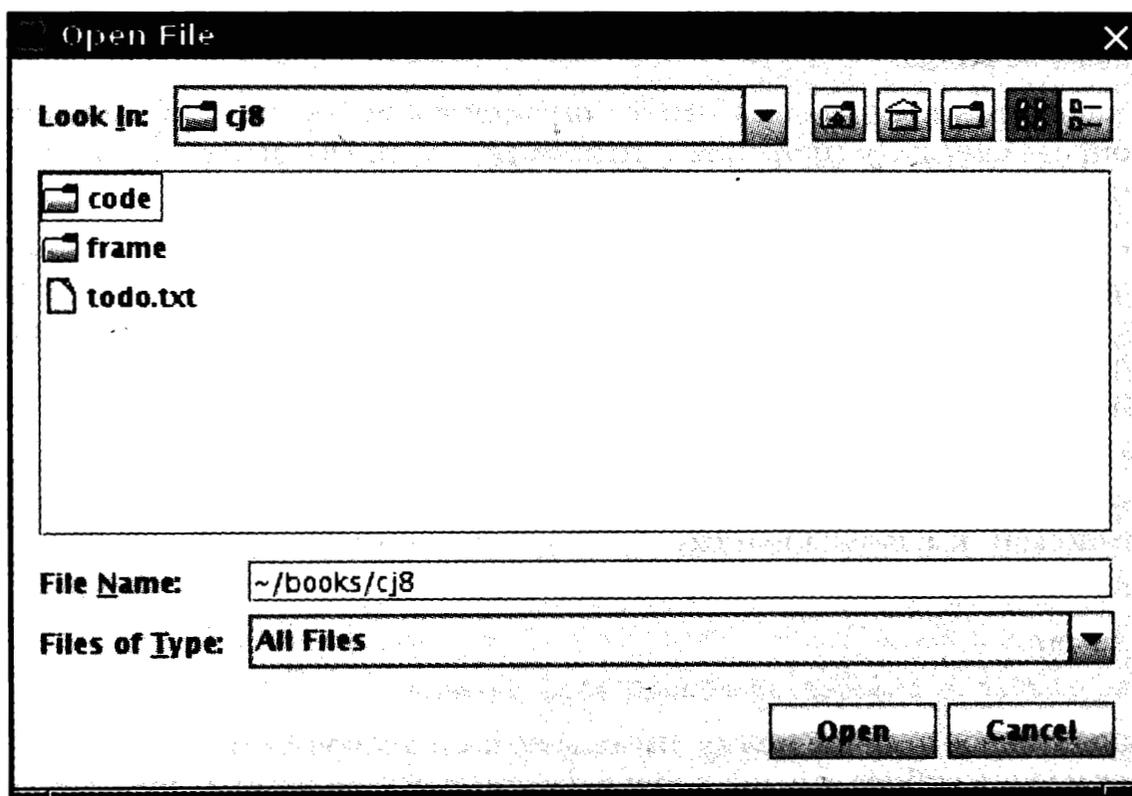


Рис. 9.40. Диалоговое окно для выбора файлов

Чтобы отобразить на экране диалоговое окно для выбора файлов и восстановить выбор, сделанный пользователем, выполните следующие действия.

1. Создайте объект класса `JFileChooser`, как показано ниже. В отличие от конструктора класса `JDialog`, в данном случае указывать родительский компонент не нужно. Такой подход позволяет повторно использовать диалоговое окно для выбора файлов во многих фреймах.

```
JFileChooser chooser = new JFileChooser();
```



СОВЕТ. Повторно использовать диалоговое окно для выбора файлов целесообразно, потому что конструктор класса `JFileChooser` иногда действует очень медленно. Особенно это заметно в Windows, когда пользователю доступно много подключенных сетевых дисков.

- Укажите каталог, вызвав метод `setCurrentDirectory()`. Например, чтобы задать текущий каталог, сделайте следующий вызов:

```
chooser.setCurrentDirectory(new File("."));
```

В этом вызове необходимо указать объект типа `File`. Подробнее класс `File` описывается в главе 1 второго тома данной книги, а до тех пор достаточно сказать, что у этого класса имеется конструктор `File(String filename)`, преобразующий символьную строку `filename` с именем файла или каталога в объект типа `File`.

- Если требуется предоставить пользователю имя файла, выбираемого по умолчанию, укажите его при вызове метода `setSelectedFile()` следующим образом:

```
chooser.setSelectedFile(new File(filename));
```

- Вызовите метод `setMultiSelectionEnabled()`, как показано ниже, чтобы дать пользователю возможность выбрать одновременно несколько файлов. Разумеется, делать это совсем не обязательно, ведь подобная возможность требуется далеко не всегда.

- Установите *фильтр файлов*, чтобы ограничить выбор пользователя файлами с определенным расширением (например, `.gif`). Подробнее о фильтрах файлов речь пойдет далее в этой главе.

- По умолчанию в данном диалоговом окне пользователь может выбирать только файлы. Если же требуется предоставить пользователю возможность выбирать целые каталоги, вызовите метод `setFileSelectionMode()` одним из следующих способов: `JFileChooser.FILES_ONLY` (по умолчанию), `JFileChooser.DIRECTORIES_ONLY` или `JFileChooser.FILES_AND_DIRECTORIES`.

- Отобразите данное диалоговое окно с помощью метода `showOpenDialog()` или `showSaveDialog()`. При вызове этих методов следует указать родительский компонент, как показано ниже.

```
int result = chooser.showOpenDialog(parent);
```

Или так:

```
int result = chooser.showSaveDialog(parent)
```

Единственное отличие между этими вызовами заключается в метке кнопки подтверждения выбора, т.е. той кнопки, на которой пользователь щелкает кнопкой мыши, выбирая файл. Можно также вызвать метод `showDialog()` и явно указать текст надписи на кнопке следующим образом:

```
int result = chooser.showDialog(parent, "Select");
```

При подобных вызовах управление возвращается только после того, как пользователь подтвердит свой выбор файла или откажется сделать выбор, закрыв диалоговое окно. Возвращаемое значение может быть одной из следующих констант: `JFileChooser.APPROVE_OPTION`, `JFileChooser.CANCEL_OPTION` или `JFileChooser.ERROR_OPTION`.

- Получите выбранный файл или несколько файлов с помощью метода `getSelectedFile()` или `getSelectedFiles()`. Эти методы возвращают один объект типа `File` или массив таких объектов. Если пользователю нужно знать лишь имя файла, вызовите метод `getPath()`, как в приведенной ниже строке кода.

```
String filename = chooser.getSelectedFile().getPath();
```

По большей части описанные выше действия довольно просты. Основные трудности возникают при использовании диалогового окна, в котором ограничивается выбор файлов пользователем. Допустим, пользователь должен выбирать только графические файлы формата GIF. Следовательно, в диалоговом окне должны отображаться только файлы, имеющие расширение `.gif`. Кроме того, пользователю нужно подсказать, что это за файлы. Но дело может усложниться, когда выбираются файлы изображений формата JPEG, поскольку они могут иметь расширение `.jpg` или `.jpeg`. Для преодоления подобных трудностей разработчики предложили следующий изящный механизм: чтобы ограничить спектр отображаемых файлов, достаточно предоставить объект подкласса, производного от абстрактного класса `javax.swing.filechooser.FileFilter`. В этом случае диалоговое окно для выбора файлов передает фильтру каждый файл и отображает только отфильтрованные файлы.

На момент написания этой книги были известны только два таких подкласса: фильтр, задаваемый по умолчанию и пропускающий все файлы, а также фильтр, пропускающий все файлы с указанным расширением. Но можно создать и свой собственный фильтр. Для этого достаточно реализовать следующие два метода, которые в классе `FileFilter` объявлены как абстрактные:

```
public boolean accept(File f);  
public String getDescription();
```

Первый из приведенных выше методов проверяет, удовлетворяет ли файл накладываемым ограничениям. А второй метод возвращает описание типа файлов, которые могут отображаться в диалоговом окне.



НА ЗАМЕТКУ! В состав пакета `java.io` входит также интерфейс `FileFilter` (совершенно не связанный с описанным выше одноименным абстрактным классом). В этом интерфейсе объявлен только один метод — `boolean accept(File f)`. Он используется в методе `listFiles()` из класса `File` для вывода списка файлов, находящихся в каталоге. Совершенно непонятно, почему разработчики библиотеки `Swing` не стали расширять этот интерфейс. Возможно, библиотека классов `Java` настолько сложная, что даже ее разработчики не знают обо всех стандартных классах и интерфейсах!

Если в программе одновременно импортируются пакеты `java.io` и `javax.swing.filechooser`, то придется каким-то образом разрешать конфликт имен. Поэтому вместо пакета `javax.swing.filechooser.*` проще импортировать класс `javax.swing.filechooser.FileFilter`.

Имея в своем распоряжении объект фильтра файлов, можно воспользоваться методом `setFileFilter()` из класса `JFileChooser`, чтобы установить этот фильтр в объекте диалогового окна для выбора файлов следующим образом:

```
chooser.setFileFilter(new FileNameExtensionFilter("Image files", "gif", "jpg");
```

Аналогичным образом можно установить несколько фильтров в следующей последовательности вызовов:

```
chooser.addChoosableFileFilter(filter1);  
chooser.addChoosableFileFilter(filter2);  
...
```

Пользователь выбирает фильтр из комбинированного списка в нижней части диалогового окна для выбора файлов. По умолчанию в нем всегда присутствует фильтр `All files` (Все файлы). Это удобно, особенно в том случае, когда пользователю

прикладной программы нужно выбрать файл с нестандартным расширением. Но если требуется подавить фильтр All files, то достаточно сделать следующий вызов:

```
chooser.setAcceptAllFileFilterUsed(false)
```



ВНИМАНИЕ! Если одно и то же диалоговое окно используется для загрузки и сохранения разнотипных файлов, вызовите метод `chooser.resetChoosableFilters()`, чтобы очистить старые фильтры перед установкой новых.

И, наконец, в диалоговом окне для выбора файлов каждый файл можно сопроводить специальной пиктограммой и кратким описанием. Для этого следует предоставить экземпляр класса, расширяющего класс `FileView` из пакета `javax.swing.filechooser`. Это довольно сложно и не всегда оправдано. Обычно внешний вид файла разработчика прикладной программы не интересует, поскольку эту задачу автоматически решают подключаемые визуальные стили ГПИ. Но если файлы определенного типа требуется сопроводить специальными пиктограммами, то можно задать свой собственный стиль отображения файлов. Для этого нужно расширить класс `FileView` и реализовать приведенные ниже пять методов, а затем вызвать метод `setFileView()`, чтобы связать нужное представление файлов с диалоговым окном для их выбора.

```
Icon getIcon(File f);
String getName(File f);
String getDescription(File f);
String getTypeDescription(File f);
Boolean isTraversable(File f);
```

Диалоговое окно для выбора файлов вызывает эти методы для каждого отображаемого в нем файла или каталога. Если метод возвращает пустую ссылку типа `null` на пиктограмму, имя или описание файла, в данном диалоговом окне используется визуальный стиль, устанавливаемый по умолчанию. И это правильный подход, поскольку он позволяет применять особый стиль только к отдельным типам файлов.

Для того чтобы выяснить, следует ли открывать каталог, выбранный пользователем, в диалоговом окне для выбора файлов вызывается метод `isTraversable()`. Следует, однако, иметь в виду, что этот метод возвращает объект класса `Boolean`, а не значение типа `boolean`! И хотя это не совсем обычно, тем не менее очень удобно. Так, если нет особых требований к визуальному стилю, достаточно вернуть пустое значение `null`. В таком случае в диалоговом окне для выбора файлов используется стиль отображения файлов, устанавливаемый по умолчанию. Иными словами, данный метод возвращает объект типа `Boolean`, чтобы дать возможность выбрать одно из трех: открывать каталог (`Boolean.TRUE`), не открывать каталог (`Boolean.FALSE`) и неважно (`null`).

Программа, рассматриваемая здесь в качестве примера, содержит простой класс представления файлов. Этот класс отображает определенную пиктограмму всякий раз, когда файл проходит фильтр. В данной программе этот класс служит для отображения панели с пиктограммами для всех графических файлов, как показано ниже.

```
class FileIconView extends FileView
{
    public FileIconView(FileFilter aFilter, Icon anIcon)
    {
        filter = aFilter;
        icon = anIcon;
    }
}
```

```
public Icon getIcon(File f)
{
    if (!f.isDirectory() && filter.accept(f))
        return icon;
    else return null;
}
private FileFilter filter;
private Icon icon;
}
```

Чтобы установить это представление файлов, в диалоговом окне для выбора файлов вызывается метод `setFileView()`:

```
chooser.setFileView(new FileIconView(filter,
    new ImageIcon("palette.gif")));
```

Панель с пиктограммами отображается в диалоговом окне для выбора файлов рядом со всеми отфильтрованными файлами, а для отображения всех остальных файлов используется представление, устанавливаемое по умолчанию. Естественно, что для отбора файлов служит один тот же фильтр, установленный в диалоговом окне для выбора файлов.



СОВЕТ. В каталоге `demo/jfc/FileChooserDemo` установки JDK можно найти более полезный класс `ExampleFileView`. Этот класс позволяет связывать пиктограммы и описания с любыми расширениями файлов.

И наконец, диалоговое окно для выбора файлов можно снабдить *вспомогательным компонентом*. В качестве примера на рис. 9.41 показан такой компонент, позволяющий отображать в миниатюрном виде содержимое выбранного в данный момент файла, помимо списка файлов.

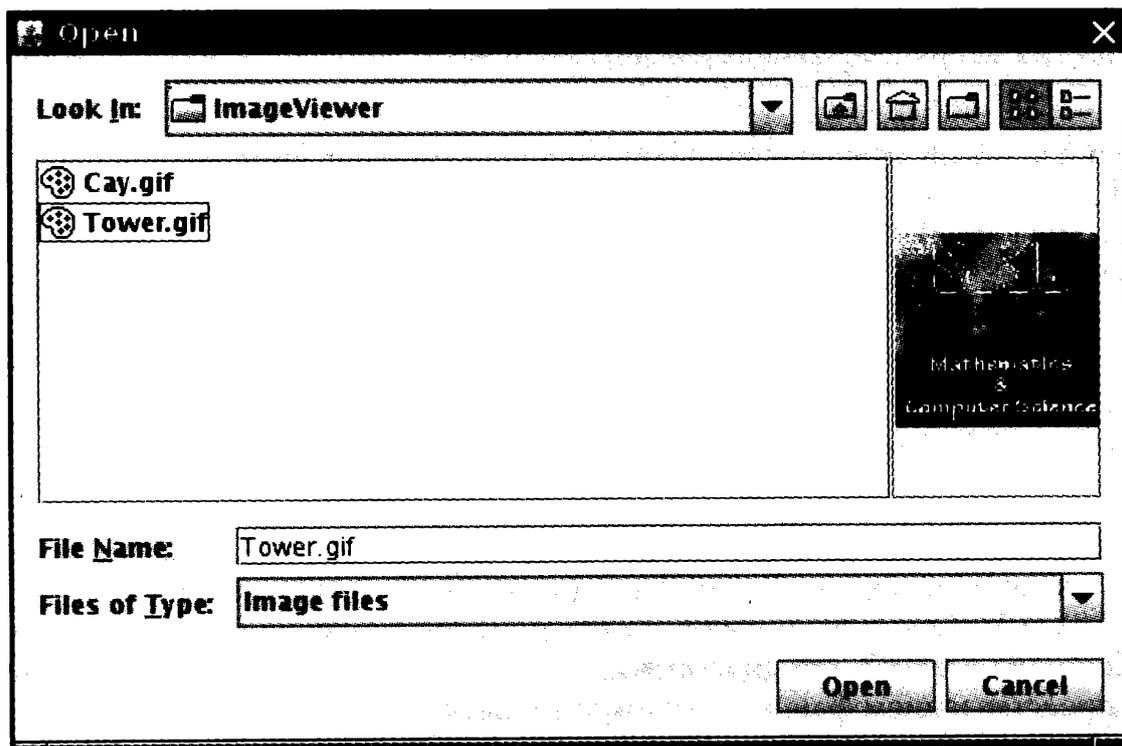


Рис. 9.41. Диалоговое окно для выбора файлов со вспомогательным компонентом для предварительного просмотра выбранных файлов

В качестве вспомогательного может служить любой компонент из библиотеки Swing. В данном примере расширяется класс `JLabel`, а в качестве пиктограммы используется уменьшенная копия изображения, хранящегося в выбранном файле, как показано ниже.

```
class ImagePreviewer extends JLabel
{
    public ImagePreviewer(JFileChooser chooser)
    {
        setPreferredSize(new Dimension(100, 100));
        setBorder(BorderFactory.createEtchedBorder());
    }

    public void loadImage(File f)
    {
        ImageIcon icon = new ImageIcon(f.getPath());
        if(icon.getIconWidth() > getWidth())
            icon = new ImageIcon(icon.getImage().getScaledInstance(
                getWidth(), -1, Image.SCALE_DEFAULT));
        setIcon(icon);
        repaint();
    }
}
```

Остается лишь преодолеть еще одно затруднение. Предварительно просматриваемое изображение требуется обновлять всякий раз, когда пользователь выбирает новый файл. С этой целью в диалоговом окне для выбора файлов применяется механизм `JavaBeans`, уведомляющий заинтересованные приемники событий об изменениях свойств данного окна. Выбранный файл — это свойство, которое можно отслеживать с помощью установленного приемника событий типа `PropertyChangeListener`. Более подробно этот механизм обсуждается в главе 8 второго тома данной книги. В приведенном ниже фрагменте кода показано, каким образом организуется перехват уведомлений, направляемых приемнику событий.

```
chooser.addPropertyChangeListener(new
    PropertyChangeListener()
    {
        public void propertyChange(PropertyChangeEvent event)
        {
            if (event.getPropertyName() ==
                JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
            {
                File newFile = (File) event.getNewValue();
                // обновить вспомогательный компонент
                . . .
            }
        }
    });
```

В рассматриваемом здесь примере программы этот код находится в теле конструктора класса `ImagePreviewer`. Исходный код этой программы приведен в листингах 9.21–9.23. Она представляет собой модифицированную версию программы `ImageViewer` из главы 2, где диалоговое окно для выбора файлов дополнено специальным представлением файлов и вспомогательным компонентом для предварительного просмотра их содержимого.

Листинг 9.21. Исходный код из файла fileChooser/ImageViewerFrame.java

```
1 package fileChooser;
2
3 import java.awt.event.*;
4 import java.io.*;
5 import javax.swing.*;
6 import javax.swing.filechooser.*;
7
8 /**
9  * Фрейм с меню для загрузки файла изображения и областью его отображения.
10 */
11 public class ImageViewerFrame extends JFrame
12 {
13     private static final int DEFAULT_WIDTH = 300;
14     private static final int DEFAULT_HEIGHT = 400;
15     private JLabel label;
16     private JFileChooser chooser;
17
18     public ImageViewerFrame()
19     {
20         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
21
22         // установить строку меню
23         JMenuBar menuBar = new JMenuBar();
24         setJMenuBar(menuBar);
25         JMenu menu = new JMenu("File");
26         menuBar.add(menu);
27
28         JMenuItem openItem = new JMenuItem("Open");
29         menu.add(openItem);
30         openItem.addActionListener(new ActionListener()
31         {
32             public void actionPerformed(ActionEvent event)
33             {
34                 chooser.setCurrentDirectory(new File("."));
35
36                 // показать диалоговое окно для выбора файлов
37                 int result = chooser.showOpenDialog(ImageViewerFrame.this);
38
39                 // если файл изображения подходит, выбрать
40                 // его в качестве пиктограммы для метки
41                 if (result == JFileChooser.APPROVE_OPTION)
42                 {
43                     String name = chooser.getSelectedFile().getPath();
44                     label.setIcon(new ImageIcon(name));
45                     pack();
46                 }
47             }
48         });
49
50         JMenuItem exitItem = new JMenuItem("Exit");
51         menu.add(exitItem);
52         exitItem.addActionListener(new ActionListener()
53         {
54             public void actionPerformed(ActionEvent event)
55             {
```

```

56         System.exit(0);
57     }
58     });
59
60     // использовать метку для показа изображений
61     label = new JLabel();
62     add(label);
63
64     // установить окно для выбора файлов
65     chooser = new JFileChooser();
66
67     // принимать все файлы с расширением .jpg, .jpeg, .gif
68     /*
69     final ExtensionFileFilter filter = new ExtensionFileFilter();
70     filter.addExtension("jpg");
71     filter.addExtension("jpeg");
72     filter.addExtension("gif");
73     filter.setDescription("Image files");
74     */
75     FileNameExtensionFilter filter =
76         new FileNameExtensionFilter("Image files", "jpg", "jpeg", "gif");
77     chooser.setFileFilter(filter);
78
79     chooser.setAccessory(new ImagePreviewer(chooser));
81
82     chooser.setFileView(new FileIconView(filter,
81         new ImageIcon("palette.gif")));
82 }
83 }

```

Листинг 9.22. Исходный код из файла fileChooser/ImagePreviewer.java

```

1 package fileChooser;
2
3 import java.awt.*;
4 import java.beans.*;
5 import java.io.*;
6 import javax.swing.*;
7
8 /**
9  * Вспомогательный компонент предварительного просмотра
10 * изображений в диалоговом окне для выбора файлов
11 */
12 public class ImagePreviewer extends JLabel
13 {
14     /**
15     * Конструирует объект типа ImagePreviewer
16     * @param chooser Диалоговое окно для выбора файлов, изменение
17     * свойств в котором влечет за собой изменение в предварительно
18     * просматриваемом виде изображения из выбранного файла
19     */
20     public ImagePreviewer(JFileChooser chooser)
21     {
22         setPreferredSize(new Dimension(100, 100));
23         setBorder(BorderFactory.createEtchedBorder());
24

```

```
25     chooser.addPropertyChangeListener(new PropertyChangeListener()
26     {
27         public void propertyChange(PropertyChangeEvent event)
28         {
29             if (event.getPropertyName() ==
30                 JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
31             {
32                 // пользователь выбрал новый файл
33                 File f = (File) event.getNewValue();
34                 if (f == null)
35                 {
36                     setIcon(null);
37                     return;
38                 }
39                 // ввести изображение в пиктограмму
40                 ImageIcon icon = new ImageIcon(f.getPath());
41
42                 // если пиктограмма слишком велика, подогнать ее по размеру
43                 if (icon.getIconWidth() > getWidth()) icon =
44                     new ImageIcon(icon.getImage()
45                         .getScaledInstance(getWidth(), -1, Image.SCALE_DEFAULT));
46
47                 setIcon(icon);
48             }
49         }
50     });
51 }
```

Листинг 9.23. Исходный код из файла fileChooser/FileIconView.java

```
1 package fileChooser;
2
3 import java.io.*;
4 import javax.swing.*;
5 import javax.swing.filechooser.*;
6 import javax.swing.filechooser.FileFilter;
7
8 /**
9  * Представление файлов для отображения пиктограмм
10 * рядом с именами всех отфильтрованных файлов
11 */
12 public class FileIconView extends FileView
13 {
14     private FileFilter filter;
15     private Icon icon;
16
17     /**
18     * Конструирует объект типа FileIconView
19     * @param aFilter Фильтр файлов. Все отфильтрованные им
20     *               файлы будут отображены с пиктограммой
21     * @param anIcon Пиктограмма, отображаемая вместе со всеми
22     *               принятыми и отфильтрованными файлами
23     */
24     public FileIconView(FileFilter aFilter, Icon anIcon)
25     {
```

```
26     filter = aFilter;
27     icon = anIcon;
28 }
29
30 public Icon getIcon(File f)
31 {
32     if (!f.isDirectory() && filter.accept(f)) return icon;
33     else return null;
34 }
35 }
```

javax.swing.JFileChooser 1.2

- `JFileChooser()`
Создает диалоговое окно для выбора файлов, которое можно использовать во многих фреймах.
- `void setCurrentDirectory(File dir)`
Задаёт исходный каталог, содержимое которого отображается в диалоговом окне для выбора файлов.
- `void setSelectedFile(File file)`
- `void setSelectedFiles(File[] file)`
Задают файл, выбираемый в диалоговом окне по умолчанию.
- `void setMultiSelectionEnabled(boolean b)`
Устанавливает или отменяет режим выбора нескольких файлов.
- `void setFileSelectionMode(int mode)`
Позволяет выбирать только файлы (по умолчанию), только каталоги или же каталоги вместе с файлами. Параметр `mode` может принимать следующие значения: `JFileChooser.FILES_ONLY`, `JFileChooser.DIRECTORIES_ONLY` и `JFileChooser.FILES_AND_DIRECTORIES`.
- `int showOpenDialog(Component parent)`
- `int showSaveDialog(Component parent)`
- `int showDialog(Component parent, String approveButtonText)`
Отображают диалоговое окно с кнопкой подтверждения выбора, обозначенной меткой `Open`, `Save` или произвольной меткой, указанной в символьной строке `approveButtonText`. Возвращают следующие значения: `APPROVE_OPTION`, `CANCEL_OPTION` (если пользователь щелкнул на кнопке `Cancel` или закрыл диалоговое окно) или `ERROR_OPTION` (если возникла ошибка).
- `File getSelectedFile()`
- `File[] getSelectedFiles()`
Возвращают файл или несколько файлов, выбранных пользователем, а если он ничего не выбрал, — пустое значение `null`.
- `void setFileFilter(FileFilter filter)`
Устанавливает маску файлов в диалоговом окне для выбора файлов. В этом окне отображаются только те файлы, для которых метод `filter.accept()` возвращает логическое значение `true`. Кроме того, вводит фильтр в список выбираемых фильтров.

- `void addChoosableFileFilter(FileFilter filter)`
Вводит фильтр в список выбираемых фильтров.
- `void setAcceptAllFileFilterUsed(boolean b)`
Вводит все файлы в комбинированный список выбираемых фильтров или удаляет их из этого списка.
- `void resetChoosableFileFilters()`
Очищает список фильтров, где остается фильтр всех файлов, если только он не удален из списка специально.
- `void setFileView(FileView view)`
Устанавливает представление файлов для предоставления сведений о файлах, отображаемых в диалоговом окне для выбора файлов.
- `void setAccessory(JComponent component)`
Устанавливает вспомогательный компонент.

javax.swing.filechooser.FileFilter 1.2

- `boolean accept(File f)`
Возвращает логическое значение `true`, если указанный файл должен отображаться в диалоговом окне.
- `String getDescription()`
Возвращает описание указанного фильтра, например "Image files (*.gif, *.jpeg)" (Файлы изображений с расширением *.gif и *.jpeg).

javax.swing.filechooser.FileNameExtensionFilter 6

- `FileNameExtensionFilter(String description, String ... extensions)`
Конструирует фильтр файлов с заданным описанием, принимающий все каталоги и все файлы, имена которых оканчиваются точкой и последующей символьной строкой одного из указанных расширений.

javax.swing.filechooser.FileView 1.2

- `String getName(File f)`
Возвращает имя файла `f` или пустое значение `null`. Обычно возвращается результат вызова метода `f.getName()`.
- `String getDescription(File f)`
Возвращает удобочитаемое описание файла `f` или пустое значение `null`. Так, если файл `f` представляет собой HTML-документ, этот метод может вернуть его заголовок.

- `String getTypeDescription(File f)`

Возвращает удобочитаемое описание типа файла *f* или пустое значение `null`. Так, если файл *f* представляет собой HTML-документ, этот метод может вернуть символьную строку "Hypertext document".

- `Icon getIcon(File f)`

Возвращает пиктограмму, назначенную для файла *f*, или пустое значение `null`. Так, если *f* — файл формата JPEG, этот метод может вернуть пиктограмму с миниатюрным видом его содержимого.

- `Boolean isTraversable(File f)`

Если пользователь может открыть указанный каталог, возвращается значение `Boolean.TRUE`. Если каталог представляет собой составной документ, может быть возвращено значение `Boolean.FALSE`. Подобно методам из файла `FileView`, этот метод может возвращать пустое значение `null`, отмечая тот факт, что в диалоговом окне для выбора файлов должно быть использовано представление файлов, устанавливаемое по умолчанию.

Диалоговые окна для выбора цвета

Как было показано ранее, качественное диалоговое окно для выбора файлов — довольно сложный элемент пользовательского интерфейса, который вряд ли стоит реализовывать самостоятельно. Многие инструментальные средства для создания ГПИ дают возможность формировать диалоговые окна для выбора даты или времени, валюты, шрифтов, цвета и т.п. Это приносит двойную выгоду: разработчики прикладных программ могут использовать готовые высококачественные компоненты, а пользователи получают удобный интерфейс.

На данный момент в библиотеке `Swing`, кроме окна для выбора файлов, предусмотрено лишь одно подобное диалоговое окно, позволяющее выбирать цвет. Это окно реализовано средствами класса `JColorChooser`. Внешний вид различных вкладок окна для выбора цвета показан на рис. 9.42–9.44. Как и класс `JFileChooser`, класс `JColorChooser` является компонентом, а не диалоговым окном. Тем не менее он содержит служебные методы, позволяющие создавать диалоговые окна для выбора цвета.

Режимное диалоговое окно для выбора цвета создается следующим образом:

```
Color selectedColor = JColorChooser.showDialog(parent, title, initialColor);
```

Выбор цвета можно обеспечить и в безрежимном диалоговом окне. Для этого нужно предоставить следующее.

- Родительский компонент.
- Заголовок диалогового окна.
- Признак, управляющий выбором режимного или безрежимного диалогового окна.
- Компонент для выбора цвета.
- Приемники событий от кнопок `OK` и `Cancel` (или пустое значение `null`, если приемники не требуются).

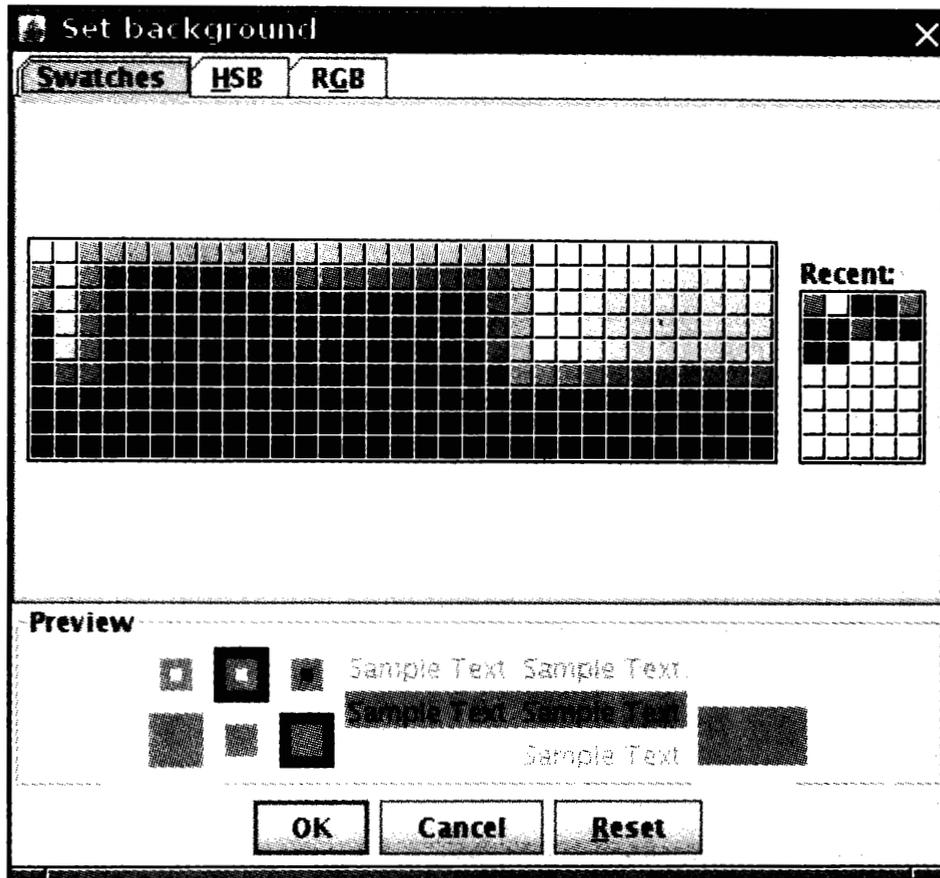


Рис. 9.42. Панель Swatches (Образцы цвета) диалогового окна для выбора цвета

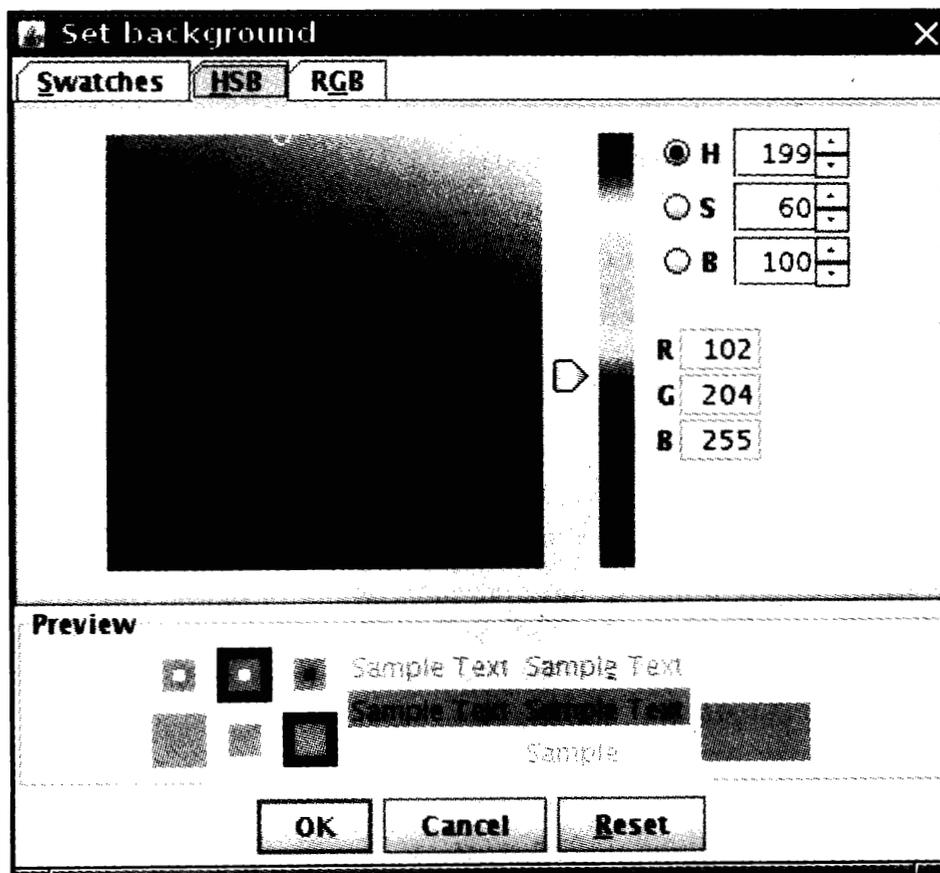


Рис. 9.43. Панель HSB (оттенок–насыщенность–яркость) диалогового окна для выбора цвета

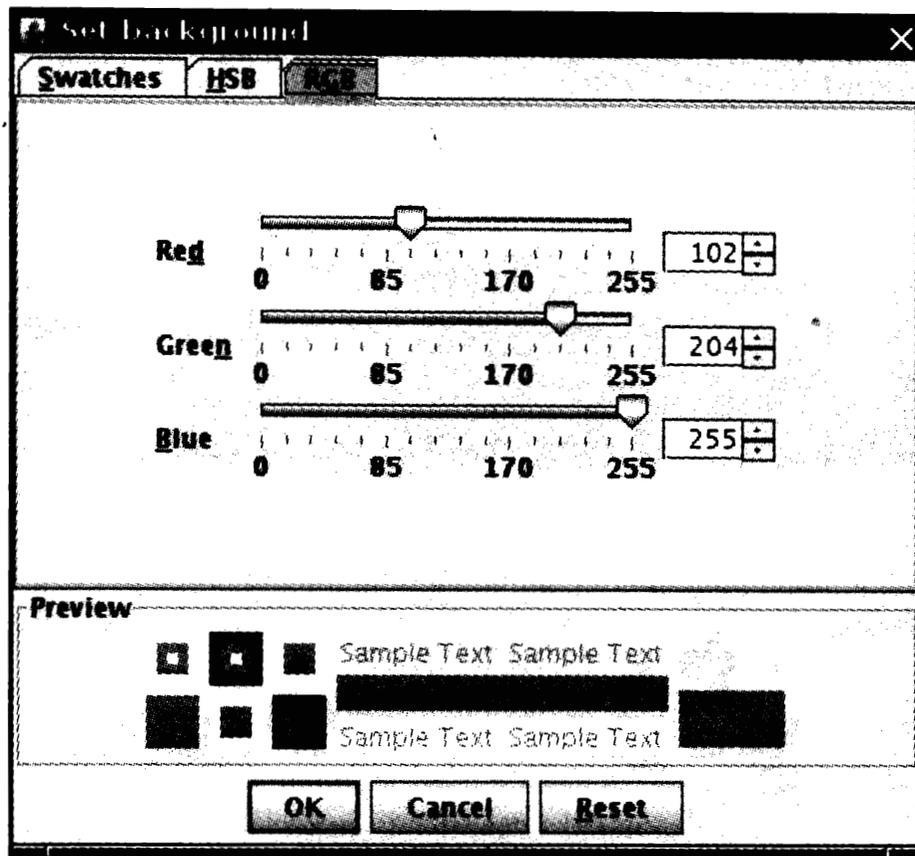


Рис. 9.44. Панель RGB (красный–зеленый–синий) диалогового окна для выбора цвета

В приведенном ниже фрагменте кода показано, как создается безрежимное диалоговое окно для установки цвета фона. Цвет фона устанавливается после щелчка на кнопке ОК.

```
chooser = new JColorChooser();
dialog = JColorChooser.createDialog(
    parent,
    "Background Color",
    false /* не режимное окно */,
    chooser,
    new ActionListener() // приемник событий от кнопки ОК
    {
        public void actionPerformed(ActionEvent event)
        {
            setBackground(chooser.getColor());
        }
    },
    null /* приемник событий от кнопки Cancel не требуется */);
```

Еще лучше предоставить пользователю возможность сразу увидеть результат выбора. Для отслеживания результатов выбора цвета необходимо получить модель выбора и приемник изменений в диалоговом окне, как показано ниже.

```
chooser.getSelectionModel().addChangeListener(new
ChangeListener()
{
    public void stateChanged(ChangeEvent event)
    {
        обработать результат вызова метода chooser.getColor();
    }
});
```

В данном случае кнопки OK и Cancel становятся лишними. Достаточно ввести в безрежимное диалоговое окно один только компонент для выбора цвета следующим образом:

```
dialog = new JDialog(parent, false /* не режимное окно */);
dialog.add(chooser);
dialog.pack();
```

В примере программы, исходный код которой приведен в листинге 9.24, демонстрируются три вида диалоговых окон. Если щелкнуть на кнопке Modal, то откроется режимное диалоговое окно, в котором следует непременно выбрать цвет, прежде чем сделать что-нибудь другое. Если же щелкнуть на кнопке Modeless, откроется безрежимное диалоговое окно, но внесенные в цвет изменения вступят в действие только после закрытия этого окна щелчком на кнопке OK. А если щелкнуть на кнопке Immediate, то откроется безрежимное диалоговое окно без кнопок. И как только в нем будет выбран новый цвет, сразу же изменится цвет фона панели.

Листинг 9.24. Исходный код из файла colorChooser/ColorChooserPanel.java

```
1 package colorChooser;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 import javax.swing.event.*;
7
8 /**
9  * Панель с кнопками для открытия трех видов
10  * диалоговых окон для выбора цвета
11  */
12 public class ColorChooserPanel extends JPanel
13 {
14     public ColorChooserPanel()
15     {
16         JButton modalButton = new JButton("Modal");
17         modalButton.addActionListener(new ModalListener());
18         add(modalButton);
19
20         JButton modelessButton = new JButton("Modeless");
21         modelessButton.addActionListener(new ModelessListener());
22         add(modelessButton);
23
24         JButton immediateButton = new JButton("Immediate");
25         immediateButton.addActionListener(new ImmediateListener());
26         add(immediateButton);
27     }
28
29     /**
30      * Этот приемник событий открывает режимное диалоговое
31      * окно для выбора цвета
32      */
33     private class ModalListener implements ActionListener
34     {
35         public void actionPerformed(ActionEvent event)
36         {
37             Color defaultColor = getBackground();
```

```
38     Color selected = JColorChooser.showDialog(
39         ColorChooserPanel.this, "Set background", defaultColor);
40     if (selected != null) setBackground(selected);
41 }
42 }
43
44 /**
45  * Этот приемник событий открывает диалоговое окно
46  * для выбора цвета. Цвет фона панели изменится, как только
47  * пользователь щелкнет на кнопке OK
48  */
49 private class ModelessListener implements ActionListener
50 {
51     private JDialog dialog;
52     private JColorChooser chooser;
53
54     public ModelessListener()
55     {
56         chooser = new JColorChooser();
57         dialog = JColorChooser.createDialog(
58             ColorChooserPanel.this, "Background Color",
59             false /* не режимное окно */, chooser,
60             new ActionListener() // приемник событий от кнопки OK
61             {
62                 public void actionPerformed(ActionEvent event)
63                 {
64                     setBackground(chooser.getColor());
65                 }, null /* приемник событий от кнопки Cancel не требуется */);
66     }
67
68     public void actionPerformed(ActionEvent event)
69     {
70         chooser.setColor(getBackground());
71         dialog.setVisible(true);
72     }
73 }
74
75 /**
76  * Этот приемник событий открывает диалоговое окно
77  * для выбора цвета. Цвет фона панели изменится, как только
78  * пользователь выберет новый цвет, не закрывая окно
79  */
80 private class ImmediateListener implements ActionListener
81 {
82     private JDialog dialog;
83     private JColorChooser chooser;
84
85     public ImmediateListener()
86     {
87         chooser = new JColorChooser();
88         chooser.getSelectionModel().addChangeListener(new ChangeListener()
89         {
90             public void stateChanged(ChangeEvent event)
91             {
92                 setBackground(chooser.getColor());
93             }
94         });
95     }
96 }
```

```
94         });
95         dialog = new JDialog((Frame) null, false /* не режимное окно */);
96         dialog.add(chooser);
97         dialog.pack();
98     }
99
100    public void actionPerformed(ActionEvent event)
101    {
102        chooser.setColor(getBackground());
103        dialog.setVisible(true);
104    }
105 }
102 }
```

javax.swing.JColorChooser 1.2

- `JColorChooser()`
Создает компонент для выбора цвета, где исходным цветом считается белый.
- `Color getColor()`
- `void setColor(Color c)`
Получают и устанавливают текущий цвет для выбора в данном компоненте.
- `static Color showDialog(Component parent, String title, Color initialColor)`
Отображает режимное диалоговое окно, содержащее компонент для выбора цвета.
Параметры:

<code>parent</code>	Компонент, в котором отображается диалоговое окно
<code>title</code>	Заголовок фрейма с диалоговым окном
<code>initialColor</code>	Исходный цвет, отображаемый в компоненте для выбора цвета
- `static JDialog createDialog(Component parent, String title, boolean modal, JColorChooser chooser, ActionListener okListener, ActionListener cancelListener)`
Создает диалоговое окно, содержащее компонент для выбора цвета.
Параметры:

<code>parent</code>	Компонент, в котором отображается диалоговое окно
<code>title</code>	Заголовок фрейма с диалоговым окном
<code>modal</code>	Принимает логическое значение true, если выполнение программы должно быть приостановлено до тех пор, пока не закроется диалоговое окно
<code>chooser</code>	Компонент для выбора цвета, вводимый в диалоговое окно
<code>okListener</code>	Приемник событий от кнопки ОК
<code>cancelListener</code>	Приемник событий от кнопки Cancel

На этом рассмотрение компонентов ГПИ завершается. Материал глав 7–9 поможет вам самостоятельно создавать простые ГПИ средствами библиотеки Swing. А более совершенные компоненты Swing и усовершенствованные методики работы с графикой обсуждаются во втором томе данной книги.

Обработка событий

В этой главе...

- ▶ Общее представление об обработке событий
- ▶ Действия
- ▶ События от мыши
- ▶ Иерархия событий в библиотеке AWT

Обработка событий имеет огромное значение для создания программ с ГПИ. Чтобы построить ГПИ на высоком профессиональном уровне, нужно очень хорошо разбираться в механизме обработки событий в Java. В этой главе поясняется, каким образом действует модель событий Java AWT. Из нее вы узнаете, как перехватывать события от компонентов пользовательского интерфейса и устройств ввода. В ней будет также показано, как обращаться с *действиями*, предлагающими более структурированный подход к обработке событий действия.

Общее представление об обработке событий

В каждой операционной среде, поддерживающей ГПИ, непрерывно отслеживаются такие события, как нажатие клавиш или щелчки кнопками мыши, а затем о них сообщается исполняемой программе, которая сама решает, как реагировать на эти события. В языках программирования, подобных Visual Basic, соответствие между событиями и кодом очевидно: некто пишет код, реагирующий на каждое интересное событие, и вводит его в так называемую *процедуру обработки событий*. Например, после щелчка на кнопке `HelpButton` в Visual Basic будет вызвана процедура обработки событий типа `HelpButton_Click`. Каждому компоненту пользовательского интерфейса в Visual Basic соответствует фиксированное множество событий, состав которого изменить нельзя.

С другой стороны, если программа, реагирующая на события, написана на языке, подобном С, то ее автор должен написать также код, непрерывно проверяющий очередь событий, которая предоставляется операционной системой. (Обычно для этого код обработки событий размещается в гигантском цикле с большим количеством операторов switch.) Очевидно, что такая технология довольно неуклюжа, да и создавать подобные программы намного труднее. Но у такого подхода имеется следующее преимущество: множество событий ничем не ограничено, в отличие от Visual Basic, где очередь событий скрыта от программиста.

Подход, применяемый в Java, представляет собой нечто среднее между подходами, принятыми в Visual Basic и С. Он довольно сложен. При обработке событий, предусмотренных в библиотеке AWT, программист полностью контролирует передачу событий от *источников* (например, кнопок или полос прокрутки) к *приемникам событий*. Назначить приемником событий можно *любой* объект. На практике для этого указывается такой объект, который может надлежащим образом отреагировать на событие. Такая *модель делегирования событий* позволяет достичь большей гибкости по сравнению с Visual Basic, где приемники событий предопределены.

У источников событий имеются методы, позволяющие связывать их с приемниками событий. Когда наступает соответствующее событие, источник извещает о нем все зарегистрированные приемники. Как и следовало ожидать от такого объектно-ориентированного языка, как Java, сведения о событии инкапсулируются в *объекте события*. В Java все события описываются подклассами, производными от класса `java.util.EventObject`. Разумеется, существуют подклассы и для каждого вида событий, например `ActionEvent` и `WindowEvent`.

Различные источники могут порождать разные виды событий. Например, кнопка может посылать объекты типа `ActionEvent`, а окно — объекты типа `WindowEvent`. Кратко механизм обработки событий в библиотеке AWT можно описать следующим образом.

- Объект приемника событий — это экземпляр класса, реализующего специальный интерфейс, называемый (естественно) *интерфейсом приемника событий*.
- Источник событий — это объект, который может регистрировать приемники событий и посылать им объекты событий.
- При наступлении события источник посылает объекты событий всем зарегистрированным приемникам.
- Приемники используют данные, инкапсулированные в объекте события, чтобы решить, как реагировать на это событие.

На рис. 8.1 схематически показаны отношения между классами обработки событий и интерфейсами. В приведенном ниже примере кода показано, каким образом указывается приемник событий.

```
ActionListener listener = . . . ;  
JButton button = new JButton("Ok");  
button.addActionListener(listener);
```

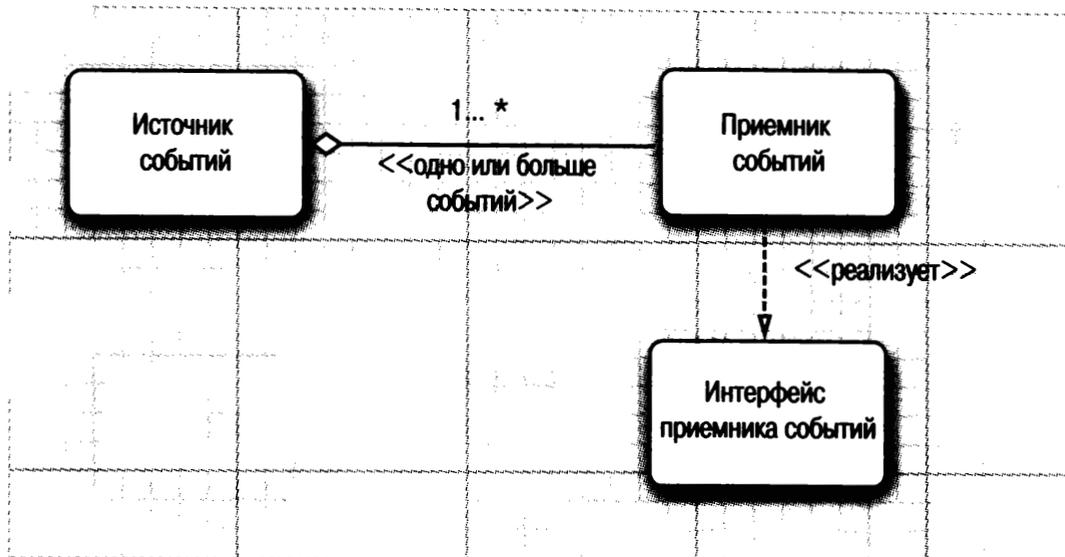


Рис. 8.1. Отношения между классами обработки событий и интерфейсами

Теперь объект `listener` оповещается о наступлении “события действия” в кнопке. Для кнопки, как и следовало ожидать, таким событием действия является щелчок на ней кнопкой мыши. Для того чтобы реализовать интерфейс `ActionListener`, в классе приемника событий должен присутствовать метод `actionPerformed()`, принимающий в качестве параметра объект типа `ActionEvent`, как показано ниже.

```

class MyListener implements ActionListener
{
    . . .
    public void actionPerformed(ActionEvent event)
    {
        // здесь следует код, реагирующий на щелчок на кнопке
        . . .
    }
}
  
```

Когда пользователь щелкает на кнопке, объект типа `JButton` создает объект типа `ActionEvent` и вызывает метод `listener.actionPerformed(event)`, передавая ему объект события. У источника событий может быть несколько приемников. В этом случае после щелчка на кнопке объект типа `JButton` вызовет метод `actionPerformed()` для всех зарегистрированных приемников событий. На рис. 8.2 схематически показано взаимодействие источника, приемника и объекта события.

Пример обработки событий от щелчков на кнопке

Для того чтобы стал понятнее принцип действия модели делегирования событий, рассмотрим подробно, каким образом обрабатываются события от щелчков на кнопке. Для этого потребуются три кнопки, расположенные на панели, а также три объекта приемников событий, добавляемые к кнопкам в качестве приемников действий над ними.

Всякий раз, когда пользователь щелкает кнопкой мыши на какой-нибудь кнопке, находящейся на панели, соответствующий приемник получает объект типа `ActionEvent`, указывающий на факт щелчка. В рассматриваемом здесь примере программы объект приемника событий, реагируя на щелчок, будет изменять цвет фона панели.

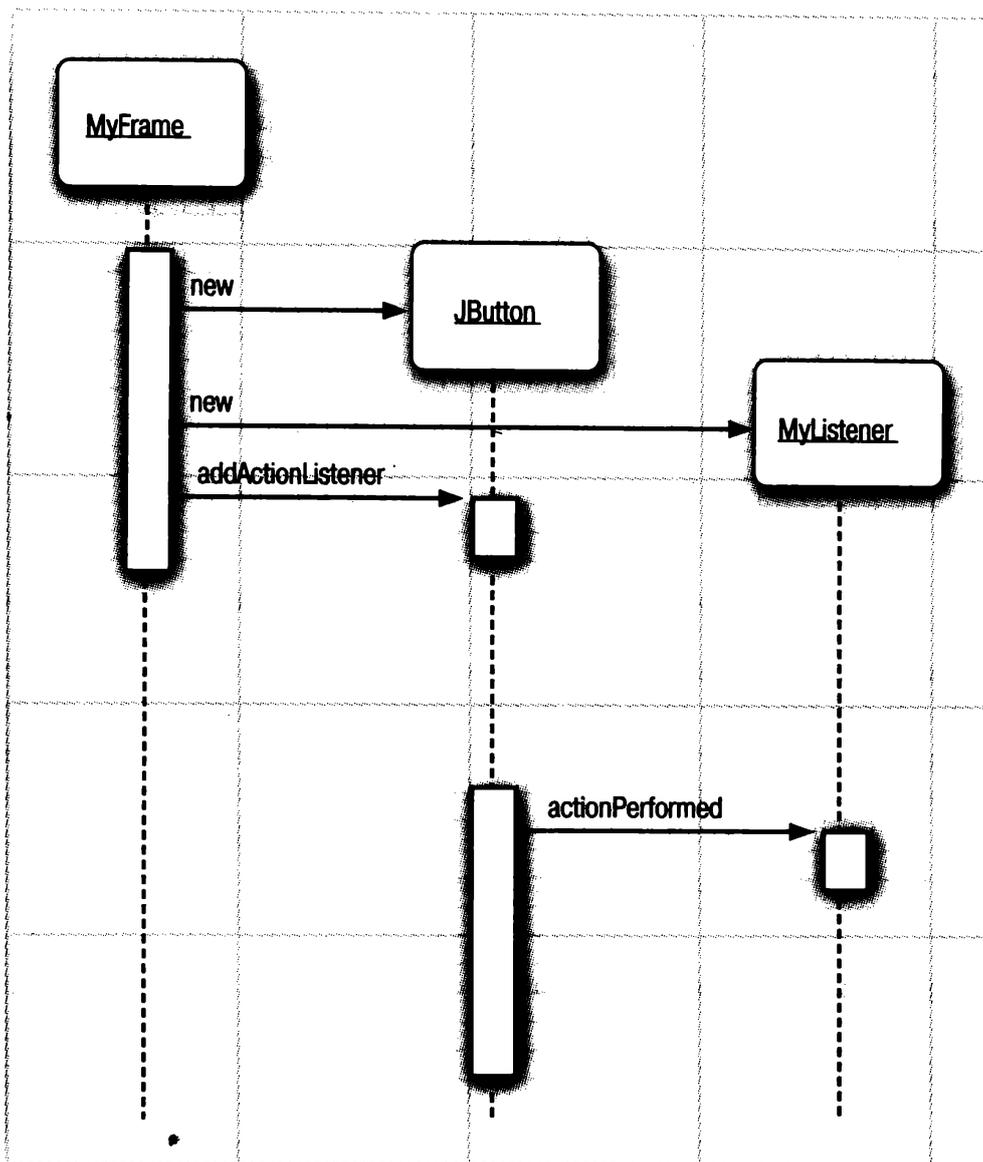


Рис. 8.2. Уведомление о событии

Но прежде чем продемонстрировать программу, реагирующую на щелчки на выбираемых кнопках, необходимо пояснить, каким образом кнопки создаются и вводятся на панели. (Подробнее об элементах ГПИ речь пойдет в главе 9.) Для того чтобы создать кнопку, в конструкторе ее класса нужно сначала задать символьную строку метки, пиктограмму или оба атрибута вместе. Ниже приведены два примера создания кнопок.

```

JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));

```

Затем вызывается метод `add()`, чтобы добавить кнопки на панели:

```

JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton("Blue");
JButton redButton = new JButton("Red");

```

```

buttonPanel.add(yellowButton);
buttonPanel.add(blueButton);
buttonPanel.add(redButton);

```

Результат выполнения этого фрагмента кода приведен на рис. 8.3.

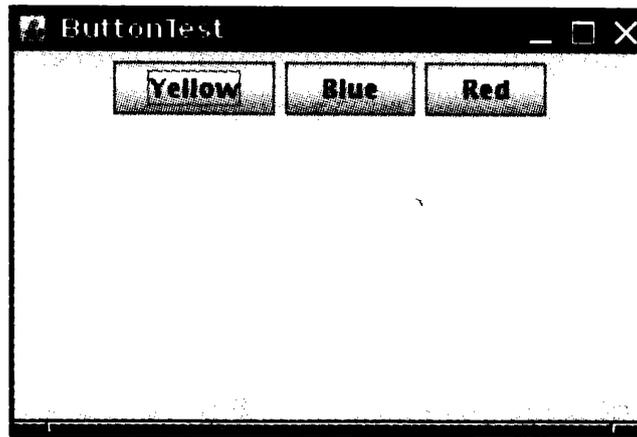


Рис. 8.3. Панель, заполненная кнопками

Далее нужно ввести код, позволяющий реагировать на эти кнопки. Для этого требуется класс, реализующий интерфейс `ActionListener`, где, как упоминалось выше, объявлен единственный метод `actionPerformed()`. Сигнатура этого метода выглядит следующим образом:

```
public void actionPerformed(ActionEvent event)
```



НА ЗАМЕТКУ! Интерфейс `ActionListener`, применяемый в данном примере, не ограничивается отслеживанием щелчков на кнопках. Он применяется и во многих других случаях, когда наступают перечисленные ниже события.

- После двойного щелчка на элементе списка.
- После выбора пункта меню.
- После нажатия клавиши `<Enter>`, когда курсор находится в текстовом поле.
- По истечении периода времени, заданного для компонента `Timer`.

Обработка этих событий более подробно рассматривается далее в этой и следующей главах. Но во всех случаях интерфейс `ActionListener` используется совершенно одинаково: метод `actionPerformed()` — единственный в интерфейсе `ActionListener` — принимает в качестве параметра объект типа `ActionEvent`. Этот объект несет в себе сведения о наступившем событии.

Допустим, что после щелчка на кнопке требуется изменить цвет фона панели. Новый цвет указывается в классе приемника событий следующим образом:

```
class ColorAction implements ActionListener
{
    private Color backgroundColor;

    public ColorAction(Color c)
    {
        backgroundColor = c;
    }

    public void actionPerformed(ActionEvent event)
    {
        // установить цвет фона панели
        . . .
    }
}
```

Затем для каждого цвета создается один объект. Все эти объекты устанавливаются в качестве приемников событий от соответствующих кнопок:

```
ColorAction yellowAction = new ColorAction(Color.YELLOW);
ColorAction blueAction = new ColorAction(Color.BLUE);
ColorAction redAction = new ColorAction(Color.RED);

yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
```

Так, если пользователь щелкнет на кнопке с меткой Yellow, вызывается метод `actionPerformed()` из объекта `yellowAction`. В поле экземпляра `backgroundColor` хранится значение `Color.YELLOW`, поэтому выполняющийся метод может установить требуемый (в данном случае желтый) цвет фона панели.

Осталось разрешить еще одно небольшое затруднение. Объект типа `ColorAction` не имеет доступа к переменной `buttonPanel`. Это затруднение можно разрешить разными путями. В частности, переменную `buttonPanel` можно указать в конструкторе класса `ColorAction`. Но удобнее сделать класс `ColorAction` внутренним по отношению к классу `ButtonFrame`. В этом случае его методы получают доступ к внешним переменным автоматически. (Более подробно о внутренних классах см. в главе 6.) Последуем по второму пути. Ниже показано, каким образом класс `ColorAction` включается в состав класса `ButtonFrame`.

```
class ButtonFrame extends JFrame
{
    private JPanel buttonPanel;
    . . .
    private class ColorAction implements ActionListener
    {
        private Color backgroundColor;
        . . .
        public void actionPerformed(ActionEvent event)
        {
            buttonPanel.setBackground(backgroundColor);
        }
    }
}
```

Проанализируем исходный код метода `actionPerformed()` более подробно. Оказывается, что поле `buttonPanel` во внутреннем классе `ColorAction` отсутствует, но имеется во внешнем классе `ButtonFrame`. Такая ситуация встречается довольно часто. Объекты приемников событий должны выполнять определенные действия, оказывающие влияние на другие объекты. И зачастую бывает очень удобно включить класс приемника событий заранее в состав другого класса, состояние которого должен видоизменить этот приемник.

В листинге 8.1 приведен весь исходный код класса фрейма, реализующего обработку событий от кнопок. Как только пользователь щелкнет на какой-нибудь кнопке, соответствующий приемник событий изменит цвет фона панели.

Листинг 8.1. Исходный код из файла `button/ButtonFrame.java`

```
1 package button;
2
3 import java.awt.*;
```

```
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Фрейм с панелью кнопок
9  */
10 public class ButtonFrame extends JFrame
11 {
12     private JPanel buttonPanel;
13     private static final int DEFAULT_WIDTH = 300;
14     private static final int DEFAULT_HEIGHT = 200;
15
16     public ButtonFrame()
17     {
18         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
19
20         // создать кнопки
21         JButton yellowButton = new JButton("Yellow");
22         JButton blueButton = new JButton("Blue");
23         JButton redButton = new JButton("Red");
24
25         buttonPanel = new JPanel();
26
27         // ввести кнопки на панели
28         buttonPanel.add(yellowButton);
29         buttonPanel.add(blueButton);
30         buttonPanel.add(redButton);
31
32         // ввести панель во фрейм
33         add(buttonPanel);
34
35         // сформировать действия кнопок
36         ColorAction yellowAction = new ColorAction(Color.YELLOW);
37         ColorAction blueAction = new ColorAction(Color.BLUE);
38         ColorAction redAction = new ColorAction(Color.RED);
39
40         // связать действия с кнопками
41         yellowButton.addActionListener(yellowAction);
42         blueButton.addActionListener(blueAction);
43         redButton.addActionListener(redAction);
44     }
45
46     /**
47      * Приемник действий, устанавливающий цвет фона панели .
48      */
49     private class ColorAction implements ActionListener
50     {
51         private Color backgroundColor;
52
53         public ColorAction(Color c)
54         {
55             backgroundColor = c;
56         }
57
58         public void actionPerformed(ActionEvent event)
59         {
```

```
60     buttonPanel.setBackground(backgroundColor);
61     }
62 }
63 }
```

javax.swing.JButton 1.2

- JButton(String label)
- JButton(Icon icon)
- JButton(String label, Icon icon)

Создают кнопку. Символьная строка, передаваемая в качестве параметра, может содержать текст, а начиная с версии Java SE 1.3 — HTML-разметку, например `<HTML>ОК</HTML>`.

java.awt.Container 1.0

- Component add(Component c)
Добавляет компонент *c* в контейнер.

Овладение внутренними классами

Некоторым не нравятся внутренние классы, потому что им кажется, что такое пространство классов и объектов замедляет выполнение программы. Попробуем разобраться в этом. Для каждого компонента пользовательского интерфейса совсем не обязательно создавать новый класс. В рассматриваемом здесь примере все три кнопки делят между собой один и тот же класс приемника событий. Разумеется, каждой из них соответствует отдельный объект этого класса. Но эти объекты невелики. Они содержат обозначение цвета и ссылку на панель. И в традиционном решении с операторами `if/else` делается ссылка на те же самые объекты цветов, которые хранятся в приемниках действий, как на локальные переменные, а не поля экземпляра.

Эта ситуация служит удачным примером того, как анонимные внутренние классы упрощают исходный код программы. Вернемся к листингу 8.1. Обратите внимание на то, что обработка событий от всех кнопок происходит совершенно одинаково и состоит из следующих задач.

1. Создание кнопки с соответствующей меткой.
2. Размещение кнопки на панели.
3. Создание приемника действий, изменяющего цвет фона панели.
4. Добавление приемника действий.

Чтобы упростить решение всех этих задач, сначала реализуется следующий вспомогательный метод:

```
public void makeButton(String name, Color backgroundColor)
{
    JButton button = new JButton(name);
```

```

buttonPanel.add(button);
ColorAction action = new ColorAction(background-color);
button.addActionListener(action);
}

```

А затем он вызывается следующим образом:

```

makeButton("yellow", Color.YELLOW);
makeButton("blue", Color.BLUE);
makeButton("red", Color.RED);

```

Но возможности упрощения программы этим не исчерпываются. Следует заметить, что класс `ColorAction` требуется лишь один раз: в методе `makeButton()`. Следовательно, его можно сделать анонимным, как показано ниже.

```

public void makeButton(String name, final Color backgroundColor)
{
    JButton button = new JButton(name);
    buttonPanel.add(button);
    button.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            buttonPanel.setBackground(backgroundColor);
        }
    });
}

```

Код приемника действий стал еще проще. Метод `actionPerformed()` теперь просто обращается к параметру `backgroundColor`. (Как и все локальные переменные, доступные во внутреннем классе, этот параметр должен быть конечным.) Явный конструктор теперь не нужен. Как упоминалось в главе 6, механизм внутренних классов автоматически генерирует конструктор, размещающий в памяти все локальные конечные переменные, используемые в одном из методов этого класса.



СОВЕТ. На первый взгляд, внутренние классы заметно усложняют исходный код программы. Но если вы научитесь выделять главное, абстрагируясь на время от остального кода, то легко освоите синтаксис внутренних классов, как в приведенном ниже примере.

```

button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        buttonPanel.setBackground(backgroundColor);
    }
});

```

Таким образом, действие кнопки заключается в установке цвета фона. Если обработчик событий состоит всего лишь из нескольких операторов, в его исходном коде нетрудно разобраться, особенно тем, кто не боится применять механизм внутренних классов.



НА ЗАМЕТКУ! Ничто не мешает назначить любой объект класса, реализующего интерфейс `ActionListener`, в качестве приемника событий от кнопки. Мы предпочитаем использовать для этой цели объекты нового класса, которые специально создаются для выполнения нужных действий кнопки. Но некоторым программистам неудобно пользоваться внутренними классами, поэтому они выбирают другой путь, создавая контейнер источников событий, реализующий интерфейс `ActionListener`. Такой контейнер устанавливает сам себя в качестве приемника примерно так:

```
yellowButton.addActionListener(this);
blueButton.addActionListener(this);
redButton.addActionListener(this);
```

Теперь у трех кнопок нет отдельных приемников событий. Они разделяют единственный объект приемника — фрейм кнопок. Следовательно, в его методе `actionPerformed()` должно быть определено, какая именно кнопка нажата:

```
class ButtonFrame extends JFrame implements ActionListener
{
    . . .
    public void actionPerformed(ActionEvent event)
    {
        Object source = event.getSource();
        if (source == yellowButton) . . .
        else if (source == blueButton) . . .
        else if (source == redButton) . . .
        else . . .
    }
}
```

Как видите, код становится запутанным, и потому такой подход не рекомендуется.

java.util.EventObject 1.1

- `Object getSource()`
Возвращает ссылку на объект, являющийся источником события.

java.awt.event.ActionEvent 1.1

- `String getActionCommand()`
Возвращает командную строку, связанную с событием действия. Если событие наступило от кнопки, в командной строке содержится метка кнопки, при условии, что она не была изменена методом `setActionCommand()`.

Создание приемников событий с единственным вызовом метода

Простые приемники событий можно указывать и без программирования внутренних классов. Допустим, имеется кнопка с меткой `Load` (Загрузить), обработчик событий которой содержит следующий единственный вызов метода:

```
frame.loadData();
```

Разумеется, для создания такого обработчика событий можно воспользоваться анонимным вложенным классом следующим образом:

```
loadButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        frame.loadData();
    }
})
```

Но класс `EventHandler` может создать такой обработчик автоматически в следующем вызове:

```
EventHandler.create(ActionListener.class, frame, "loadData")
```

Конечно, обработчик все равно придется установить, как показано ниже.

```
loadButton.addActionListener(  
    EventHandler.create(ActionListener.class, frame, "loadData"));
```

Если в обработчике вызывается метод с одним параметром, который может быть получен из параметра события, то для создания метода `create()` можно воспользоваться другой формой. Например, следующий вызов:

```
EventHandler.create(ActionListener.class, frame, "loadData", "source.text")
```

равнозначен такому коду:

```
new ActionListener()  
{  
    public void actionPerformed(ActionEvent event)  
    {  
        frame.loadData(((JTextField) event.getSource()).getText());  
    }  
}
```

Имена свойств `source` и `text` превращаются в вызовы методов `getSource()` и `getText()`.



ВНИМАНИЕ! Второй аргумент в вызове метода `EventHandler.create()` должен принадлежать открытому классу. В противном случае механизм рефлексии окажется не в состоянии обнаружить и вызвать целевой метод.

java.beans.EventHandler 1.4

- `static Object create(Class listenerInterface, Object target, String action)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty, String listenerMethod)`

Создают прокси-объект, реализующий заданный интерфейс. В результате указанный метод или все методы интерфейса выполняют заданное действие над целевым объектом.

Действие может быть обозначено именем метода или свойством объекта. Если это свойство, то выполняется метод установки. Так, если параметр `action` имеет значение `text`, действие превращается в вызов метода `setText()`.

Свойство события состоит из одного или нескольких имен свойств, разделенных точками. Первое свойство задает считывание параметра метода из обработчика, второе свойство — считывание результирующего объекта и т.д. Например, свойство `source.text` превращается в вызовы методов `getSource()` и `getText()`.

Пример изменения визуального стиля

По умолчанию в Swing-программах применяется визуальный стиль Metal. Изменить этот стиль можно двумя способами. В подкаталоге `jre/lib` можно предусмотреть файл `swing.properties` и задать в нем с помощью свойства `swing.defaultlaf` имя класса, определяющего нужный стиль:

```
wing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

Следует заметить, что визуальный стиль Metal находится в пакете `javax.swing`, а другие стили находятся в пакете `com.sun.java`. Они не обязательно должны присутствовать в каждой установке Java. В настоящее время по соображениям, связанным с соблюдением авторского права, пакеты визуальных стилей для операционных систем Windows и Mac OS поставляются только с исполняющими средами, ориентированными на эти системы.



СОВЕТ. Строки, начинающиеся со знака `#`, в файлах, описывающих свойства визуальных стилей, игнорируются. Это дает возможность предусмотреть в файле `swing.properties` несколько стилей, закомментировав знаком `#` ненужные следующим образом:

```
#swing.defaultlaf=javax.swing.plaf.metal.MetalLookAndFeel
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
#swing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

Чтобы изменить визуальный стиль пользовательского интерфейса Swing-программы, ее нужно запустить заново. Программа прочитает содержимое файла `swing.properties` лишь один раз при запуске.

Второй способ позволяет изменить визуальный стиль динамически. Для этого вызывается метод `UIManager.setLookAndFeel()`, для которого указывается имя класса требуемого стиля. Затем вызывается статический метод `SwingUtilities.updateComponentTreeUI()`, с помощью которого обновляется весь набор компонентов. Этому методу достаточно передать один компонент, а остальные он найдет автоматически. Если нужный стиль не будет обнаружен или произойдет ошибка при его загрузке, в методе `UIManager.setLookAndFeel()` могут быть сгенерированы различные исключения. Мы не будем пока что уделять особого внимания обработке исключений, поскольку эта тема будет полностью освещена в главе 11.

Ниже приведен пример кода, в котором задается визуальный стиль Motif для пользовательского интерфейса программы.

```
String plaf = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
try
{
    UIManager.setLookAndFeel(plaf);
    SwingUtilities.updateComponentTreeUI(panel);
}
catch(Exception e) { e.printStackTrace(); }
```

Для того чтобы пронумеровать установленные реализации визуальных стилей, нужно сделать следующий вызов:

```
UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
```

После этого можно получить имя каждого стиля и реализующего его класса, используя приведенный ниже код.

```
String name = infos[i].getName();  
String className = infos[i].getClassName();
```

В листинге 8.2 приведен исходный код завершенной программы, демонстрирующей смену визуальных стилей (рис. 8.4). Она очень похожа на программу из листинга 8.1. В соответствии с упомянутыми выше рекомендациями в этой программе применены вспомогательный метод `makeButton()` и анонимный внутренний класс, чтобы задать действия над кнопками, а именно смену визуальных стилей.

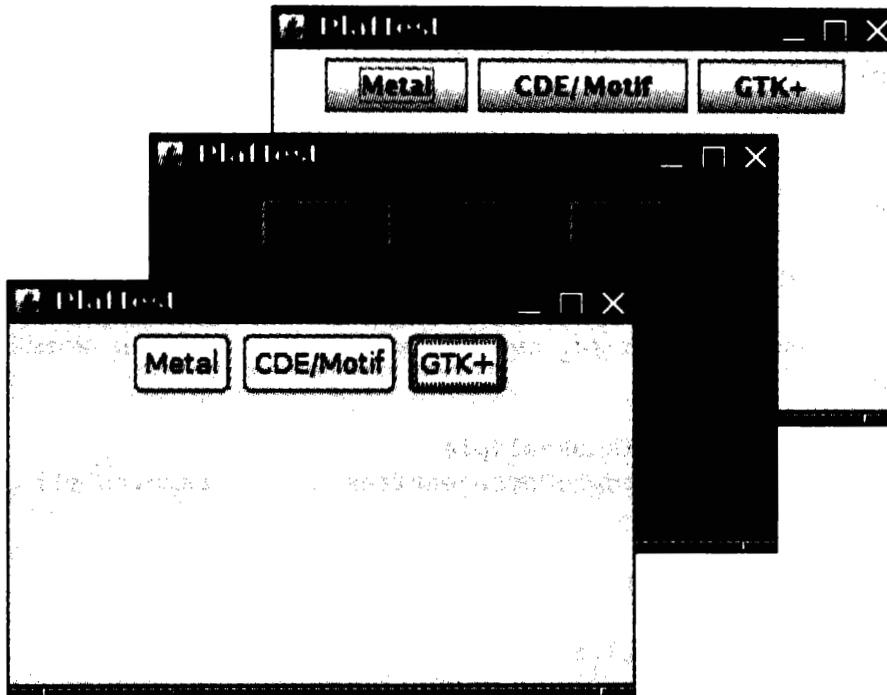


Рис. 8.4. Смена визуальных стилей

Листинг 8.2. Исходный код из файла `plaf/PlafFrame.java`

```
1 package plaf;  
2  
3 import java.awt.event.*;  
4 import javax.swing.*;  
5  
6 /**  
7  * Фрейм с панелью кнопок для смены визуального стиля  
8  */  
9 public class PlafFrame extends JFrame  
10 {  
11     private JPanel buttonPanel;  
12     public PlafFrame()  
13     {  
14         buttonPanel = new JPanel();  
15  
16         UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();  
17         for (UIManager.LookAndFeelInfo info : infos)  
18             makeButton(info.getName(), info.getClassName());  
19  
20         add(buttonPanel);  
21         pack();  
22     }  
}
```

```
23
24 /**
25  * Изменяет подключаемый стиль после щелчка на кнопке
26  * @param name Имя кнопки
27  * @param plafName Имя класса визуального стиля
28  */
29 void makeButton(String name, final String plafName)
30 {
31     //, ввести кнопку на панели
32
33     JButton button = new JButton(name);
34     buttonPanel.add(button);
35
36     // установить действие для кнопки
37
38     button.addActionListener(new ActionListener()
39     {
40         public void actionPerformed(ActionEvent event)
41         {
42             // действие для кнопки: сменить визуальный стиль на новый
43             try
44             {
45                 UIManager.setLookAndFeel(plafName);
46                 SwingUtilities.updateComponentTreeUI(PlafFrame.this);
47                 pack();
48             }
49             catch (Exception e)
50             {
51                 e.printStackTrace();
52             }
53         }
54     });
55 }
56 }
```

У этой программы имеется одно заметное преимущество. Метод `actionPerformed()` во внутреннем классе приемника действий должен передать методу `updateComponentTreeUI()` ссылку `this` на объект внешнего класса `PlafPanel`. Как пояснялось в главе 6, ссылка `this` на объект внешнего класса должна быть снабжена префиксом с именем этого класса:

```
SwingUtilities.updateComponentTreeUI(PlafPanel.this);
```

javax.swing.UIManager 1.2

- `static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()`
Получает массив объектов, описывающих реализации установленных визуальных стилей.
- `static setLookAndFeel(String className)`
Устанавливает текущий стиль, используя заданное имя класса (например, "javax.swing.plaf.metal.MetalLookAndFeel").

javax.swing.UIManager.LookAndFeelInfo 1.2

- String getName()
Возвращает имя стиля.
- String getClassName()
Возвращает имя класса, реализующего стиль.

Классы адаптеров

Не все события обрабатываются так же просто, как и события от кнопок. Если речь идет о профессионально разработанной прикладной программе, она должна непременно завершаться корректно, а ее пользователю должно быть гарантировано, что результаты выполненной им работы не будут утрачены. Так, если пользователь закрыл фрейм, можно вывести на экран диалоговое окно и предупредить его о необходимости сохранить результаты своей работы и только после его согласия завершить выполнение прикладной программы.

Когда пользователь пытается закрыть фрейм, объект типа `JFrame` становится источником события типа `WindowEvent`. Для того чтобы перехватить это событие, требуется соответствующий объект приемника событий, который следует добавить в список приемников событий в окне следующим образом:

```
WindowListener listener = . . . ;  
frame.addWindowListener(listener);
```

Приемник событий должен быть экземпляром класса, реализующего интерфейс `WindowListener`. В интерфейсе `WindowListener` имеется семь методов. Фрейм вызывает их в ответ на семь разных событий, которые могут произойти в окне. Их имена отражают их назначение. Исключением может быть лишь слово `iconified`, которое в Windows означает “свернутое” окно. Ниже показано, как выглядит весь интерфейс `WindowListener`.

```
public interface WindowListener  
{  
    void windowOpened(WindowEvent e);  
    void windowClosing(WindowEvent e);  
    void windowClosed(WindowEvent e);  
    void windowIconified(WindowEvent e);  
    void windowDeiconified(WindowEvent e);  
    void windowActivated(WindowEvent e);  
    void windowDeactivated(WindowEvent e);  
}
```



НА ЗАМЕТКУ! Если требуется проверить, было ли окно увеличено до максимальных размеров, следует установить класс `WindowStateListener`. Методы этого класса вкратце описываются далее в выдержке из документации на прикладной интерфейс API.

Как обычно, любой класс, реализующий какой-нибудь интерфейс, должен реализовать все его методы. В данном случае это означает создание тела *семи* методов. Но ведь нас интересует только один из них — метод `windowClosing()`. Разумеется, можно определить класс, реализующий интерфейс `WindowListener`, введя в тело его

метода `windowClosing()` вызов `System.exit(0)`, а тела остальных шести методов оставить пустыми, как показано ниже.

```
class Terminator implements WindowListener
{
    public void windowClosing(WindowEvent e)
    {
        if (пользователь согласен)
            System.exit(0);
    }

    public void windowOpened(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

Создавать код для шести методов, которые ничего не делают, — неблагоприятное занятие. Чтобы упростить эту задачу, для каждого из интерфейсов приемников событий в AWT, где имеется несколько методов, создается класс *адаптера*, реализующий все эти методы, причем тела их пусты. Например, класс `WindowAdapter` содержит семь методов, не выполняющих никаких операций. Следовательно, класс адаптера автоматически удовлетворяет требованиям, предъявляемым к реализации соответствующего интерфейса. Класс адаптера можно расширить и уточнить нужные виды реакции на некоторые, но не на все виды событий в интерфейсе. (Обратите внимание на то, что интерфейс `ActionListener` содержит только один метод, поэтому для него класс адаптера не нужен.)

В качестве примера рассмотрим применение адаптера окна. Класс `WindowAdapter` можно расширить, унаследовав шесть пустых методов и переопределив метод `windowClosing()` следующим образом:

```
class Terminator extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        if (пользователь согласен)
            System.exit(0);
    }
}
```

Теперь объект класса `Terminator` можно зарегистрировать в качестве приемника событий, как показано ниже.

```
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

Как только во фрейме будет сгенерировано событие, оно будет передано приемнику событий (объекту `listener`) при вызове одного из его семи методов (рис. 8.5). Шесть из этих методов ничего не делают, а метод `windowClosing()` выполняет вызов `System.exit(0)`, завершая работу прикладной программы.

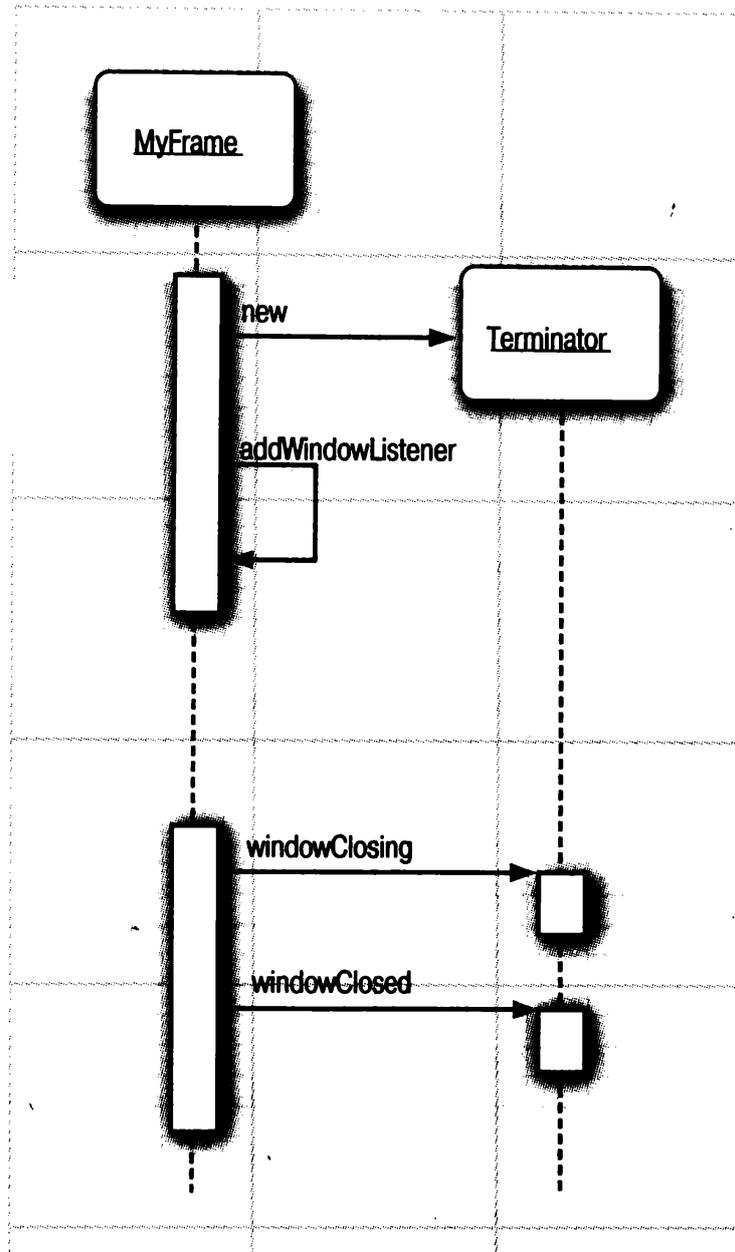


Рис. 8.5. Обработка событий в окне



ВНИМАНИЕ! Если, расширяя класс адаптера, вы неправильно укажете имя метода, компилятор не сообщит об ошибке. Так, если вы создадите метод `windowIsClosing()` в подклассе, производном от класса `WindowAdapter`, то получите новый класс, содержащий восемь методов, причем метод `windowClosing()` не будет выполнять никаких действий.

Создание класса приемника событий, расширяющего класс `WindowAdapter`, — несомненный шаг вперед, но можно достичь еще большего. Объекту приемника событий не обязательно присваивать имя, достаточно написать следующий оператор:

```
frame.addWindowListener(new Terminator());
```

Но и это еще не все! В качестве приемника событий можно использовать анонимный внутренний класс фрейма следующим образом:

```
frame.addWindowListener(new
    WindowAdapter()
{
    public void windowClosing(WindowEvent e)
```

```
{
    if (пользователь согласен)
        System.exit(0);
}
});
```

В этом фрагменте кода выполняются следующие действия.

- Определяется анонимный класс, не имеющий имени и расширяющий класс `WindowAdapter`.
- Переопределяется метод `windowClosing()` в этом анонимном классе. Как и прежде, данный метод отвечает за выход из программы.
- От класса `WindowAdapter` наследуются остальные шесть методов, не выполняющих никаких действий.
- Создается объект класса. Этот объект также анонимный и не имеет имени.
- Методу `addWindowListener()` передается анонимный объект.

Следует еще раз подчеркнуть, что к употреблению в коде анонимных внутренних классов нужно привыкнуть. А наградой за настойчивость станет предельная краткость полученного в итоге кода.

java.awt.event.WindowListener 1.1

- `void windowOpened(WindowEvent e)`
Вызывается после открытия окна.
- `void windowClosing(WindowEvent e)`
Вызывается, когда пользователь выдает диспетчеру окон команду закрыть окно. Следует иметь в виду, что окно закроется только в том случае, если для него будет вызван метод `hide()` или `dispose()`.
- `void windowClosed(WindowEvent e)`
Вызывается после закрытия окна.
- `void windowIconified(WindowEvent e)`
Вызывается после свертывания окна.
- `void windowDeiconified(WindowEvent e)`
Вызывается после развертывания окна.
- `void windowActivated(WindowEvent e)`
Вызывается после активизации окна. Активным может быть только фрейм или диалоговое окно. Обычно диспетчер окон специально выделяет активное окно, например, подсвечивает его заголовок.
- `void windowDeactivated(WindowEvent e)`
Вызывается после того, как окно становится неактивным.

java.awt.event.WindowStateListener 1.4

- void windowStateChanged(WindowEvent event)

Вызывается после того, как окно было полностью развернуто, свернуто или восстановлено до нормальных размеров.

java.awt.event.WindowEvent 1.1

- int getNewState() 1.4
- int getOldState() 1.4

Возвращают новое или старое состояние окна при наступлении события, связанного с изменением его состояния. Возвращаемое целое значение может быть одним из следующих:

```
Frame.NORMAL  
Frame.ICONIFIED  
Frame.MAXIMIZED_HORIZ  
Frame.MAXIMIZED_VERT  
Frame.MAXIMIZED_BOTH
```

Действия

Одну и ту же команду можно выполнить разными способами. В частности, пользователь может выбрать пункт меню, нажать соответствующую клавишу или щелкнуть на кнопке непосредственно на панели инструментов. В этом случае очень удобна рассматриваемая здесь модель делегирования событий в библиотеке AWT: достаточно связать все эти события с одним и тем же приемником. Допустим, `blueAction` — это приемник действий, в методе `actionPerformed()` которого цвет фона изменяется на синий. Один и тот же объект можно связать с разными источниками событий, перечисленными ниже.

- Кнопка Blue (Синий) на панели инструментов.
- Пункт меню Blue.
- Нажатие комбинации клавиш <Ctrl+B>.

Команда, изменяющая цвет фона, выполняется одинаково, независимо от того, что именно привело к ее выполнению — щелчок на кнопке, выбор пункта меню или нажатие комбинации клавиш. В библиотеке Swing предусмотрен очень полезный механизм, позволяющий инкапсулировать команды и связывать их с несколькими источниками событий. Этим механизмом служит интерфейс `Action`. *Действие* представляет собой объект, инкапсулирующий следующее.

- Описание команды (в виде текстовой строки или пиктограммы).
- Параметры, необходимые для выполнения команды (в данном случае для выбора нужного цвета).

Интерфейс Action содержит следующие методы:

```
void actionPerformed(ActionEvent event)
void setEnabled(boolean b)
boolean isEnabled()
void putValue(String key, Object value)
Object getValue(String key)
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

Первый метод похож на аналогичный метод из интерфейса ActionListener. На самом деле интерфейс Action расширяет интерфейс ActionListener. Следовательно, вместо объекта класса, реализующего интерфейс ActionListener, можно использовать объект класса, реализующего интерфейс Action.

Следующие два метода позволяют активизировать или запретить действие, а также проверить, активизировано ли указанное действие в настоящий момент. Если пункт меню или кнопка на панели инструментов связаны с запрещенным действием, они выделяются светло-серым цветом как недоступные.

Методы putValue() и getValue() позволяют записывать и извлекать из памяти произвольные пары "имя-значение" из объектов действий класса, реализующего интерфейс Action. Так, в паре предопределенных строк Action.NAME и Action.SMALL_ICON имена и пиктограммы действий в объекте действия сохраняются следующим образом:

```
action.putValue(Action.NAME, "Blue");
action.putValue(Action.SMALL_ICON, new ImageIcon("blue-ball.gif"));
```

В табл. 8.1 перечислены предопределенные имена действий.

Таблица 8.1. Предопределенные имена действий

Имя	Значение
NAME	Имя действия. Отображается на кнопке и в названии пункта меню
SMALL_ICON	Место для хранения пиктограммы, которая отображается на кнопке, в пункте меню или на панели инструментов
SHORT_DESCRIPTION	Краткое описание пиктограммы, отображаемое во всплывающей подсказке
LONG_DESCRIPTION	Подробное описание пиктограммы. Может использоваться для подсказки. Не применяется ни в одном из компонентов библиотеки Swing
MNEMONIC_KEY	Мнемоническое сокращение. Отображается в пункте меню (см. главу 9)
ACCELERATOR_KEY	Место для хранения комбинации клавиш. Не применяется ни в одном из компонентов библиотеки Swing
ACTION_COMMAND_KEY	Применялось раньше в устаревшем теперь методе registeredKeyboardAction()
DEFAULT	Может быть полезным для хранения разнообразных объектов. Не применяется ни в одном из компонентов библиотеки Swing

Если в меню или на панели инструментов добавляется какое-то действие, его имя и пиктограмма автоматически извлекаются из памяти и отображаются в меню и на панели. Значение SHORT_DESCRIPTION выводится во всплывающей подсказке.

Последние два метода из интерфейса Action позволяют уведомить другие объекты, в частности, меню и панели инструментов, об изменении свойств объекта действия. Так, если меню введено в качестве приемника для изменений свойств в объекте действия и это действие впоследствии было запрещено, то при отображении меню на экране соответствующее имя действия может быть выделено светло-серым как недоступное. Приемники изменений свойств представляют собой общую конструкцию, входящую в состав модели компонентов JavaBeans. Подробнее компоненты JavaBeans и их свойства будут рассматриваться во втором томе данной книги.

Следует, однако, иметь в виду, что Action является *интерфейсом*, а не классом. Любой класс, реализующий этот интерфейс, должен реализовать семь только что рассмотренных методов. Правда, сделать это совсем не трудно, поскольку все они, кроме первого метода, содержатся в классе AbstractAction, который предназначен для хранения пар "имя-значение", а также для управления приемниками изменений свойств. На практике для этого достаточно создать соответствующий подкласс и ввести в него метод actionPerformed().

В качестве примера попробуем создать объект действия, изменяющего цвет фона. Для этого в памяти размещаются имя команды, соответствующая пиктограмма и требующийся цвет. Код цвета записывается в таблицу, состоящую из пар "имя-значение", предусмотренных в классе AbstractAction. Ниже приведен исходный код класса ColorAction, в котором выполняются все эти операции. В конструкторе этого класса задаются пары "имя-значение", а в методе actionPerformed() изменяется цвет фона.

```
public class ColorAction extends AbstractAction
{
    public ColorAction(String name, Icon icon, Color c)
    {
        putValue(Action.NAME, name);
        putValue(Action.SMALL_ICON, icon);
        putValue("color", c);
        putValue(Action.SHORT_DESCRIPTION, "Set panel color to " +
            name.toLowerCase());
    }

    public void actionPerformed(ActionEvent event)
    {
        Color c = (Color) getValue("color");
        buttonPanel.setBackground(c);
    }
}
```

В программе, рассматриваемой здесь в качестве примера, сначала создаются три объекта данного класса:

```
Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
```

Затем действие по изменению цвета связывается с соответствующей кнопкой. Для этой цели служит конструктор класса JButton, получающий объект типа ColorAction в качестве параметра, как показано ниже.

```
JButton blueButton = new JButton(blueAction);
```

Этот конструктор считывает имя и пиктограмму действия, размещает его краткое описание во всплывающей подсказке и регистрирует объект типа ColorAction в качестве приемника действий. Пиктограмма и всплывающая подсказка показаны на рис. 8.6. Как будет показано в следующей главе, так же просто действия можно внедрять и в меню.

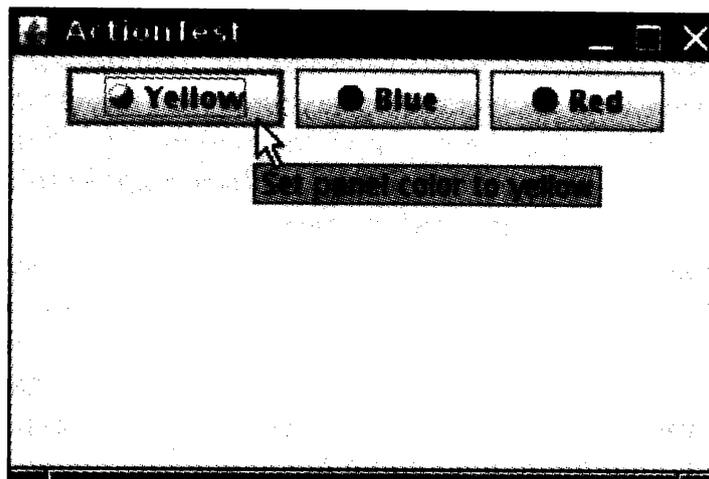


Рис. 8.6. Кнопки с пиктограммами из объектов действий

И, наконец, объекты действий нужно связать с клавишами, чтобы эти действия выполнялись, когда пользователь нажимает соответствующие клавиши. Для того чтобы связать действия с нажатием клавиш, сначала нужно создать объект класса `KeyStroke`. Это удобный класс, инкапсулирующий описание клавиш следующим образом:

```
KeyStroke ctrlBKey = KeyStroke.getKeyStroke("ctrl B");
```

Прежде чем сделать следующий шаг, необходимо разъяснить понятие *фокуса ввода с клавиатуры*. Пользовательский интерфейс может состоять из многих кнопок, меню, полос прокрутки и других компонентов. Нажатия кнопок передаются компонентам, обладающим фокусом ввода. Такой компонент обычно (но не всегда) выделяется для большей наглядности. Например, в визуальном стиле Java кнопка с фокусом ввода имеет тонкую прямоугольную рамку вокруг текста надписи. Для перемещения фокуса ввода между компонентами пользовательского интерфейса можно нажимать клавишу `<Tab>`. Когда же нажимается клавиша пробела, кнопка, обладающая в данный момент фокусом ввода, оказывается выбранной. Другие клавиши вызывают иные действия; например, клавиши со стрелками служат для управления полосами прокрутки.

Но в данном случае нажатия клавиш не нужно посылать компоненту, владеющему фокусом ввода. Вместо этого каждая из кнопок должна обрабатывать события, связанные с нажатием клавиш, и реагировать на комбинации клавиш `<Ctrl+Y>`, `<Ctrl+B>` и `<Ctrl+R>`.

Это часто встречающееся затруднение, и поэтому разработчики библиотеки Swing предложили удобный способ его разрешения. Каждый объект класса `JComponent` содержит три *привязки ввода*, связывающих объекты класса `KeyStroke` с действиями. Эти привязки ввода соответствуют разным условиям, как следует из табл. 8.2.

Таблица 8.2. Условия для привязки ввода

Условие	Вызываемое действие
<code>WHEN_FOCUSED</code>	Когда данный компонент находится в фокусе ввода с клавиатуры
<code>WHEN_ANCESTOR_OF_FOCUSED_COMPONENT</code>	Когда данный компонент содержит другой компонент, находящийся в фокусе ввода с клавиатуры
<code>WHEN_IN_FOCUSED_WINDOW</code>	Когда данный компонент содержится в том же окне, что и компонент, находящийся в фокусе ввода с клавиатуры

При нажатии клавиши условия привязки ввода проверяются в следующем порядке.

1. Проверяется условие привязки ввода `WHEN_FOCUSED` компонента, владеющего фокусом ввода. Если предусмотрена реакция на нажатие клавиши, выполняется соответствующее действие. И если действие разрешено, то обработка прекращается.
2. Начиная с компонента, обладающего фокусом ввода, проверяется условие привязки ввода `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` его родительского компонента. Как только обнаруживается привязка ввода с клавиатуры, выполняется соответствующее действие. И если действие разрешено, то обработка прекращается.
3. Проверяются в окне, обладающем фокусом ввода, все *видимые и активизированные* компоненты с зарегистрированными нажатиями клавиш по условию привязки ввода `WHEN_IN_FOCUSED_WINDOW`. Каждый из этих компонентов (в порядке регистрации нажатий клавиш) получает возможность выполнить соответствующее действие. Как только будет выполнено первое разрешенное действие, обработка прекратится. Если же нажатия клавиш зарегистрированы по нескольким условиям привязки ввода `WHEN_IN_FOCUSED_WINDOW`, то результаты обработки на данном этапе могут быть неоднозначными.

Привязку ввода можно получить из компонента с помощью метода `getInputMap()` следующим образом:

```
InputMap imap = panel.getInputMap(JComponent.WHEN_FOCUSED);
```

Условие привязки ввода `WHEN_FOCUSED` означает, что оно проверяется, если компонент обладает фокусом ввода. В данном случае это условие не проверяется, поскольку фокусом ввода владеет одна кнопка, а не панель в целом. Каждое из оставшихся двух условий привязки ввода также позволяет очень легко изменить цвет фона в ответ на нажатия клавиш. В рассматриваемом здесь примере программы будет проверяться условие привязки ввода `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT`.

Класс `InputMap` не связывает напрямую объекты типа `KeyStroke` с объектами класса `ColorAction`, реализующего интерфейс `Action`. Вместо этого он выполняет первую привязку к произвольным объектам, а вторую привязку, реализованную в классе `ActionMap`, — объектов к действиями. Благодаря этому упрощается выполнение одних и тех же действий при нажатии клавиш, зарегистрированных по разным условиям привязки ввода.

Таким образом, у каждого компонента имеются три привязки ввода и одна привязка действия. Для того чтобы связать их вместе, каждому действию нужно присвоить соответствующее имя. Ниже показано, каким образом комбинация клавиш связывается с нужным действием.

```
imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");  
ActionMap amap = panel.getActionMap();  
amap.put("panel.yellow", yellowAction);
```

Если требуется задать отсутствие действия, то обычно указывается символьная строка `"none"` (отсутствует). Это позволяет легко запретить реагирование на нажатие определенной комбинации клавиш, как показано ниже.

```
imap.put(KeyStroke.getKeyStroke("ctrl C"), "none");
```



ВНИМАНИЕ! В документации на JDK предполагается, что названия клавиш и соответствующего действия совпадают. Такое решение вряд ли можно считать оптимальным. Название действия отображается на кнопке и в пункте меню, но оно может изменяться в процессе разработки. Это, в частности, неизбежно при интернационализации пользовательского интерфейса на разных языках. Поэтому рекомендуется присваивать действиям названия независимо от названий, отображаемых на экране.

Итак, чтобы одно и то же действие выполнялось в ответ на щелчок на кнопке, выбор пункта меню или нажатие клавиши, следует предпринять описанные ниже шаги.

1. Реализовать класс, расширяющий класс `AbstractAction`. Один класс можно будет использовать для программирования разных взаимосвязанных действий.
2. Создать объект класса действия.
3. Сконструировать кнопку или пункт меню из объекта действия. Конструктор прочтет текст метки и пиктограмму из объекта действия.
4. Для действий, которые выполняются в ответ на нажатие клавиш, нужно предпринять дополнительные шаги.
 - Сначала следует обнаружить в окне компонент верхнего уровня, например, панель, содержащую все остальные компоненты.
 - Затем проверить условие привязки ввода `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` компонента верхнего уровня.
 - Создать объект типа `KeyStroke` для нужного нажатия клавиш.
 - Создать объект, соответствующий нажатию клавиш, например, символьную строку, описывающую нужное действие.
 - Добавить пару (нажатие клавиш, ответное действие) к привязке ввода.
 - И наконец, получить привязку действия для компонента верхнего уровня, а затем добавить пару (ответное действие, объект действия) к привязке действия.

В листинге 8.3 приведен весь исходный код программы, привязывающей нажатия как кнопок, так и клавиш к соответствующим действиям. Эти действия выполняются в ответ на нажатия кнопок или комбинаций клавиш `<Ctrl+Y>`, `<Ctrl+B>` или `<Ctrl+R>`.

Листинг 8.3. Исходный код из файла `action/ActionFrame.java`

```
1 package action;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Фрейм с панелью для демонстрации действий по изменению цвета
9  */
10 public class ActionFrame extends JFrame
11 {
12     private JPanel buttonPanel;
13     private static final int DEFAULT_WIDTH = 300;
14     private static final int DEFAULT_HEIGHT = 200;
15 }
```

```
16 public ActionFrame()
17 {
18     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
19
20     buttonPanel = new JPanel();
21
22     // определить действия
23     Action yellowAction = new ColorAction("Yellow",
24         new ImageIcon("yellow-ball.gif"), Color.YELLOW);
25     Action blueAction = new ColorAction("Blue",
26         new ImageIcon("blue-ball.gif"), Color.BLUE);
27     Action redAction = new ColorAction("Red",
28         new ImageIcon("red-ball.gif"), Color.RED);
29
30     // ввести кнопки для этих действий
31     buttonPanel.add(new JButton(yellowAction));
32     buttonPanel.add(new JButton(blueAction));
33     buttonPanel.add(new JButton(redAction));
34
35     // ввести панель во фрейм
36     add(buttonPanel);
37
38     // привязать клавиши <Y>, <B>, <R> к надписям на кнопках
39     InputMap imap =
40         buttonPanel.getInputMap(JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
41     imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
42     imap.put(KeyStroke.getKeyStroke("ctrl B"), "panel.blue");
43     imap.put(KeyStroke.getKeyStroke("ctrl R"), "panel.red");
44
45     // привязать надписи на кнопках панели к действиям
46     ActionMap amap = buttonPanel.getActionMap();
47     amap.put("panel.yellow", yellowAction);
48     amap.put("panel.blue", blueAction);
49     amap.put("panel.red", redAction);
50 }
51
52 public class ColorAction extends AbstractAction
53 {
54     /**
55      * Конструирует действие по изменению цвета
56      * @param name Надпись на кнопке
57      * @param icon Пиктограмма на кнопке
58      * @param c Цвет фона
59      */
60     public ColorAction(String name, Icon icon, Color c)
61     {
62         putValue(Action.NAME, name);
63         putValue(Action.SMALL_ICON, icon);
64         putValue(Action.SHORT_DESCRIPTION, "Set panel color to " +
65             name.toLowerCase());
66         putValue("color", c);
67     }
68
69     public void actionPerformed(ActionEvent event)
70     {
71         Color c = (Color) getValue("color");
72         buttonPanel.setBackground(c);
73     }
74 }
```

javax.swing.Action 1.2

- `boolean isEnabled()`
- `void setEnabled(boolean b)`

Получают или устанавливают свойство `enabled` объекта данного действия.

- `void putValue(String key, Object value)`
- Размещает пару "имя-значение" в объекте действия.

Параметры: `key` Имя компонента, сохраняемое в объекте действия.
Может быть задано любой строкой, но ряд имен имеет специальное значение (см. табл. 8.1)

`value` Объект, связанный с именем

- `Object getValue(String key)`
Возвращает значение из сохраненной пары "имя-значение".

javax.swing.KeyStroke 1.2

- `static KeyStroke getKeyStroke(String description)`

Конструирует объект типа `KeyStroke` из удобочитаемого описания (последовательности символьных строк, разделяемых пробелами). Описание начинается с нулевого или большего количества модификаторов `shift control ctrl meta alt altGraf` и оканчивается словом `typed` и последующим символом (например, "typed a") или необязательным спецификатором события (по умолчанию — `pressed` или `released`) и последующим кодом клавиши. Код клавиши, снабженный префиксом `VK_`, должен соответствовать константе `KeyEvent`; например, код клавиши `<INSERT>` соответствует константе `KeyEvent.VK_INSERT`.

javax.swing.JComponent 1.2

- `ActionMap getActionMap() 1.3`

Возвращает привязку действия, связывающую назначенные клавиши действий, которые могут быть произвольными объектами, с объектами класса, реализующего интерфейс `Action`.

- `InputMap getInputMap(int flag) 1.3`

Получает привязку ввода, связывающую нажатия клавиш с клавишами действий.

Параметры: `flag` Одно из условий для фокуса ввода с клавиатуры,
по которому инициируется действие (см. табл. 8.2)

События от мыши

Для того чтобы предоставить пользователю возможность щелкнуть на кнопке или выбрать пункт меню, совсем не обязательно обрабатывать явным образом события от мыши. Операции с мышью автоматически обрабатываются компонентами пользовательского интерфейса. Но если пользователь должен иметь возможность рисовать мышью, то придется отслеживать ее перемещения, щелчки и события, наступающие при перетаскивании объектов на экране.

В этом разделе будет рассмотрен пример простого графического редактора, позволяющего создавать, перемещать и стирать квадраты на холсте (рис. 8.7).

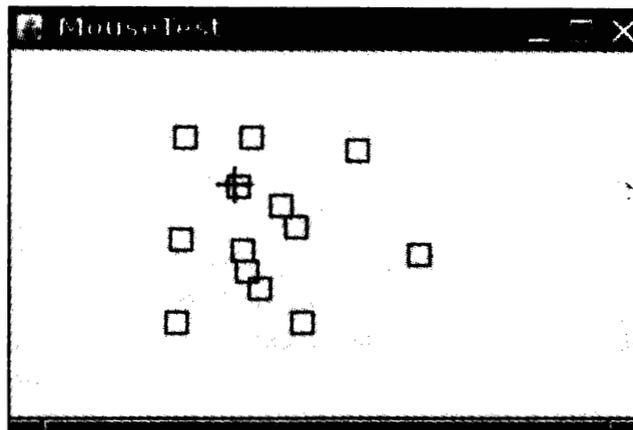


Рис. 8.7. Программа, демонстрирующая обработку событий от мыши

Когда пользователь щелкнет кнопкой мыши, вызываются следующие три метода объекта приемника событий: `mousePressed()`, если кнопка мыши нажата; `mouseReleased()`, если кнопка отпущена; а также `mouseClicked()`, если произведен щелчок. Если требуется отследить только сам щелчок, применять первые два метода не обязательно. Вызывая методы `getX()` и `getY()` для объекта типа `MouseEvent`, передаваемого в качестве параметра, можно определить координаты положения курсора мыши в момент щелчка. А если требуется отличить обычный щелчок от двойного и тройного (!), то вызывается метод `getClickCount()`.

Некоторые разработчики приложений причиняют пользователям массу неудобств, вынуждая их щелкать кнопкой мыши при нажатых модифицирующих клавишах, например `<Control+Shift+щелчок>` для манипулирования элементами ГПИ. Мы находим такую практику предосудительной, но если вы не согласны с нами, то непременно обнаружите, что проверка нажатий кнопок мыши вместе с модифицирующими клавишами оказывается довольно запутанной. Для проверки установленных модификаторов служат битовые маски. В исходном прикладном интерфейсе API маски, соответствующие двум кнопкам мыши, совпадали с масками модифицирующих клавиш:

```
BUTTON2_MASK == ALT_MASK  
BUTTON3_MASK == META_MASK
```

Это было сделано для того, чтобы пользователи однокнопочных мышей могли имитировать нажатие других кнопок мыши, удерживая нажатыми модифицирующие клавиши на клавиатуре. Но в версии Java SE 1.4 был принят другой подход, предусматривающий следующие маски:

```
BUTTON1_DOWN_MASK  
BUTTON2_DOWN_MASK  
BUTTON3_DOWN_MASK  
SHIFT_DOWN_MASK  
CTRL_DOWN_MASK  
ALT_DOWN_MASK  
ALT_GRAPH_DOWN_MASK  
META_DOWN_MASK
```

Метод `getModifiersEx()` позволяет распознавать события, наступающие при одновременном нажатии кнопок мыши и модифицирующих клавиш. Обратите внимание на то, что маска `BUTTON3_DOWN_MASK` позволяет проверять нажатие правой кнопки мыши в Windows. Например, для того чтобы определить, была ли нажата правая кнопка мыши, можно воспользоваться следующим кодом:

```
if ((event.getModifiersEx() & InputEvent.BUTTON3_DOWN_MASK) != 0)  
. . . // обработки события, наступающего при нажатии правой кнопки мыши
```

В программе, рассматриваемой здесь в качестве примера, применяются методы `mousePressed()` и `mouseClicked()`. Если щелкнуть кнопкой мыши на пикселе, не принадлежащем ни одному из нарисованных квадратов, на экране появится новый квадрат. Эта процедура реализована в методе `mousePressed()`, поэтому реакция на щелчок кнопкой мыши произойдет немедленно, не дожидаясь отпущения кнопки мыши. Двойной щелчок внутри какого-нибудь квадрата приведет к его стиранию. Эта процедура реализована в методе `mouseClicked()`, поскольку щелчки кнопкой мыши нужно подсчитывать. Ниже приведен исходный код обоих методов.

```
public void mousePressed(MouseEvent event)  
{  
    current = find(event.getPoint());  
    if (current == null) // за пределами квадрата  
        add(event.getPoint());  
}  
  
public void mouseClicked(MouseEvent event)  
{  
    current = find(event.getPoint());  
    if (current != null && event.getClickCount() >= 2)  
        remove(current);  
}
```

При перемещении мыши по окну последнее получает постоянный поток событий, связанных с движением мыши. Для их отслеживания служат отдельные интерфейсы `MouseListener` и `MouseMotionListener`. Это сделано из соображений эффективности, чтобы приемник событий от щелчков кнопкой мыши игнорировал события, наступающие при перемещении мыши.

Рассматриваемая здесь программа отслеживает события от перемещений мыши, чтобы изменить вид курсора (на крестообразный), как только он выйдет за пределы квадрата. Эта процедура реализована в методе `getPredefinedCursor()` из класса `Cursor`. В табл. 8.3 приведены константы, используемые в этом методе, а также показаны виды курсоров для Windows.

Таблица 8.3. Виды курсоров и соответствующие константы

Пиктограмма	Константа	Пиктограмма	Константа
	DEFAULT_CURSOR		NE_RESIZE_CURSOR
	CROSSHAIR_CURSOR		E_RESIZE_CURSOR
	HAND_CURSOR		SE_RESIZE_CURSOR
	MOVE_CURSOR		S_RESIZE_CURSOR
	TEXT_CURSOR		SW_RESIZE_CURSOR
	WAIT_CURSOR		W_RESIZE_CURSOR
	N_RESIZE_CURSOR		NW_RESIZE_CURSOR

Ниже приведен исходный код метода `mouseMoved()`, объявленного в интерфейсе `MouseListener` и реализованного в рассматриваемой здесь программе для отслеживания перемещений мыши и установки соответствующего курсора.

```
public void mouseMoved(MouseEvent event)
{
    if (find(event.getPoint()) == null)
        setCursor(Cursor.getDefaultCursor());
    else
        setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
}
```



НА ЗАМЕТКУ! Используя метод `createCustomCursor()` из класса `Toolkit`, можно определить новый вид курсора следующим образом:

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image img = tk.getImage("dynamite.gif");
Cursor dynamiteCursor =
    tk.createCustomCursor(img, new Point(10, 10), "dynamite stick");
```

Первый параметр метода `createCustomCursor()` задает изображение курсора, второй его параметр определяет координаты точки выбора курсора, а третий представляет собой символьную строку, описывающую курсор. Эта строка служит для поддержки специальных возможностей, позволяющих, например, озвучивать голосом вид курсора, что удобно для пользователей с нарушениями зрения или вообще не смотрящих на экран.

Если перемещение мыши осуществляется при ее нажатой кнопке, вместо метода `mouseClicked()` вызывается метод `mouseDragged()`. В данном примере квадраты можно перетаскивать по экрану. При этом квадрат перемещается таким образом, чтобы его центр располагался в той точке, где находится указатель мыши. Содержимое холста перерисовывается, чтобы отобразить новое положение указателя мыши. Ниже приведен исходный код метода `mouseClicked()`.

```
public void mouseDragged(MouseEvent event)
{
    if (current != null)
    {
```

```

int x = event.getX();
int y = event.getY();
current.setFrame(x - SIDELENGTH / 2,
    y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
repaint();
}
}

```



НА ЗАМЕТКУ! Метод `mouseMoved()` вызывается только в том случае, если указатель мыши находится внутри компонента. Но метод `mouseDragged()` вызывается даже тогда, когда указатель мыши находится за пределами компонента.

Имеются еще два метода, обрабатывающих события от мыши: `mouseEntered()` и `mouseExited()`. Эти методы вызываются в тех случаях, когда указатель мыши входит в компонент и выходит из него.

И наконец, поясним, каким образом отслеживаются и обрабатываются события от мыши. В ответ на щелчок кнопкой мыши вызывается метод `mouseClicked()`, входящий в состав интерфейса `MouseListener`. Во многих приложениях отслеживаются только щелчки кнопкой мыши, а перемещения мыши происходят слишком часто, поэтому события, связанные с перемещением мыши и перетаскиванием объектов, определяются в отдельном интерфейсе `MouseMotionListener`.

В рассматриваемой здесь программе отслеживаются и обрабатываются оба упомянутых выше вида событий от мыши. Для этого в ней определены два внутренних класса: `MouseHandler` и `MouseMotionHandler`. Класс `MouseHandler` расширяет класс `MouseAdapter`, поскольку в нем определяются только два из пяти методов интерфейса `MouseListener`. А класс `MouseMotionHandler` реализует интерфейс `MouseMotionListener` и определяет оба его метода. Весь исходный код данной программы приведен в листингах 8.4 (фрейм с панелью) и 8.5 (компонент для операций, выполняемых мышью).

Листинг 8.4. Исходный код из файла `mouse/MouseFrame.java`

```

1 package mouse;
2
3 import javax.swing.*;
4
5 /**
6  * Фрейм, содержащий панель для проверки операций, выполняемых мышью
7  */
8 public class MouseFrame extends JFrame
9 {
10     public MouseFrame()
11     {
12         add(new MouseComponent());
13         pack();
14     }
15 }

```

Листинг 8.5. Исходный код из файла `mouse/MouseComponent.java`

```

1 package mouse;
2
3 import java.awt.*;

```

```
4 import java.awt.event.*;
5 import java.awt.geom.*;
6 import java.util.*;
7 import javax.swing.*;
8
9 /**
10  * Компонент для операций мышью по добавлению и удалению квадратов
11  */
12 public class MouseComponent extends JComponent
13 {
14     private static final int SIDELENGTH = 10;
15     private ArrayList<Rectangle2D> squares;
16     private Rectangle2D current;
17
18     public MouseComponent()
19     {
20         squares = new ArrayList<>();
21         current = null;
22
23         addMouseListener(new MouseHandler());
24         addMouseMotionListener(new MouseMotionHandler());
25     }
26
27     public void paintComponent(Graphics g)
28     {
29         Graphics2D g2 = (Graphics2D) g;
30
31         // нарисовать все квадраты
32         for (Rectangle2D r : squares)
33             g2.draw(r);
34     }
35     /**
36      * Обнаруживает первый квадрат, содержащий заданную точку
37      * @param p Точка
38      * @return Первый квадрат, содержащий точку p
39      */
40     public Rectangle2D find(Point2D p)
41     {
42         for (Rectangle2D r : squares)
43         {
44             if (r.contains(p)) return r;
45         }
46         return null;
47     }
48
49     /**
50      * Добавляет квадрат в коллекцию
51      * @param p Центр квадрата
52      */
53     public void add(Point2D p)
54     {
55         double x = p.getX();
56         double y = p.getY();
57
58         current = new Rectangle2D.Double(x - SIDELENGTH / 2,
59             y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
60         squares.add(current);
61         repaint();
62     }
63 }
```

```
62 }
63
64 /**
65  * Удаляет квадрат из коллекции
66  * @param s Удаляемый квадрат
67  */
68 public void remove(Rectangle2D s)
69 {
70     if (s == null) return;
71     if (s == current) current = null;
72     squares.remove(s);
73     repaint();
74 }
75 // квадрат, содержащий указатель мыши
76 private class MouseHandler extends MouseAdapter
77 {
78     public void mousePressed(MouseEvent event)
79     {
80         // добавить новый квадрат, если курсор находится вне квадрата
81         current = find(event.getPoint());
82         if (current == null) add(event.getPoint());
83     }
84     public void mouseClicked(MouseEvent event)
85     {
86         // удалить текущий квадрат, если на нем произведен двойной щелчок
87         current = find(event.getPoint());
88         if (current != null && event.getClickCount() >= 2) remove(current);
89     }
90 }
91
92 private class MouseMotionHandler implements MouseMotionListener
93 {
94     public void mouseMoved(MouseEvent event)
95     {
96         // задать курсор в виде перекрестья,
97         // если он находится внутри квадрата
98         if (find(event.getPoint()) == null)
99             setCursor(Cursor.getDefaultCursor());
100        else setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
101    }
102    public void mouseDragged(MouseEvent event)
103    {
104        if (current != null)
105        {
106            int x = event.getX();
107            int y = event.getY();
108
109            // перетащить текущий квадрат, чтобы отцентровать его
110            // в точке с координатами (x, y)
111            current.setFrame(x - SIDELENGTH / 2,
112                            y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
113            repaint();
114        }
115    }
116 }
```

java.awt.event.MouseEvent 1.1

- `int getX()`
- `int getY()`
- `Point getPoint()`

Возвращают горизонтальную (*x*) и вертикальную (*y*) координаты или точку, в которой наступило событие. Отсчет производится от левого верхнего угла компонента, являющегося источником события.

- `int getClickCount()`

Возвращает количество последовательных щелчков кнопкой мыши, связанных с данным событием. (Промежуток времени, в пределах которого подсчитываются щелчки, зависит от операционной системы.)

java.awt.event.InputEvent 1.1

- `int getModifiersEx() 1.4`

Возвращает модификаторы нажатия кнопок мыши или клавиш для данного события. При проверке возвращаемого значения используются перечисленные ниже маски.

```
BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK
```

- `static String getModifiersExText(int modifiers) 1.4`

Возвращает символьную строку, описывающую модификаторы нажатия кнопок мыши или клавиш из заданного набора, например "Shift+Button1".

java.awt.Toolkit 1.0

- `public Cursor createCustomCursor(Image image, Point hotSpot, String name) 1.2`
- Создает новый объект для специального курсора.

<i>image</i>	Изображение активного курсора
<i>hotSpot</i>	Точка выбора курсора (например, наконечник стрелки или центр перекрестья)
<i>name</i>	Описание курсора для сред с поддержкой специальных возможностей

java.awt.Component 1.0

- `public void setCursor(Cursor cursor) 1.1`
Изменяет внешний вид курсора.

Иерархия событий в библиотеке AWT

После обсуждения механизма обработки событий завершим эту главу общим обзором архитектуры обработки событий в библиотеке AWT. Как упоминалось вкратце ранее, для обработки событий в Java применяется объектно-ориентированный подход. Все события являются производными от класса `EventObject` из пакета `java.util`. (Общий суперкласс не назвали именем `Event`, потому что это имя носил класс событий в старой модели. Хотя старая модель в настоящее время не рекомендована к применению, тем не менее ее классы все еще входят в библиотеку Java.)

У класса `EventObject` имеется подкласс `AWTEvent`, являющийся родительским для всех классов событий из библиотеки AWT. На рис. 8.8 схематически показана иерархия наследования классов событий в библиотеке AWT. Некоторые компоненты из библиотеки Swing формируют объекты других видов событий и непосредственно расширяют класс `EventObject`.

Объекты событий инкапсулируют данные, которые источник событий передает приемникам. По мере необходимости объекты событий, переданные объекту приемника событий, могут быть проанализированы с помощью методов `getSource()` и `getActionCommand()`.

Некоторые классы событий из библиотеки AWT не имеют никакой практической ценности для программирующих на Java. Например, объекты типа `PaintEvent` размещаются из AWT в очереди событий, но эти объекты не доставляются приемнику событий. Программисты должны сами переопределить метод `paintComponent()`, чтобы управлять перерисовкой. В AWT иницируется также ряд событий, интересующих лишь системных программистов. Такие события могут служить для поддержки иероглифических языков, автоматизации тестирования роботов и решения прочих задач. В данной книге эти специализированные типы событий не рассматриваются.

Семантические и низкоуровневые события

События делятся в библиотеке AWT на *низкоуровневые* и *семантические*. Семантические события описывают действия пользователя, например щелчок на кнопке, поэтому событие типа `ActionEvent` является семантическим. А низкоуровневые события обеспечивают возможность подобных действий. Если пользователь щелкнул на кнопке, значит, он нажал кнопку мыши, возможно, переместил курсор по экрану и отпустил кнопку мыши (причем курсор мыши должен находиться в пределах выбираемой кнопки). Семантические события могут быть также иницированы нажатием клавиш, например, для перемещения по кнопкам на панели с помощью клавиши `<Tab>`. Аналогично перемещение бегунка по полосе прокрутки относится к семантическим событиям, тогда как перетаскивание объекта мышью — к низкоуровневым.

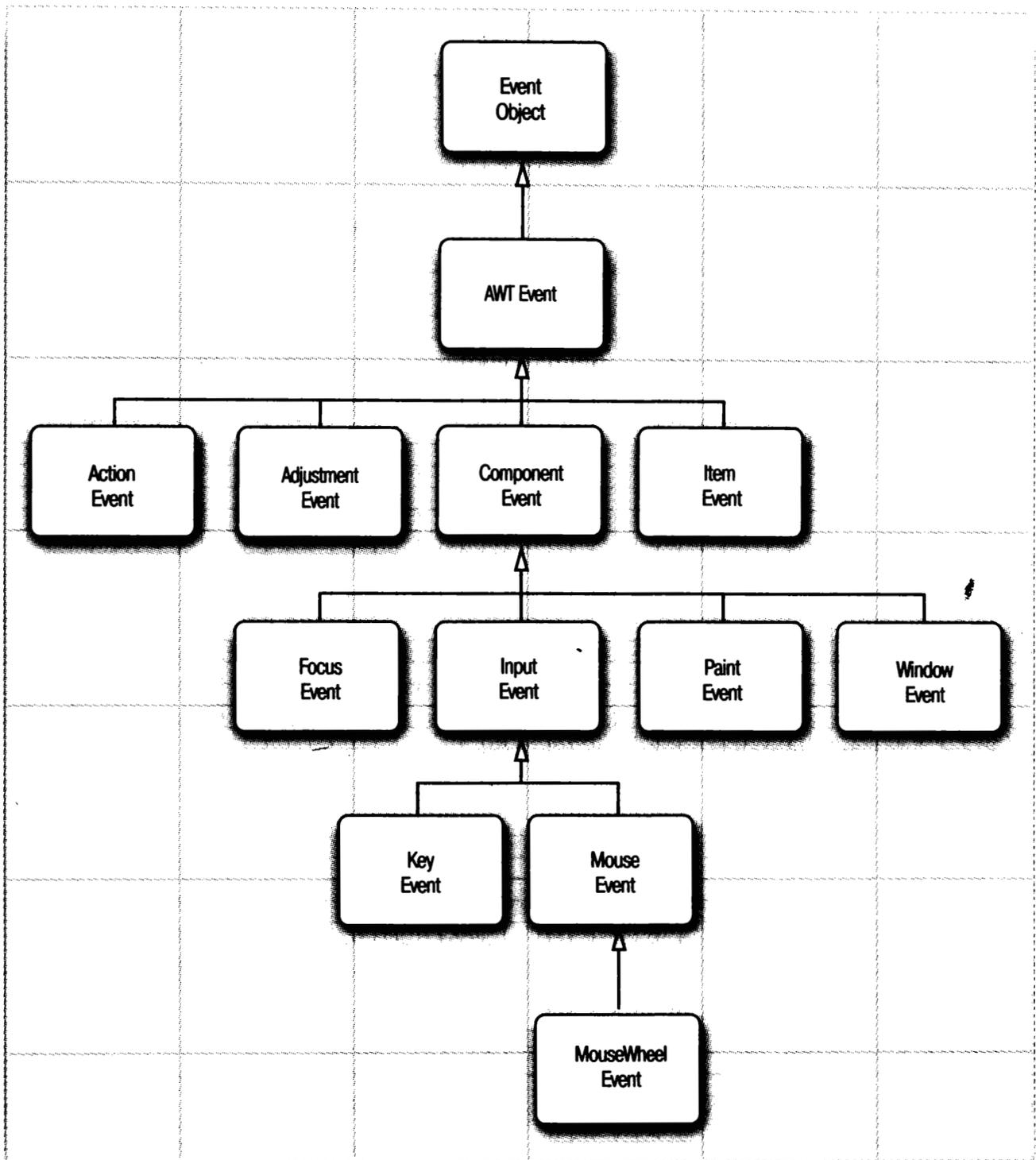


Рис. 8.8. Схематическое представление иерархии наследования классов событий в библиотеке AWT

Из классов, входящих в пакет `java.awt.event` и описывающих семантические события, чаще всего используются следующие.

- Класс `ActionEvent` (щелчок на кнопке, выбор пункта меню, выбор элемента из списка, нажатие клавиши `<Enter>` при вводе текста в поле).
- Класс `AdjustmentEvent` (перемещение бегунка на полосе прокрутки).
- Класс `ItemEvent` (выбор одной из кнопок-переключателей, установка одного из флажков или выбор элемента из списка).

Из классов низкоуровневых событий чаще всего применяются следующие.

- Класс `KeyEvent` (нажатие или отпускание клавиши).
- Класс `MouseEvent` (нажатие и отпускание кнопки мыши, перемещение курсора мыши, перетаскивание курсора, т.е. его перемещение при нажатой кнопке мыши).
- Класс `MouseEvent` (вращение колесика мыши).
- Класс `FocusEvent` (получение или потеря фокуса ввода).
- Класс `WindowEvent` (изменение состояния окна).

Эти события отслеживаются и обрабатываются в следующих интерфейсах:

<code>ActionListener</code>	<code>MouseMotionListener</code>
<code>AdjustmentListener</code>	<code>MouseWheelListener</code>
<code>FocusListener</code>	<code>WindowListener</code>
<code>ItemListener</code>	<code>WindowFocusListener</code>
<code>KeyListener</code>	<code>WindowStateListener</code>
<code>MouseListener</code>	

Одни из интерфейсов приемников событий в библиотеке AWT, а именно те, у которых не один метод, сопровождаются классом адаптера, реализующим все его методы как пустые и ничего не делающие. (А у других интерфейсов имеется лишь один метод, так что никакой выгоды класс адаптера им не приносит.) Ниже перечислены наиболее употребительные классы адаптеров.

<code>FocusAdapter</code>	<code>MouseMotionAdapter</code>
<code>KeyAdapter</code>	<code>WindowAdapter</code>
<code>MouseAdapter</code>	

В табл. 8.4 приведены наиболее важные интерфейсы приемников событий в библиотеке AWT, события и их источники. В пакет `java.swing.event` входят дополнительные события, характерные для компонентов библиотеки Swing. Некоторые из них будут рассмотрены в следующей главе.

Таблица 8.4. Компоненты библиотеки AWT для обработки событий

Интерфейс	Методы	Параметр/Методы доступа	Источник событий
<code>ActionListener</code>	<code>actionPerformed</code>	<code>ActionEvent</code> • <code>getActionCommand</code> • <code>getModifiers</code>	<code>AbstractButton</code> <code>JComboBox</code> <code>JTextField</code> <code>Timer</code>
<code>AdjustmentListener</code>	<code>adjustmentValueChanged</code>	<code>AdjustmentEvent</code> • <code>getAdjustable</code> • <code>getAdjustmentType</code> • <code>getValue</code>	<code>JScrollbar</code>
<code>ItemListener</code>	<code>itemStateChanged</code>	<code>ItemEvent</code> • <code>getItem</code> • <code>getItemSelectable</code> • <code>getStateChange</code>	<code>AbstractButton</code> <code>JComboBox</code>

Окончание табл. 8.4

Интерфейс	Методы	Параметр/Методы доступа	Источник событий
FocusListener	focusGained focusLost	FocusEvent • isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent • getKeyChar • getKeyCode • getKeyModifiersText • getKeyText • isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent • getClickCount • getX • getY • getPoint • translatePoint	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent • getWheelRotation • getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent • getWindow	Window
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent • getOppositeWindow	Window
WindowStateListener	windowStateChanged	WindowEvent • getOldState • getNewState	Window

На этом обсуждение обработки событий в библиотеке AWT завершается. В следующей главе будет показано совместное применение наиболее употребительных компонентов из библиотеки Swing, а также подробно рассмотрены иницилируемые ими события.

Развертывание приложений и апплетов

В этой главе...

- ▶ Файлы формата JAR
- ▶ Технология Java Web Start
- ▶ Апплеты
- ▶ Сохранение глобальных параметров настройки приложений

Приступая к чтению этой главы, вы уже должны чувствовать себя уверенно, пользуясь большинством языковых средств Java. Ведь из предыдущих глав данной книги вы получили достаточное представление об основах программирования графических приложений на Java. Теперь, когда вы готовы к разработке приложений для своих пользователей, вам потребуется также знать, как подготавливать их к развертыванию на компьютерах пользователей. Традиционный способ развертывания, который шумно рекламировался в первые годы существования Java, состоял в применении апплетов. *Апплет* — это особого рода прикладная программа на Java, которую браузер, совместимый с Java, может загружать из Интернета и затем выполнять. Такие прикладные программы первоначально предназначались для того, чтобы освободить пользователей от утомительного процесса установки программного обеспечения, предоставив им доступ к программам с любого компьютера с поддержкой Java или устройства с доступом к Интернету.

По разным причинам апплеты так и не оправдали возлагавшиеся на них надежды. Поэтому в начале этой главы будет рассмотрен порядок упаковки приложений, а затем технология Java Web Start, предоставляющая альтернативный способ доставки интернет-ориентированных приложений с учетом недостатков, характерных для

апплетов. И в конце этой главы будет показано, в каких случаях следует по-прежнему применять апплеты и как сохранять данные о конфигурации и глобальных параметрах настройки приложений.

Файлы формата JAR

Приложение обычно упаковывается для того, чтобы предоставить в распоряжение пользователя единственный файл, а не целую структуру каталогов, заполненную файлами классов. Специально для этой цели был разработан формат архивных файлов Java Archive (JAR). Файл формата JAR (в дальнейшем просто JAR-файл) может содержать, помимо файлов классов, файлы других типов, в том числе файлы изображений и звуковые файлы. Более того, JAR-файлы уплотняются по широко известному алгоритму сжатия данных в формате ZIP.



СОВЕТ. В качестве альтернативы сжатию в формате ZIP применяется алгоритм pack200 как более эффективный для сжатия файлов классов. По заявлениям компании Oracle этот алгоритм обеспечивает степень сжатия файлов классов до 90%. Подробнее об этом можно узнать по адресу <http://docs.oracle.com/javase/1.5.0/docs/guide/deployment/deployment-guide/pack200.html>.

Для создания JAR-файлов служит утилита `jar`. (При установке JDK по умолчанию утилита `jar` располагается в каталоге `jdk/bin`.) Новый JAR-файл создается с помощью следующего синтаксиса командной строки:

```
jar cvf имяJAR-файла файл_1 файл_2 ...
```

Например:

```
jar cvf CalculatorClasses.jar *.java icon.gif
```

Ниже приведена общая форма команды `jar`.

```
jar параметры файл_1 файл_2 ...
```

В табл. 10.1 перечислены все параметры утилиты `jar`. Они напоминают параметры команды `tar`, хорошо известной пользователям операционной системы Unix.

В JAR-файлы можно упаковывать прикладные программы, программные компоненты (иногда называемые *beans*, поскольку они относятся к разряду компонентов JavaBeans, как поясняется в главе 8 второго тома данной книги), а также библиотеки кода. Например, библиотека исполняющей системы JDK содержится в очень крупном файле `rt.jar`.

Таблица 10.1. Параметры утилиты `jar`

Параметр	Описание
<code>c</code>	Создает новый или пустой архив и добавляет в него файлы. Если в качестве имени файла указано имя каталога, утилита <code>jar</code> обрабатывает его рекурсивно
<code>C</code>	Временно изменяет каталог. Например, следующая команда направляет файлы в подкаталог <code>classes</code> : <code>jar cvf JARFileName.jar -C classes *.class</code>
<code>e</code>	Создает точку входа в манифест (см. раздел "Исполняемые JAR-файлы" далее в этой главе)

Окончание табл. 10.1

Параметр	Описание
f	Задаёт имя JAR-файла в качестве второго параметра командной строки. Если этот параметр пропущен, то утилита <code>jar</code> выводит результат в стандартный поток вывода (при создании JAR-файла) или вводит его из стандартного потока ввода (при извлечении или просмотре содержимого JAR-файла)
i	Создаёт индексный файл (для ускорения поиска в крупных архивах)
m	Добавляет в JAR-файл манифест, представляющий собой описание содержимого архива и его происхождения. Манифест создается по умолчанию для каждого архива, но для подробного описания содержимого JAR-файла можно создать свой собственный манифест
M	Отменяет создание манифеста
t	Отображает содержание архива
u	Обновляет существующий JAR-файл
v	Выводит подробные сведения об архиве
x	Извлекает файлы. Если указано несколько имен файлов, извлекаются только они. В противном случае извлекаются все файлы
0	Сохраняет данные в архиве, не упаковывая их в формате ZIP

Файл манифеста

Помимо файлов классов, изображений и прочих ресурсов, каждый JAR-файл содержит также *файл манифеста*, описывающий особые характеристики данного архива. Файл манифеста называется `MANIFEST.MF` и находится в специальном подкаталоге `META-INF`. Минимально допустимый манифест не особенно интересен, как показано ниже.

```
Manifest-Version: 1.0
```

Сложные манифесты содержат много больше элементов описания, группируемых по разделам. Первый раздел манифеста называется *главным*. Он относится ко всему JAR-файлу. Остальные элементы описания относятся к отдельным файлам, пакетам или URL. Эти элементы должны начинаться со слова `Name`. Разделы отделяются друг от друга пустыми строками, как показано в приведенном ниже примере.

```
Manifest-Version: 1.0
```

строки описания данного архива

```
Name: Wozzle.class
```

строки описания данного файла

```
Name: com/mycompany/mypkg/
```

строки описания данного пакета

Для того чтобы отредактировать этот манифест, достаточно ввести в него нужные строки, сохранив их в обычном текстовом файле, а затем выполнить следующую команду:

```
jar cfm имяJAR-файла имяФайлаМанифеста
```

Например, для того чтобы создать новый JAR-файл с манифестом, нужно вызвать утилиту `jar` из командной строки следующим образом:

```
jar cfm MyArchive.jar manifest.mf com/mycompany/mypkg/*.class
```

Чтобы добавить в манифест существующего JAR-файла новые строки, достаточно ввести и сохранить их в текстовом файле, а затем выполнить следующую команду:

```
jar ufm MyArchive.jar manifest-additions.mf
```



НА ЗАМЕТКУ! Подробнее о форматах архивных JAR-файлов, а также файлов манифестов можно узнать по адресу <http://docs.oracle.com/javase/7/docs/technotes/guides/jar>.

Исполняемые JAR-файлы

С помощью параметра `e` утилиты `jar` можно указать *точку входа* в прикладную программу, т.е. класс, который будет обычно указываться при запуске программы по команде `java`, как показано ниже.

```
jar cvfe MyProgram.jar com.mycompany.myprog.MainAppClass добавляемые файлы
```

С другой стороны, в манифесте можно указать *главный класс* прикладной программы, включая оператор в следующей форме:

```
Main-Class: com.mycompany.myprog.MainAppClass
```

Только не добавляйте расширение `.class` к имени главного класса. Независимо от способа указания точки входа в прикладную программу ее пользователи могут запустить ее на выполнение, введя следующую команду:

```
java -jar MyProgram.jar
```



ВНИМАНИЕ! Последняя строка в манифесте должна оканчиваться символом перевода строки. В противном случае манифест не будет прочитан правильно. Это распространенная ошибка — когда создается текстовый файл с единственной строкой `Main-Class`, но без ограничителя строки.

В зависимости от конфигурации операционной системы приложения можно запускать двойным щелчком на пиктограмме JAR-файла. Ниже описываются особенности запуска приложений из JAR-файлов в разных операционных системах.

- В Windows установщик исполняющей системы Java образует сопоставление файлов с расширением `.jar` для запуска подобных файлов по команде `javaw -jar`. (В отличие от команды `java`, команда `javaw` не открывает окно командной оболочки.)
- В Solaris операционная система распознает “магическое число” (т.е. системный код) JAR-файла и запускает его по команде `java -jar`.
- В Mac OS операционная система распознает расширение файла `.jar` и выполняет программу на Java после двойного щелчка на JAR-файле.

Но программа на Java, находящаяся в JAR-файле, имеет тот же вид, что и собственное приложение операционной системы. В Windows можно использовать утилиты-оболочки сторонних производителей, превращающие JAR-файлы в исполняемые файлы Windows. Такая оболочка представляет собой программу для Windows с известным расширением `.exe`, которая запускает виртуальную машину Java (JVM) или же сообщает пользователю, что делать, если эта машина не найдена. Для запуска прикладных программ из JAR-файлов имеется немало коммерческих и свободно

доступных программных продуктов вроде Jsmooth (<http://jsmooth.sourceforge.net>) и Launch4J (<http://launch4j.sourceforge.net>). Генератор установок с открытым кодом IzPack (<http://izpack.org>) также содержит собственную систему запуска программ на Java. Подробнее об этом можно узнать по адресу <http://www.javalobby.org/articles/java2exe>.

На платформе Mac OS дело обстоит немного проще. Утилита Jar Bundler, входящая в состав ИСР XCode, позволяет превратить JAR-файл в первоклассное приложение для Mac OS.

Ресурсы

Классы, применяемые в апплетах и приложениях, часто ассоциируются с файлами данных. Ниже перечислены примеры таких файлов. В Java все эти файлы называются *ресурсами*.

- Графические и звуковые файлы.
- Текстовые файлы, содержащие строки сообщений и метки кнопок.
- Файлы, содержащие двоичные данные, например, для описания разметки карты.



НА ЗАМЕТКУ! В операционной системе Windows термин *ресурсы* имеет более узкий смысл. В этой системе к ресурсам также относятся пиктограммы, метки кнопок и т.п., но они должны быть связаны с исполняемыми файлами, а доступ к ним должен обеспечивать стандартный программный интерфейс. В языке Java, напротив, ресурсы находятся в отдельных файлах, но не являются частью файлов классов, а доступ к ним и интерпретация их данных должны быть организованы в каждой программе.

Рассмотрим в качестве примера класс `AboutPanel`, выводящий на экран сообщение, как показано на рис. 10.1.

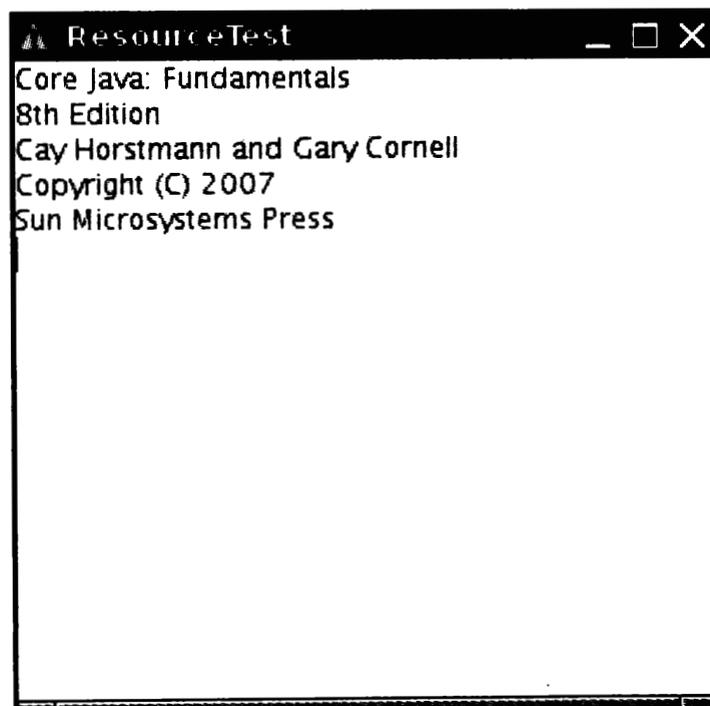


Рис. 10.1. Отображение ресурса из JAR-файла

Безусловно, название книги и год выпуска, указанные на панели, в следующем издании книги изменятся. Для того чтобы легко отслеживать такие изменения, текст описания книги следует разместить в отдельном файле, а не встраивать его в программу в виде символьной строки. Но где же следует разместить файл `about.txt`? Разумеется, было бы удобно разместить его вместе с остальными файлами программы, например в JAR-файле.

Загрузчику классов известно, где находятся файлы классов, если они содержатся в каталогах, указанных в переменной окружения `CLASSPATH`, в архиве или на веб-сервере. Механизм ресурсов предоставляет аналогичные удобства и при обращении с файлами, которые не являются файлами классов. Для этого необходимо выполнить описанные ниже действия.

1. Получить объект типа `Class`, имеющий ресурс, например `AboutPanel.class`.
2. Если ресурс представляет собой графический или звуковой файл, вызвать метод `getResource(имя_файла)`, чтобы определить местонахождения ресурса в виде URL. Затем прочитать его непосредственно с помощью метода `getImage()` или `getAudioClip()`.
3. Если ресурс не является графическим или звуковым файлом, вызвать метод `getResourceAsStream()`, чтобы прочитать данные из файла.

Дело в том, что загрузчик класса помнит, где располагается класс, и может обнаружить там же файлы ресурсов. Например, чтобы создать пиктограмму, хранящуюся в файле `about.gif`, следует воспользоваться приведенным ниже фрагментом кода. Этот код, по существу, означает следующее: "искать файл `about.gif` там же, где находился файл `AboutPanel.class`".

```
URL url = ResourceTest.class.getResource("about.gif");
Image img = new ImageIcon(url).getImage();
```

Чтобы прочитать содержимое файла `about.txt`, нужно написать следующий фрагмент кода:

```
InputStream stream = ResourceTest.class.getResourceAsStream("about.txt");
Scanner in = new Scanner(stream);
```

Ресурсы совсем не обязательно размещать в одном каталоге с файлом классов. Вместо этого их можно разместить в подкаталоге. Имя ресурса с учетом иерархии подкаталогов выглядит следующим образом:

```
data/text/about.txt
```

Это относительное имя ресурса, поэтому оно определяется относительно пакета, которому принадлежит класс, загружающий этот ресурс. Обратите внимание на то, что в этом имени всегда нужно использовать разделитель `/`, независимо от тех разделителей, которые применяются в операционной системе. Например, на платформе Windows загрузчик ресурсов автоматически преобразует символ `/` в разделитель `\`.

Имя ресурса, начинающееся с символа `/`, называется абсолютным. По этому имени ресурс обнаруживается точно так же, как и класс. Например, приведенный ниже ресурс находится в каталоге `corejava`. Он может быть подкаталогом одного из каталогов, указанных в переменной окружения `CLASSPATH`, находиться в JAR-файле или на веб-сервере, если речь идет об апплете.

```
/corejava/title.txt
```

Несмотря на то что ресурс загружается автоматически, стандартных методов интерпретации содержимого файла ресурсов не существует. В каждой прикладной программе приходится по-своему интерпретировать данные, находящиеся в подобных файлах.

Кроме того, ресурсы применяются для интернационализации программ. В файлах ресурсов, например, содержатся сообщения и метки на разных языках. Каждый такой файл соответствует отдельному языку. В прикладном интерфейсе API, который будет обсуждаться в главе 5 второго тома данной книги, для интернационализации поддерживается стандартный способ организации и доступа к этим файлам локализации.

В листинге 10.1 приведен исходный код программы, демонстрирующей порядок загрузки ресурсов. Скомпилируйте, соберите JAR-файл и запустите его на выполнение, выполнив следующие команды:

```
javac ResourceTest.java
jar cvfm ResourceTest.jar ResourceTest.mf *.class *.gif *.txt
java -jar ResourceTest.jar
```

Переместите JAR-файл в другой каталог и запустите его снова, чтобы убедиться, что программа прочитает все, что ей нужно, из JAR-файла, а не из текущего каталога.

Листинг 10.1. Исходный код из файла resource/ResourceTest.java

```
1 package resource;
2
3 import java.awt.*;
4 import java.io.*;
5 import java.net.*;
6 import java.util.*;
7 import javax.swing.*;
8
9 /**
10  * @version 1.4 2007-04-30
11  * @author Cay Horstmann
12  */
13 public class ResourceTest
14 {
15     public static void main(String[] args)
16     {
17         EventQueue.invokeLater(new Runnable()
18         {
19             public void run()
20             {
21                 JFrame frame = new ResourceTestFrame();
22                 frame.setTitle("ResourceTest");
23                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24                 frame.setVisible(true);
25             }
26         });
27     }
28 }
29
30 /**
31  * Фрейм, в котором загружаются ресурсы изображений и текста
32  */
33 class ResourceTestFrame extends JFrame
```

```

34 {
35     private static final int DEFAULT_WIDTH = 300;
36     private static final int DEFAULT_HEIGHT = 300;
37
38     public ResourceTestFrame()
39     {
40         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
41         URL aboutURL = getClass().getResource("about.gif");
42         Image img = new ImageIcon(aboutURL).getImage();
43         setIconImage(img);
44
45         JTextArea textArea = new JTextArea();
46         InputStream stream = getClass().getResourceAsStream("about.txt");
47         Scanner in = new Scanner(stream);
48         while (in.hasNext())
49             textArea.append(in.nextLine() + "\n");
50         add(textArea);
51     }
52 }

```

java.lang.Class 1.0

- URL getResource(String name) 1.1
- InputStream getResourceAsStream(String name) 1.1

Находят ресурс там же, где расположен класс, и возвращают его URL или поток ввода, который можно использовать для загрузки. Если ресурс не найден, эти методы возвращают пустое значение null, не генерируя исключений и не сообщая об ошибках ввода-вывода.

Герметизация пакетов

Как упоминалось в главе 4, пакет, написанный на Java, можно *герметизировать*, чтобы в него больше нельзя было добавлять новые классы. Разработчики обычно применяют герметизацию в том случае, если они используют классы, методы и поля, имеющие область действия, ограниченную пакетом.

Так, если герметизируется пакет com.mycompany.util, то ни один из классов, находящихся за пределами герметизированного архива, нельзя будет определить с помощью следующего оператора:

```
package com.mycompany.util;
```

Для этого все классы из пакета размещаются в архивном JAR-файле. По умолчанию пакеты в JAR-файле не герметизированы. Изменить глобальные установки по умолчанию можно, введя в главный раздел файла манифеста следующую строку:

```
Sealed: true
```

Для каждого пакета можно отдельно указать, следует ли его герметизировать, добавив в файл манифеста новый раздел, как показано ниже.

```
Name: com/mycompany/util/
Sealed: true
```

```
Name: com/mycompany/misc/
Sealed: false
```

Для того чтобы герметизировать пакет, необходимо создать текстовый файл с инструкциями манифеста, а затем выполнить приведенную ниже команду.

```
jar cvfm MyArchive.jar manifest.mf добавляемые файлы
```

Технология Java Web Start

Java Web Start — это технология, предназначенная для доставки приложений через Интернет. Приложения Java Web Start имеют следующие характеристики.

- Обычно доставляются с помощью браузера. Как только приложение Java Web Start загрузится, оно может быть запущено без браузера.
- Не действуют в окне браузера. Приложение отображается в своем собственном окне вне браузера.
- Не пользуются реализацией браузера на Java. Браузер запускает внешнюю программу при получении дескриптора приложения. Для этой цели служит механизм, аналогичный применяемому для запуска вспомогательных приложений вроде Adobe Acrobat или RealAudio.
- Приложениям, снабженным цифровой подписью, могут быть предоставлены права свободного доступа к локальной машине. Приложения, не имеющие подписи, запускаются в изолированной среде исполнения (так называемой “песочнице”), что препятствует выполнению потенциально опасных операций.

Для того чтобы подготовить приложение к доставке с помощью технологии Java Web Start, необходимо упаковать его в один или несколько JAR-файлов. Затем подготавливается файл описания в формате JNLP (Java Network Launch Protocol — сетевой протокол запуска приложений на Java). Далее файлы размещаются на веб-сервере. После этого следует убедиться, что на веб-сервере тип `application/x-java-jnlp-file` расширения MIME связан с расширением файлов `.jnlp`. (В браузерах тип MIME служит для определения вспомогательного приложения, которое следует запустить для поддержки документа.) Более подробные сведения по данному вопросу можно найти в документации, сопровождающей конкретный веб-сервер.



СОВЕТ. Чтобы поэкспериментировать с технологией Java Web Start, установите программу Tomcat. Найти ее можно по адресу <http://tomcat.apache.org/>. Программа Tomcat служит контейнером для сервлетов и JSP-страниц, но она поддерживает и веб-страницы. Эта программа предварительно настроена на правильный тип MIME для обработки JNLP-файлов.

Попробуйте доставить с помощью технологии Java Web Start прикладную программу, реализующую калькулятор и рассмотренную в главе 9. Для этого выполните действия, приведенные ниже.

1. Скомпилируйте программу, выполнив следующую команду:

```
javac -classpath ./path/to/javaws.jar webstart/*.java
```

2. Создайте JAR-файл, выполнив следующую команду:

```
jar cvfe Calculator.jar webstart.Calculator webstart/*.class
```

3. Подготовьте запускающий файл `Calculator.jnlp`, содержимое которого приведено ниже.

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase=
  "http://localhost:8080/calculator/" href="Calculator.jnlp">
  <information>
    <title>Calculator Demo Application</title>
    <vendor>Cay S. Horstmann</vendor>
    <description>A Calculator</description>
    <offline-allowed/>
  </information>
  <resources>
    <java version="1.6.0+"/>
    <jar href="Calculator.jar"/>
  </resources>
  <application-desc/>
</jnlp>
```

4. (Следует иметь в виду, что номером версии должен быть 1.6.0, а не 6.0.) Формат запускающего файла очевиден. С полным описанием его структуры можно ознакомиться по адресу <http://docs.oracle.com/javase/tutorial/deployment/deploymentInDepth/jnlpFileSyntax.html>.
5. Если вы пользуетесь программой Tomcat, создайте каталог `tomcat/webapps/calculator`, где `tomcat` — каталог, в котором установлена программа Tomcat. Создайте подкаталог `tomcat/webapps/calculator/WEB-INF` и разместите в нем минимальную версию файла `web.xml` со следующим содержимым:

```
<?xml version="1.0" encoding="utf-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd">
</web-app>
```

6. Разместите JAR- и JNLP-файл на веб-сервере таким образом, чтобы URL совпадал с тем, что указано в атрибуте `codebase` из JNLP-файла. Если пользуетесь программой Tomcat, разместите файлы в каталоге `tomcat/webapps/calculator`.
7. Убедитесь, правильно ли настроен браузер на Java Web Start, проверив, связан ли тип `application/x-java-jnlp-file` расширения MIME с приложением `javaws`. Если на вашем компьютере установлен пакет JDK, настройка выполняется автоматически.
8. Запустите Tomcat.
9. Укажите в окне браузера веб-адрес местонахождения JNLP-файла. Так, если вы пользуетесь Tomcat, перейдите по адресу `http://localhost:8080/calculator/Calculator.jnlp`.
10. В итоге должно появиться окно запуска Java Web Start, аналогичное приведенному на рис. 10.2.
11. Вскоре после этого должно появиться окно калькулятора, оформление которого свидетельствует о том, что он является приложением Java Web Start (рис. 10.3).

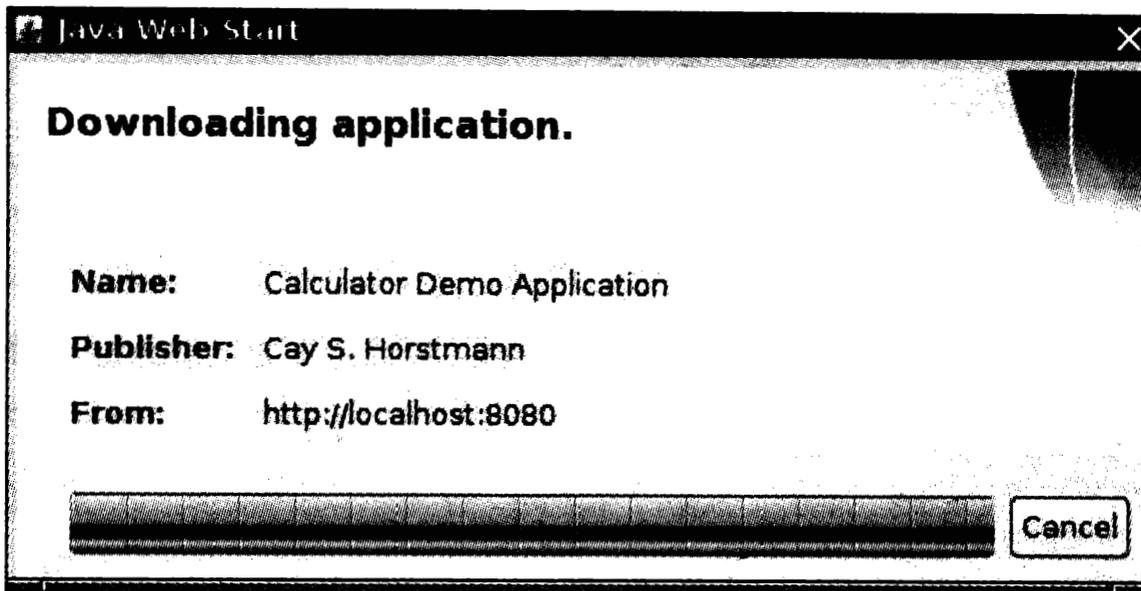


Рис. 10.2. Окно запуска Java Web Start

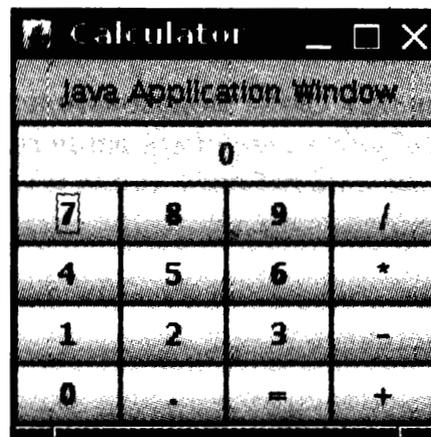


Рис. 10.3. Приложение Calculator, доставленное с помощью технологии Java Web Start

12. Когда вы снова обратитесь к JNLP-файлу, приложение будет извлечено из кеша. Просмотреть содержимое кеша можно с помощью панели управления Java Plug-in (рис. 10.4). Если вы работаете в Windows, панель управления Java Plug-in можно найти на системной панели управления. А в Linux запустите эту панель управления из каталога `jdk/jre/bin/ControlPanel`.



СОВЕТ. Если вы не хотите запускать веб-сервер прежде, чем протестируете конфигурацию JNLP, то можете временно переопределить URL в атрибуте `codebase` из запускающего файла, выполнив следующую команду:

```
javaws -codebase file://каталог_программы JNLP-файл
```

Например, в UNIX вы можете просто выдать следующую команду из каталога, содержащего JNLP-файл:

```
javaws -codebase file://`pwd` WebStartCalculator.jnlp
```

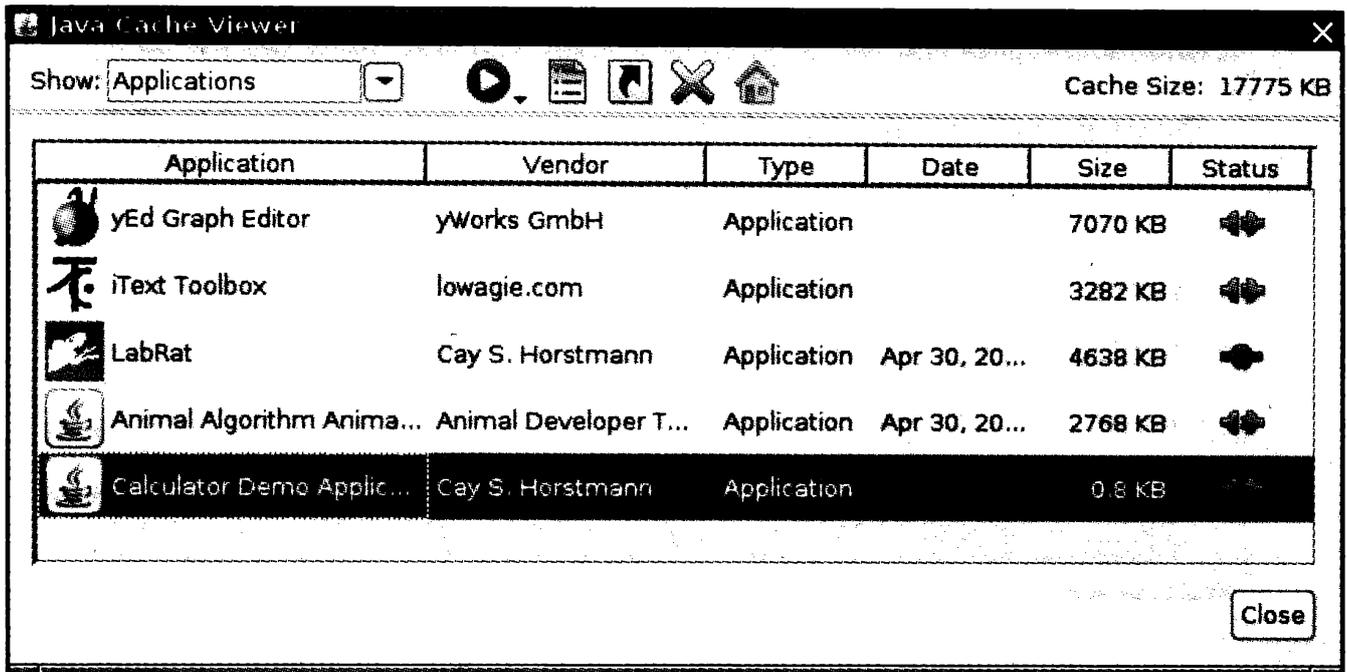


Рис. 10.4. Просмотр кеша приложений

Разумеется, не стоит заставлять своих пользователей запускать средство просмотра кеша всякий раз, когда требуется запустить ваше приложение. Вы можете поручить установщику создать ярлыки на рабочем столе и команды в меню, введя в JNLP-файл следующие строки:

```
<shortcut>
  <desktop/>
  <menu submenu="Accessories"/>
</shortcut>
```

Когда пользователь впервые загружает приложение, отображается “предупреждение об интеграции рабочего стола” (рис. 10.5).

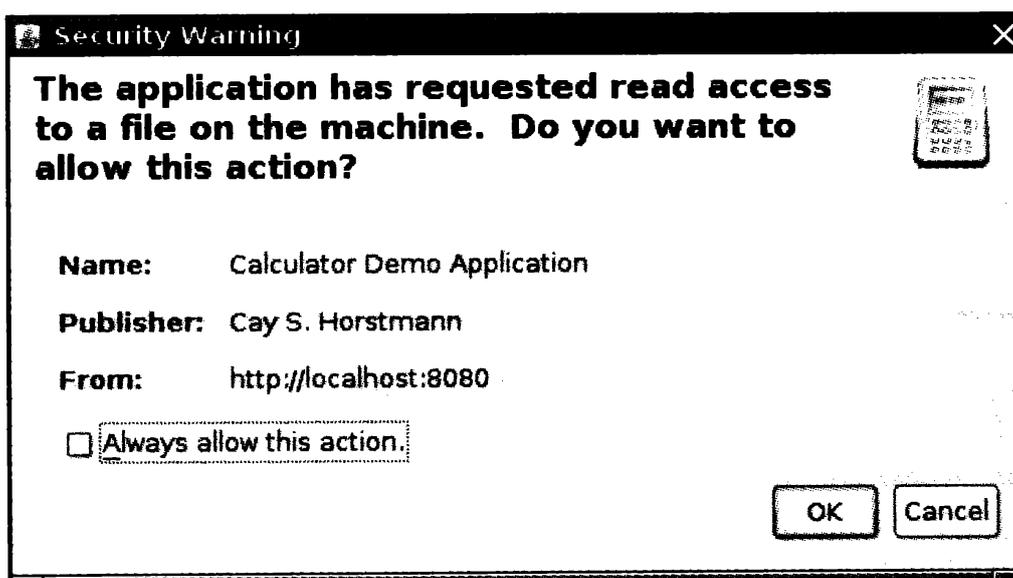


Рис. 10.5. Предупреждение об интеграции рабочего стола

Для команды меню и экрана запуска следует также предоставить пиктограмму. Компания Oracle рекомендует использовать пиктограммы размерами 32×32 и

64×64. Для этого разместите файлы с пиктограммами на веб-сервере вместе с JNLP- и JAR-файлами, а затем введите следующие строки в разделе `information` JNLP-файла:

```
<icon href="calc_icon32.png" width="32" height="32" />  
<icon href="calc_icon64.png" width="64" height="64" />
```

Следует также иметь в виду, что эти пиктограммы не имеют отношения к пиктограмме самого приложения. Если требуется, чтобы приложение имело свою пиктограмму, то вам придется добавить отдельный файл с пиктограммой в JAR-файл и вызвать метод `setIconImage()` из класса фрейма (для примера см. листинг 10.1).

“Песочница”

Всякий раз, когда код загружается с удаленного сайта, а затем исполняется локально, на первый план выступает вопрос безопасности. Приложение Java Web Start может быть запущено щелчком на единственной ссылке. При посещении веб-сайта автоматически запускаются все апплеты на текущей веб-странице. Если пользователь щелкнет на ссылке или перейдет на веб-страницу, на его компьютере может быть установлен произвольный код, с помощью которого злоумышленники получают возможность для хищения конфиденциальной информации, доступа к финансовой информации или использования компьютера пользователя для рассылки нежелательных почтовых сообщений (так называемого *спама*).

Чтобы исключить злоупотребление технологией Java, пришлось обзавестись моделью безопасности, более подробно рассматриваемой во втором томе данной книги. Доступ ко всем системным ресурсам контролирует *диспетчер безопасности*. По умолчанию он разрешает только безвредные операции. Чтобы разрешить дополнительные операции, код должен быть снабжен цифровой подписью, а пользователь должен утвердить подписанный сертификат.

Что же может делать удаленный код на всех платформах? Всегда разрешается показывать изображения, воспроизводить звук, принимать от пользователя нажатия клавиш и щелчки кнопками мыши, а также посылать введенные пользователем данные обратно на сетевой узел (или так называемый *хост*), с которого был загружен код. Этих функциональных возможностей достаточно для отображения фактов и цифр или для получения данных, введенных пользователем для размещения заказа. Изолированная среда исполнения часто называется “песочницей”. Код, исполняющийся в пределах “песочницы”, никак не может повлиять на пользовательскую систему или следить за ней.

В частности, на программы, выполняющиеся в “песочнице”, накладываются следующие ограничения.

- Они вообще не могут запускать локально исполняемые программы.
- Они не могут выполнять операции чтения и записи данных в локальной файловой системе.
- Они не могут извлекать никаких данных о локальном компьютере, за исключением установленной версии Java и некоторых безвредных подробностей, касающихся операционной системы. В частности, код в “песочнице” не может получить имя пользователя, адрес его электронной почты и т.п.
- Удаленно загруженные программы не могут взаимодействовать ни с одним из хостов, за исключением того сервера, с которого они были загружены. Такой

сервер называется *исходящим хостом*. Данное правило часто формулируется следующим образом: “удаленный код может звонить только домой”. Оно защищает пользователей от кода, который может попытаться следить за внутри-сетевыми ресурсами.

- Все всплывающие окна выдают предупреждающее сообщение. Это сообщение является защитным средством, гарантирующим, что пользователь не спутает его с окном локального приложения. Такой прием исключает ситуацию, когда злонамеренный пользователь посещает веб-страницу, обманным путем запускает на выполнение удаленный код, а затем вводит пароль и номер кредитной карточки, и эти конфиденциальные могут быть затем отправлены обратно на веб-сервер. В ранних версиях JDK сообщение в подобных случаях было весьма угрожающим: “Untrusted Java Applet Window” (Окно апплета Java, не заслуживающего доверия). В каждой последующей версии формулировка немного смягчалась: сначала “Unauthenticated Java Applet Window” (Окно неаутентифицированного апплета Java), а затем “Warning: Java Applet Window” (Предупреждение: окно апплета Java). Ныне на эти три сообщения обращают внимание только самые наблюдательные пользователи.

Подписанный код

Ограничения, накладываемые в “песочнице”, зачастую оказываются слишком строгими. Например, нетрудно представить себе приложение Java Web Start или апплет, которому требуется доступ к локальным файлам во внутренней корпоративной сети. Существует возможность для организации очень строгого контроля прав, предоставляемых конкретному приложению, о чем подробнее речь пойдет в главе 9 второго тома данной книги. Конечно, приложение может просто запросить все права настольного приложения, и большая часть приложений Java Web Start так и делает. Это достигается введением следующих дескрипторов в JNLP-файл:

```
<security>
  <all-permissions/>
</security>
```

Чтобы запускаться вне “песочницы”, JAR-файлы приложения Java Web Start должны быть снабжены цифровой подписью. Подписанный JAR-файл удостоверяется сертификатом, подтверждающим подлинность того, кто его подписал. Криптографическая техника гарантирует, что такой сертификат не может быть подделан и что любые попытки проникновения в подписанный файл будут обнаружены.

Допустим, вы получили приложение, созданное и подписанное фирмой уWorks GmbH, с применением сертификата, изданного центром сертификации Thawte (рис. 10.6). Когда вы получаете такое приложение, можете быть уверены в следующем.

1. Код в точности такой, каким он был на момент подписания, и никто посторонний не подделал его.
2. Сигнатура действительно принадлежит фирме уWorks GmbH.
3. Сертификат действительно издан центром Thawte (в Java Web Start предусмотрена проверка сертификатов Thawte, а также некоторых других поставщиков).

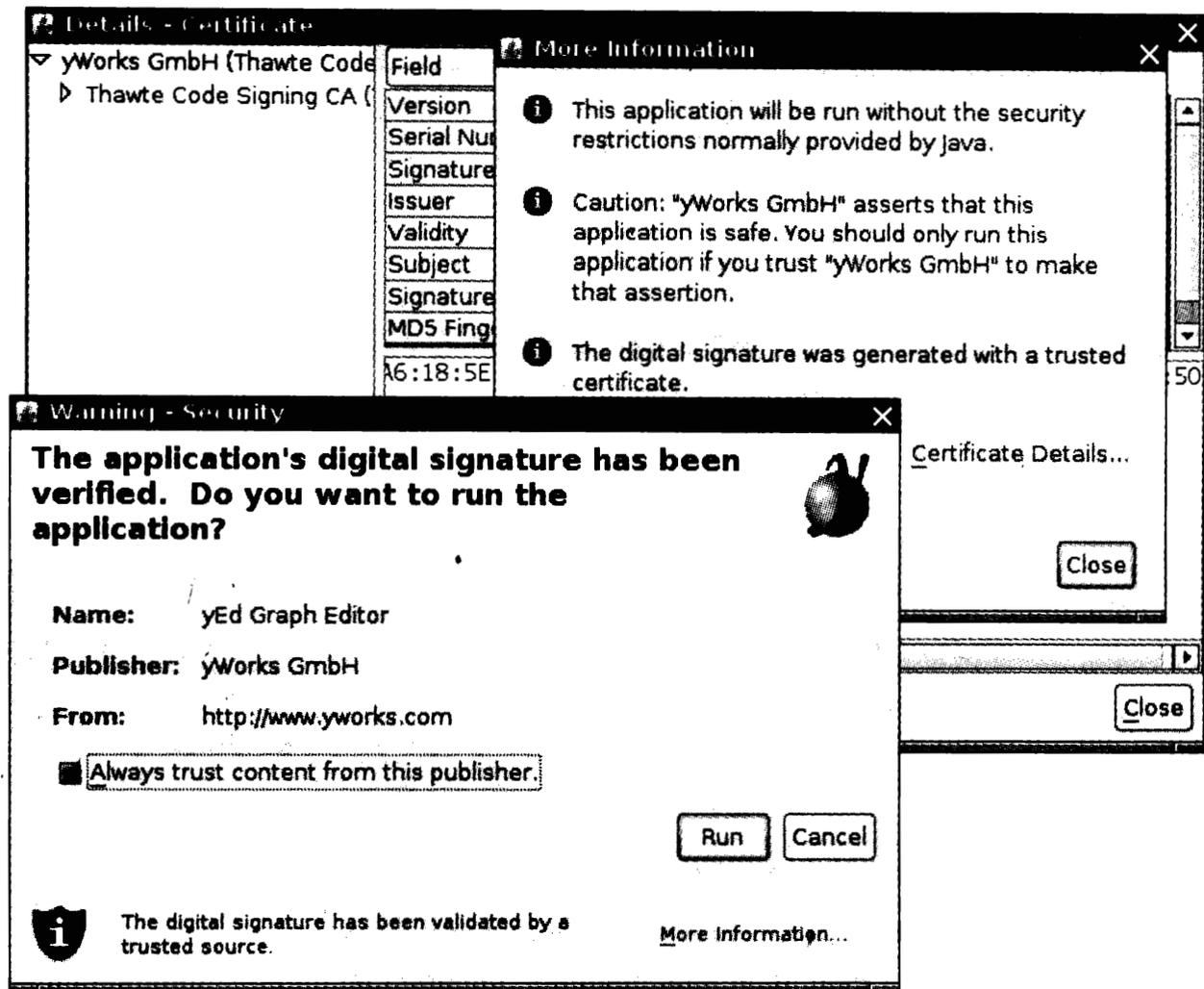


Рис. 10.6. Надежный сертификат

К сожалению, это все, что вам известно. Но вы не знаете главного: действительно ли код безопасен. В действительности, если вы щелкнете на ссылке *More Information* (Дополнительные сведения), то узнаете, что приложение будет запущено без ограничений безопасности, обычно накладываемых Java. Так следует ли устанавливать и запускать такое приложение? На самом деле ответ на этот вопрос зависит от степени вашего доверия к фирме yWorks GmbH.

Получение сертификата от одного из поддерживаемых поставщиков стоит несколько сотен долларов в год. И многие разработчики просто формируют свой собственный сертификат и используют его для подписи кода. Конечно, у Java Web Start нет возможности проверить подлинность таких сертификатов. Поэтому, когда вы получаете приложение с таким сертификатом, вам должно быть известно следующее.

1. Код в точности такой, каким был на момент подписи, и никто его не подделал.
2. Кто-то подписал этот код, но Java Web Start не может проверить, кто это был.

Но все эти меры могут оказаться бесполезными. Ведь любой может подделать код, а затем подписать его, выдав себя за его автора. И несмотря на это, Java Web Start благополучно предоставит сертификат на ваше утверждение (рис. 10.7). Теоретически возможно проверить достоверность сертификата другим способом, но немногие пользователи обладают достаточной технической подготовкой, чтобы сделать это.

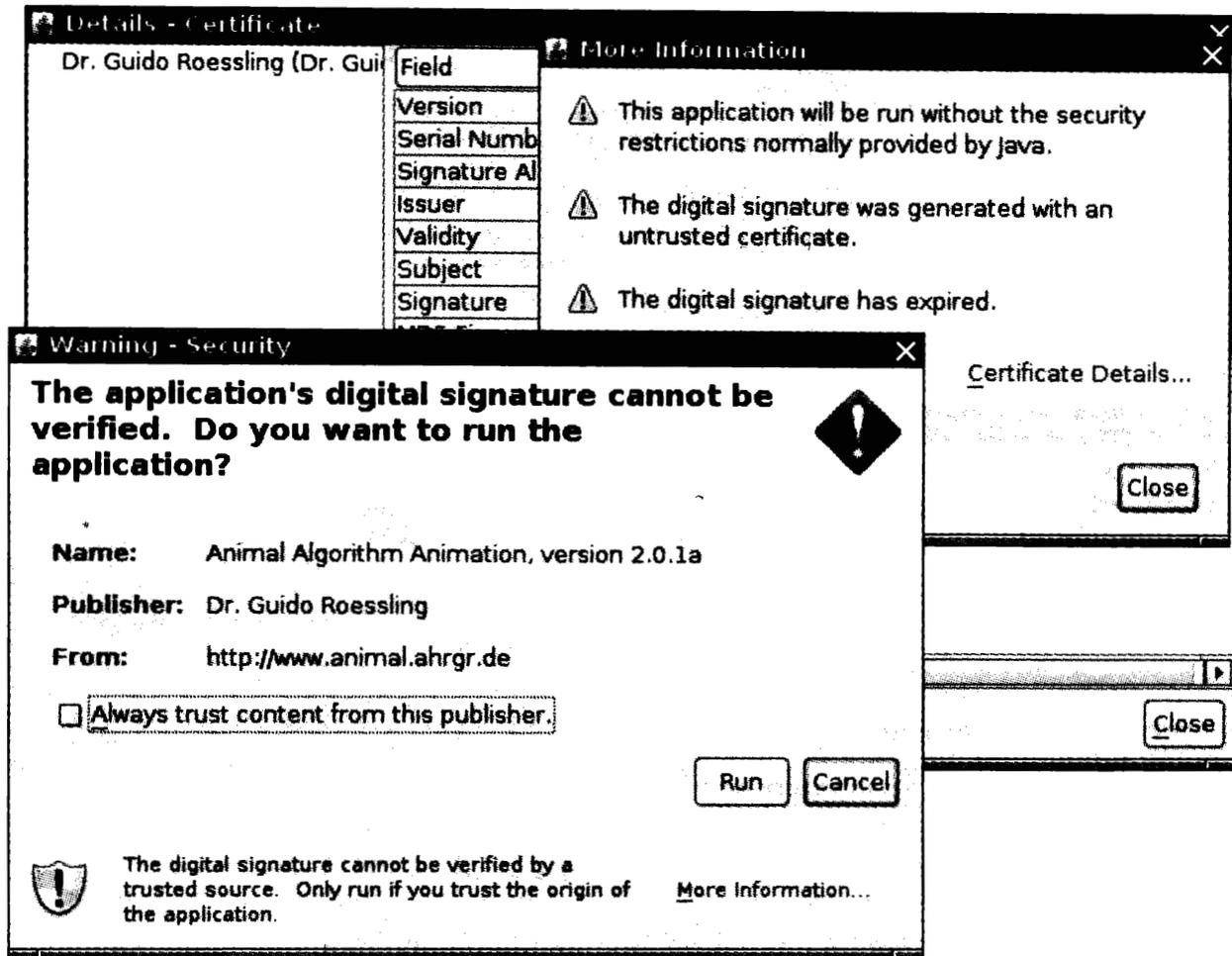


Рис. 10.7. Ненадежный сертификат

Конечно, многие каждый день загружают и запускают приложения из Интернета. Если вы обнаружите, что ваши пользователи доверяют вашим приложениям и вашей веб-инфраструктуре, то действуйте смело, самостоятельно подписывая сертификаты на свои приложения. (Подробнее об этом можно узнать по адресу <http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/developersguide/development.html>). В противном случае обеспечьте безопасность пользователей ваших приложений, не покидая пределы “песочницы”. Используя средства прикладного интерфейса JNLP API (о котором речь пойдет в следующем разделе), вы можете разрешать своей прикладной программе избирательно обращаться к ресурсам с согласия пользователей.

Прикладной интерфейс JNLP API

Прикладной интерфейс JNLP API позволяет неподписанному приложению запускаться на выполнение в “песочнице” и в то же время получать безопасный доступ к локальным ресурсам. В частности, в этом интерфейсе предусмотрены службы для загрузки и сохранения файлов. Приложение не видит файловой системы и не может определить имена файлов. Вместо этого отображается диалоговое окно с файлами, и пользователь приложения выбирает нужный файл. Перед появлением диалогового окна для выбора файлов пользователь получает соответствующее предупреждение и должен дать согласие на продолжение работы (рис. 10.8). Более того, данный прикладной интерфейс на самом деле не предоставляет приложению доступа к объекту типа File. В частности, приложение не может самостоятельно обнаружить файл.

Поэтому программистам предоставляются инструментальные средства для реализации операций открытия и сохранения файлов, но системная информация остается как можно более скрытой от не заслуживающих доверия приложений.

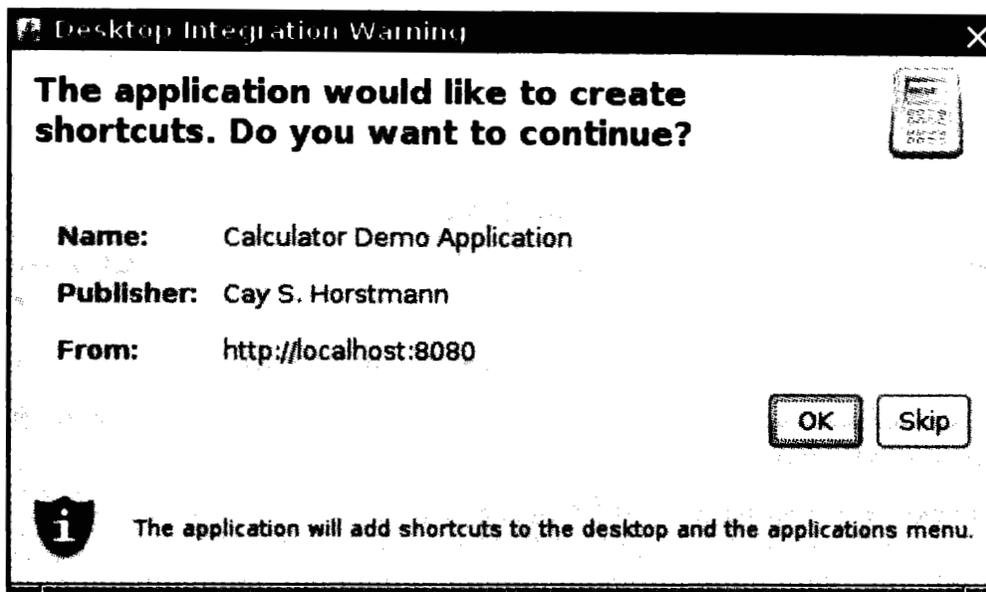


Рис. 10.8. Окно предупреждения о безопасном использовании приложения Java Web Start

Прикладной интерфейс JNLP API предоставляет разработчику следующие возможности.

- Загрузка и сохранение файлов.
- Доступ к буферу обмена.
- Печать.
- Отображение документа в стандартном браузере.
- Хранение и извлечение конфигурационных данных.
- Средства, позволяющие убедиться в том, что выполняется только один экземпляр приложения.

Чтобы воспользоваться этими возможностями, следует обратиться к классу `ServiceManager`, как показано ниже.

```
FileSaveService service =  
    (FileSaveService) ServiceManager.lookup("javax.jnlp.FileSaveService");
```

Если соответствующая служба окажется недоступной, генерируется исключение типа `UnavailableServiceException`.



НА ЗАМЕТКУ! Для компиляции программы, в которой используется прикладной интерфейс JNLP API, необходимо указать файл `javaws.jar` в пути для поиска классов. Этот файл находится в подкаталоге `jre/lib` каталога установки JDK.

А теперь перейдем к рассмотрению наиболее полезных служб прикладного интерфейса JNLP API. Чтобы сохранить файл, следует указать в диалоговом окне

предположения первоначальный путь и расширения файлов, а также сохраняемые данные и предполагаемое имя файла. Рассмотрим следующий пример кода:

```
service.saveFileDialog(".", new String[] { "txt" }, data, "calc.txt");
```

Данные должны быть направлены в поток ввода `InputStream`. Иногда эта задача оказывается довольно сложной. В примере программы, исходный код которой приведен в листинге 10.2, для этой цели применяется описанная ниже методика.

1. Создается экземпляр класса `ByteArrayOutputStream`, предназначенный для хранения байтов, записываемых на диск.
2. Создается экземпляр класса `PrintStream`, направляющий эти данные в поток `ByteArrayOutputStream`.
3. Сохраняемые данные выводятся в поток `PrintStream`.
4. Создается экземпляр класса `ByteArrayInputStream`, вводящий сохраненные данные побайтно.
5. Поток передается методу `saveFileDialog()`.

Более подробно потоки ввода-вывода описываются в главе 1 второго тома данной книги. А до тех пор можете не обращать внимания на особенности обращения с ними в рассматриваемом здесь примере программы.

Для ввода данных из файла имеется служба `FileOpenService`. Ее метод `openFileDialog()` получает первоначальный путь и расширения файлов, а возвращает объект типа `FileContents`. Затем вызывается метод `getInputStream()` и данные вводятся из файла. Если пользователь не выберет файл, метод `openFileDialog()` возвратит пустое значение `null`. Ниже показано, как все это осуществляется непосредственно в коде.

```
FileOpenService service = (FileOpenService) ServiceManager.lookup("javax.  
jnlp.FileOpenService");  
FileContents contents = service.openFileDialog(".", new String[] { "txt" });  
if (contents != null)  
{  
    InputStream in = contents.getInputStream();  
    . . .  
}
```

Следует иметь в виду, что в вашей прикладной программе имя и местоположение файла неизвестно. Но если требуется открыть конкретный файл, то для этой цели можно воспользоваться службой `ExtendedService`:

```
ExtendedService service = (ExtendedService) ServiceManager.lookup("javax.  
jnlp.ExtendedService");  
FileContents contents = service.openFile(new File("c:\\autoexec.bat"));  
if (contents != null)  
{  
    OutputStream out = contents.getOutputStream();  
    . . .  
}
```

Пользователь вашей прикладной программы должен разрешить доступ к файлу, как показано на рис. 10.9.

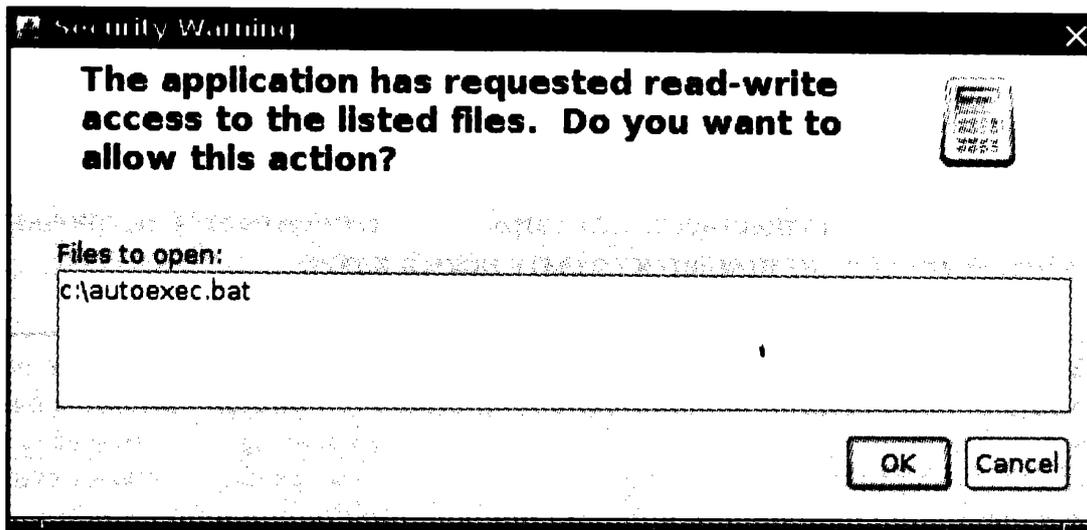


Рис. 10.9. Предупреждение о доступе к файлу

Для того чтобы отобразить документ в стандартном браузере, воспользуйтесь службой `BasicService`, как показано ниже. Но имейте в виду, что в некоторых системах стандартный браузер не установлен.

```
BasicService service =  
    (BasicService) ServiceManager.lookup("javax.jnlp.BasicService");  
if (service.isWebBrowserSupported())  
    service.showDocument(url);  
else . . .
```

Существует устаревшая служба `PersistentService`, позволяющая приложению сохранять небольшие объемы данных о настройках и извлекать ее, если приложение запускается снова. Такая возможность напоминает механизм cookie-файлов, применяемый при обмене данными по протоколу HTTP. В качестве ключей в данной службе постоянного хранилища используются URL. Эти URL не обязательно должны ссылаться на подлинные ресурсы. Они лишь представляют собой удобную иерархическую схему для именования файлов. По каждому URL-ключу приложение может записать произвольные двоичные данные. (Объем записываемых данных может быть ограничен размером их блока.)

Для того чтобы изолировать приложения друг от друга, в каждом из них должны использоваться ключи, начинающиеся с базового веб-адреса (URL), указанного в атрибуте `codebase` из JNLP-файла. Так, если приложение загружено с веб-страницы `http://myserver.com/apps`, то в нем могут использоваться только ключи в форме `http://myserver.com/apps/subkey1/subkey2/...`. Попытка получить доступ к другим ключам окажется заведомо неудачной.

Для определения базового веб-адреса (URL) в приложении может быть вызван метод `getCodebase()` из службы `BasicService`. Новый ключ создается с помощью метода `create()` из службы `PersistenceService` следующим образом:

```
URL url = new URL(codeBase, "mykey");  
service.create(url, maxSize);
```

Для доступа к данным, связанным с конкретным ключом, вызывается метод `get()`. Этот метод возвращает экземпляр класса `FileContents`, с помощью которого можно читать и записывать данные по указанному ключу, как показано в приведенном ниже примере кода.

```
FileContents contents = service.get(url);
InputStream in = contents.getInputStream();
OutputStream out = contents.getOutputStream(true); // true = overwrite
```

К сожалению, удобного способа определить, существует ли указанный ключ или его нужно создать заново, не существует. Вызывая метод `get()`, можно лишь надеяться, что ключ все-таки существует. Если при этом генерируется исключение типа `FileNotFoundException`, то придется создать новый ключ.



НА ЗАМЕТКУ! Как приложения Java Web Start, так и апплеты могут выводить данные на печать, используя обычные средства из прикладного интерфейса API. При этом появляется диалоговое окно, запрашивающее у пользователя согласие на доступ к принтеру. Более подробно средства прикладного интерфейса API для вывода на печать будут рассматриваться в главе 7 второго тома данной книги.

Программа, исходный код которой приведен в листинге 10.2, представляет собой видоизмененный вариант прикладной программы, реализующей калькулятор. Этот калькулятор имеет виртуальную бумажную ленту, на которую записываются результаты всех вычислений. Список всех предыдущих вычислений можно сохранять и вводить из файла. Для демонстрации постоянного хранилища данных в этой прикладной программе допускается задавать заголовок фрейма. Если запустить ее снова, она извлечет этот заголовок из постоянного хранилища (рис. 10.10).

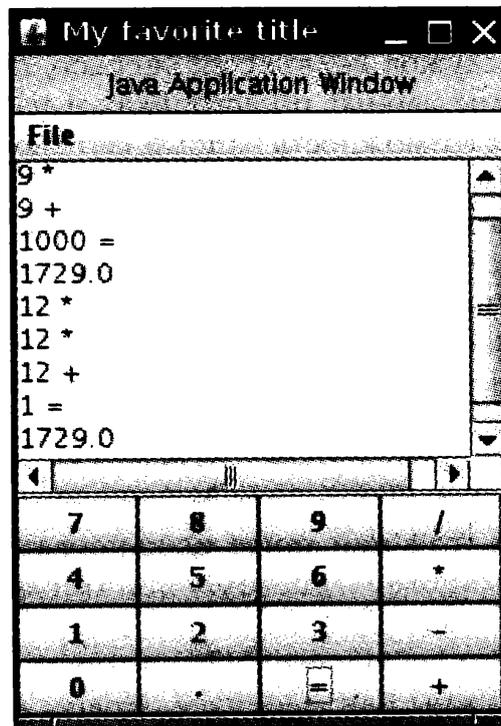


Рис. 10.10. Приложение WebStartCalculator

Листинг 10.2. Исходный код из файла `webstart/CalculatorFrame.java`

```
1 package webstart;
2
3 import java.awt.event.*;
4 import java.beans.*;
5 import java.io.*;
```

```
6 import java.net.*;
7 import javax.jnlp.*;
8 import javax.swing.*;
9 /**
10  * Фрейм с панелью калькулятора и меню для загрузки и
11  * сохранения предыстории калькуляций
12  */
13 public class CalculatorFrame extends JFrame
14 {
15     private CalculatorPanel panel;
16
17     public CalculatorFrame()
18     {
19         setTitle();
20         panel = new CalculatorPanel();
21         add(panel);
22
23         JMenu fileMenu = new JMenu("File");
24         JMenuBar menuBar = new JMenuBar();
25         menuBar.add(fileMenu);
26         setJMenuBar(menuBar);
27
28         JMenuItem openItem = fileMenu.add("Open");
29         openItem.addActionListener(EventHandler.create(
30             ActionListener.class, this, "open"));
31         JMenuItem saveItem = fileMenu.add("Save");
32         saveItem.addActionListener(EventHandler.create(
33             ActionListener.class, this, "save"));
34
35         pack();
36     }
37
38     /**
39     * Получает заголовок из постоянного хранилища или запрашивает
40     * заголовок у пользователя, если он не был прежде сохранен
41     */
42     public void setTitle()
43     {
44         try
45         {
46             String title = null;
47
48             BasicService basic =
49                 (BasicService) ServiceManager.lookup("javax.jnlp.BasicService");
50             URL codeBase = basic.getCodeBase();
51
52             PersistenceService service = (PersistenceService) ServiceManager
53                 .lookup("javax.jnlp.PersistenceService");
54             URL key = new URL(codeBase, "title");
55
56             try
57             {
58                 FileContents contents = service.get(key);
59                 InputStream in = contents.getInputStream();
60                 BufferedReader reader =
61                     new BufferedReader(new InputStreamReader(in));
62                 title = reader.readLine();
63             }
64             catch (FileNotFoundException e)
```

```
65     {
66         title =
67             JOptionPane.showInputDialog("Please supply a frame title:");
68         if (title == null) return;
69
70         service.create(key, 100);
71         FileContents contents = service.get(key);
72         OutputStream out = contents.getOutputStream(true);
73         PrintStream printOut = new PrintStream(out);
74         printOut.print(title);
75     }
76     setTitle(title);
77 }
78 catch (UnavailableServiceException e)
79 {
80     JOptionPane.showMessageDialog(this, e);
81 }
82 catch (MalformedURLException e)
83 {
84     JOptionPane.showMessageDialog(this, e);
85 }
86 catch (IOException e)
87 {
88     JOptionPane.showMessageDialog(this, e);
89 }
90 }
91
92 /**
92  * Открывает файл предыстории и обновляет отображаемые данные
93  */
94 public void open()
95 {
96     try
97     {
98         FileOpenService service = (FileOpenService) ServiceManager
99             .lookup("javax.jnlp.FileOpenService");
100        FileContents contents =
101            service.openFileDialog(".", new String[] { "txt" });
102
103        JOptionPane.showMessageDialog(this, contents.getName());
104        if (contents != null)
105        {
106            InputStream in = contents.getInputStream();
107            BufferedReader reader =
108                new BufferedReader(new InputStreamReader(in));
109            String line;
110            while ((line = reader.readLine()) != null)
111            {
112                panel.append(line);
113                panel.append("\n");
114            }
115        }
116    }
117    catch (UnavailableServiceException e)
118    {
119        JOptionPane.showMessageDialog(this, e);
120    }
121    catch (IOException e)
```

```
122     {
123         JOptionPane.showMessageDialog(this, e);
124     }
125 }
126 /**
127  * Сохраняет предысторию калькуляций в файле
128  */
129 public void save()
130 {
131     try
132     {
133         ByteArrayOutputStream out = new ByteArrayOutputStream();
134         PrintStream printOut = new PrintStream(out);
135         printOut.print(panel.getText());
136         InputStream data = new ByteArrayInputStream(out.toByteArray());
137         FileSaveService service = (FileSaveService) ServiceManager
138             .lookup("javax.jnlp.FileSaveService");
139         service.saveFileDialog(".", new String[]
140             { "txt" }, data, "calc.txt");
141     }
142     catch (UnavailableServiceException e)
143     {
144         JOptionPane.showMessageDialog(this, e);
145     }
146     catch (IOException e)
147     {
148         JOptionPane.showMessageDialog(this, e);
149     }
150 }
```

javax.jnlp.ServiceManager

- `static String[] getServiceNames()`
Возвращает имена всех доступных служб.
- `static Object lookup(String name)`
Возвращает службу по заданному имени.

javax.jnlp.BasicService

- `URL getCodeBase()`
Возвращает базовый веб-адрес или каталог, содержащий код данного приложения.
- `boolean isWebBrowserSupported()`
Возвращает логическое значение `true`, если в среде Java Web Start был запущен веб-браузер.
- `boolean showDocument(URL url)`
Пытается отобразить ресурс по заданному URL в браузере. Если запрос выполнен успешно, возвращает логическое значение `true`.

javax.jnlp.FileContents

- `InputStream getInputStream()`
Возвращает поток ввода для чтения содержимого файла.
- `OutputStream getOutputStream(boolean overwrite)`
Возвращает поток вывода для записи в файл. Если параметр `overwrite` принимает логическое значение `true`, содержимое файла перезаписывается.
- `String getName()`
Возвращает имя файла (но не полный путь к нему).
- `boolean canRead()`
- `boolean canWrite()`
Возвращают логическое значение `true`, если базовый файл можно прочитать или записать.

javax.jnlp.FileOpenService

- `FileContents openFileDialog(String pathHint, String[] extensions)`
- `FileContents[] openMultiFileDialog(String pathHint, String[] extensions)`
Выводят предупреждающее сообщение и диалоговое окно для выбора файлов. Возвращают дескрипторы файлов, выбранных пользователем, или пустое значение `null`, если пользователь не выбрал ни одного файла.

javax.jnlp.FileSaveService

- `FileContents saveFileDialog(String pathHint, String[] extensions, InputStream data, String nameHint)`
- `FileContents saveAsFileDialog(String pathHint, String[] extensions, FileContents data)`
Выводят предупреждающее сообщение и диалоговое окно для выбора файлов. Записывают данные и возвращают дескрипторы файлов, выбранных пользователем, или пустое значение `null`, если пользователь не выбрал ни одного файла.

javax.jnlp.PersistenceService

- `long create(URL key, long maxsize)`
Создает в постоянном хранилище запись по указанному ключу. Возвращает максимальный размер памяти, предоставляемый для записи данных в постоянном хранилище.
- `void delete(URL key)`
Удаляет запись по указанному ключу.

- `String[] getNames(URL url)`

Возвращает относительные имена всех ключей, начинающихся с заданного URL.

- `FileContents get(URL key)`

Получает дескриптор содержимого, с помощью которого можно видоизменить данные, связанные с указанным ключом. Если по указанному ключу ничего не найдено, генерируется исключение типа `FileNotFoundException`.

Апплеты

Апплеты — это прикладные программы на Java, включаемые в состав HTML-страниц. Сама HTML-страница должна сообщать браузеру, какой именно апплет следует загрузить и где разместить каждый апплет на странице. Как и следовало ожидать, для применения апплета требуется специальный дескриптор, который сообщает браузеру, откуда взять файлы соответствующих классов и каким образом апплет располагается на веб-странице (размер, местоположение и т.п.). Затем браузер загружает нужные файлы классов из Интернета (или из каталога на пользовательской машине) и автоматически запускает апплет на выполнение.

Когда появились апплеты, для просмотра веб-страниц приходилось пользоваться браузером HotJava, специально созданным компанией Sun Microsystems для поддержки апплетов. Популярность апплетов существенно возросла, когда компания Netscape включила виртуальную машину Java в свой браузер. Вскоре такая возможность появилась и в браузере Internet Explorer. К сожалению, не обошлось и без трудностей. Компания Netscape не стала поддерживать больше современные версии Java, а корпорация Microsoft склонялась к решению совсем отказаться от Java.

Для преодоления подобных трудностей было разработано инструментальное средство под названием Java Plug-In. Используя различные механизмы расширения возможностей браузеров, это инструментальное средство можно совершенно свободно встроить в наиболее распространенные разновидности браузеров, чтобы выполнять апплеты во внешней среде исполнения Java.



НА ЗАМЕТКУ! Чтобы запустить апплеты, исходный код которых приведен далее в этой главе, необходимо установить в своем браузере текущую версию подключаемого модуля Java Plug-In и убедиться в том, что браузер связан с ним. Подробнее о загрузке и настройке Java Plug-In можно узнать по адресу <http://java.com>.

Простой апплет

По традиции начнем с простейшего апплета, переделав упоминавшуюся ранее программу NotHelloWord на апплет. В рамках этой книги для реализации апплетов будет использоваться библиотека Swing. Все рассматриваемые здесь и далее апплеты будут расширять класс `JApplet`, являющийся суперклассом для всех апплетов, создаваемых средствами библиотеки Swing. Как показано на рис. 10.11, класс `JApplet` является производным от обычного класса `Applet`.

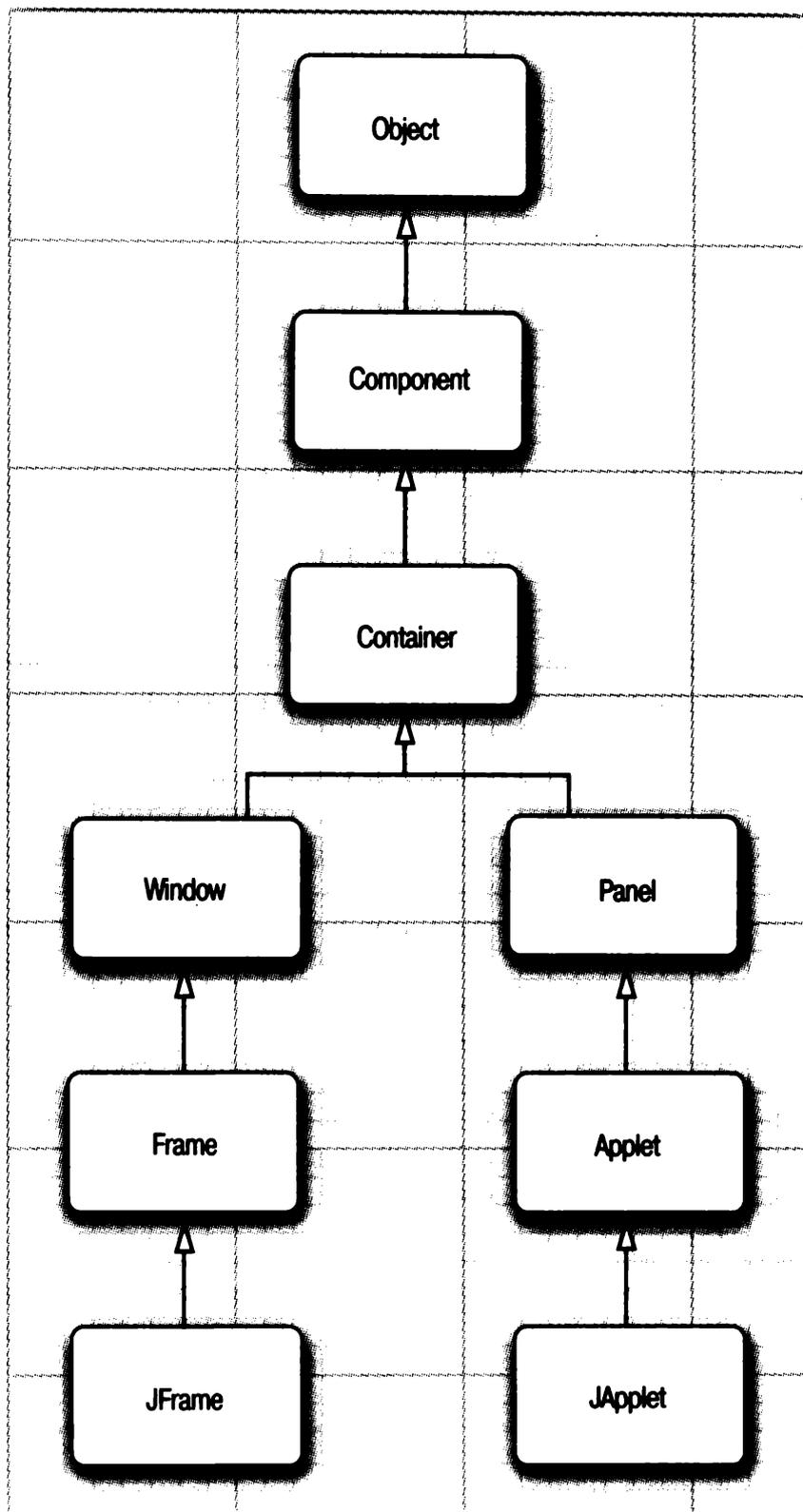


Рис. 10.11. Иерархия наследования классов апплетов



НА ЗАМЕТКУ! Если апплет содержит компоненты библиотеки Swing, следует расширить класс `JApplet`. Компоненты Swing не будут правильно отображаться, находясь в объекте обычного класса `Applet`.

В листинге 10.3 приведен исходный код апплета, выполняющего те же функции, что и программа NotHelloWord. Обратите внимание на сходство этого апплета с программой из главы 7. Но поскольку апплет существует на веб-странице, то и нет никакой нужды указывать специальный метод для выхода из апплета.

Листинг 10.3. Исходный код из файла applet/NotHelloWorld.java

```
1 package applet;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * @version 1.23 2012-05-14
8  * @author Cay Horstmann
9  */
10 public class NotHelloWorld extends JApplet
11 {
12     public void init()
13     {
14         EventQueue.invokeLater(new Runnable()
15         {
16             public void run()
17             {
18                 JLabel label = new JLabel("Not a Hello, World applet",
19                     SwingConstants.CENTER);
20                 add(label);
21             }
22         });
23     }
24 }
```

Для того чтобы подготовить апплет к запуску в браузере, выполните следующие действия.

1. Скомпилируйте исходные файлы Java в файлы классов.
2. Создайте HTML-файл, сообщающий браузеру, какой именно файл класса следует загрузить первым и какие размеры имеет апплет.

Ниже приведено содержимое этого HTML-файла.

```
<applet code="applet/NotHelloWorld.class" width="300" height="300">
</applet>
```

Прежде чем просматривать апплет в окне браузера, желательно проверить его с помощью утилиты `appletviewer`, специально предназначенной для просмотра апплетов. Эта утилита входит в состав JDK. Чтобы проверить апплет с помощью утилиты `appletviewer`, выполните следующую команду:

```
appletviewer NotHelloWorldApplet.html
```

В командной строке указывается имя HTML-файла, а не файла класса. На рис. 10.12 показано окно утилиты `appletviewer`, в котором отображается апплет `NotHelloWorldApplet`.

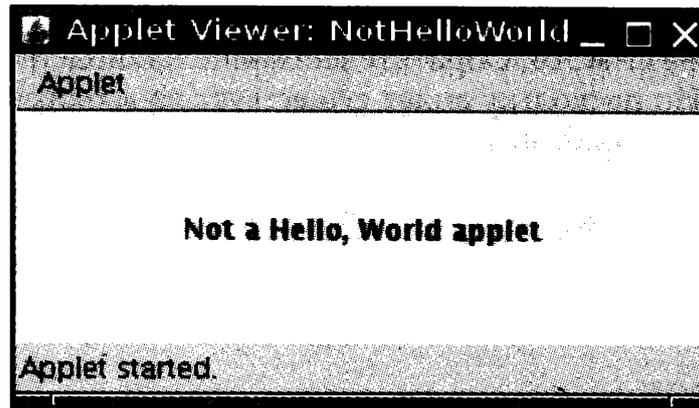


Рис. 10.12. Вид апплета NotHelloWorldApplet в программе просмотра апплетов



СОВЕТ. Апплеты можно также запускать на выполнение, находясь в текстовом редакторе или в ИСР. Так, в редакторе Emacs для этого достаточно выбрать команду меню JDE⇒Run Applet (JDE⇒Запустить апплет), а в ИСР Eclipse — команду меню Run⇒Run as⇒Java Applet (Запустить⇒Запустить как⇒Апплет Java).

Утилитой `appletviewer` удобно пользоваться на первых этапах тестирования апплетов, но затем все равно наступает момент, когда требуется запустить разработанный апплет в браузере, чтобы почувствовать себя на месте пользователя. В частности, утилита `appletviewer` показывает на экране только апплет без окружающего текста. Если HTML-файл содержит ряд дескрипторов `<applet>`, то утилита `appletviewer` откроет несколько окон для всех указанных апплетов.

Для того чтобы просмотреть апплет надлежащим образом, загрузите HTML-файл в браузер (рис. 10.13). Если апплет не отображается, то, скорее всего, в браузере используется встроенная виртуальная машина Java. Чтобы разрешить это затруднение, вам придется установить подключаемый модуль Java Plug-In и настроить браузер на взаимодействие с ним.

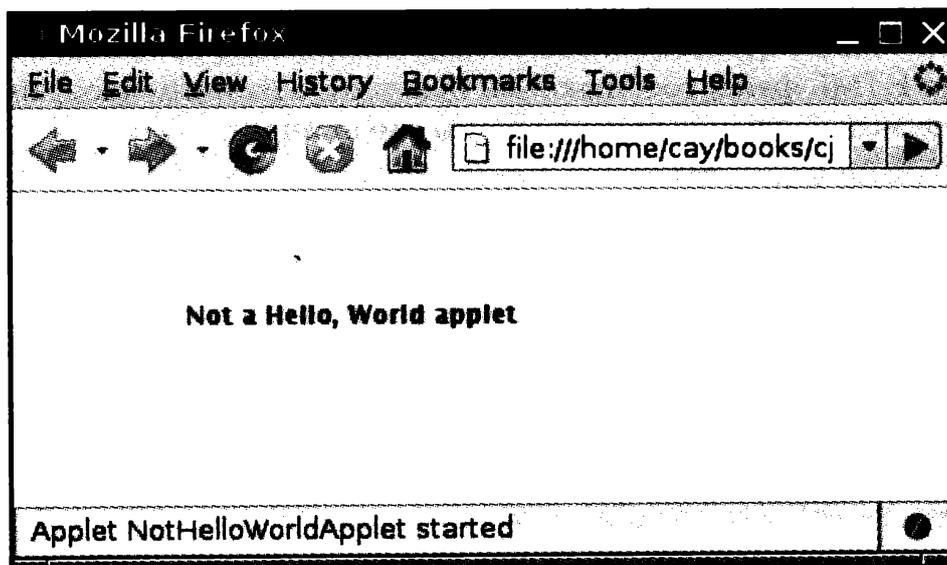


Рис. 10.13. Просмотр апплета в браузере



СОВЕТ. Если вы внесли изменения в апплет и скомпилировали его снова, вам придется перезапустить браузер, чтобы он смог загрузить новые файлы классов. Простое обновление HTML-страницы не приводит к загрузке нового кода. Утомительной процедуры повторного запуска браузера можно избежать, открыв консоль Java и введя команду `ж`, которая очищает кеш загрузчика классов. После этого вы можете повторно открывать HTML-файл, и всякий раз будет использоваться новый код апплета. Если вы работаете в Windows, щелкните на пиктограмме Java Plug-In, доступной на панели управления, а в Linux воспользуйтесь командой `jcontrol` и запросите вывод на экран консоли Java. Консоль будет отображаться при каждой загрузке апплета.

Преобразование приложения в апплеты

Приложение с ГПИ нетрудно преобразовать в апплет и встроить в веб-страницу. Примечательно, что весь код пользовательского интерфейса можно при этом оставить без изменений. Итак, для преобразования приложения в апплет выполните следующие действия.

1. Создайте HTML-страницу с соответствующим дескриптором для загрузки кода апплета.
2. Определите подкласс, производный от класса `JApplet`. Этот класс должен быть открытым (`public`), иначе апплет нельзя будет загрузить.
3. Удалите из приложения метод `main()`. Окно фрейма для приложения создавать не следует, поскольку оно будет отображаться в окне браузера.
4. Перенесите все операторы инициализации из конструктора окна фрейма в метод апплета `init()`. Специально создавать объект для апплета не обязательно — браузер сам создаст его и вызовет метод `init()`.
5. Удалите вызов метода `setSize()`. В апплетах размеры задаются в HTML-файле с помощью атрибутов `width` и `height`.
6. Удалите вызов метода `setDefaultCloseOperation()`. Апплет нельзя закрыть; он прекращает работу вместе с браузером.
7. Если в приложении имеется вызов метода `setTitle()`, его следует удалить, поскольку заголовки у апплетов отсутствуют. (Разумеется, можно указать заголовок самой веб-страницы, используя HTML-дескриптор `<title>`.)
8. Не вызывайте метод `setVisible(true)`. Апплет отображается на экране автоматически.

`java.applet.Applet 1.0`

- `void init()`
Вызывается при первой загрузке апплета. Переопределите этот метод и разместите в нем весь код, выполняющий инициализацию.
- `void start()`
Переопределите этот метод, чтобы он содержал весь код, который должен выполняться **всякий раз**, когда пользователь открывает в окне браузера веб-страницу, содержащую данный апплет. Обычно этот метод повторно активизирует поток.

- `void stop()`
Переопределите этот метод, чтобы он содержал весь код, который должен выполняться *всякий раз*, когда пользователь закрывает в окне браузера веб-страницу, содержащую данный апплет. Обычно этот метод завершает поток.
- `void destroy()`
Переопределите этот метод, чтобы он содержал весь код, который должен выполняться *всякий раз*, когда пользователь выходит из браузера.
- `void resize(int width, int height)`
Задаёт новые размеры апплета. Если бы этот метод нормально взаимодействовал с веб-страницами, он был бы очень удобен. К сожалению, с современными браузерами этот метод не работает, поскольку он конфликтует с их механизмами компоновки веб-страниц.

HTML-дескриптор `<applet>` и его атрибуты

Дескриптор `<applet>` в своей самой элементарной форме выглядит приблизительно так:

```
<applet code="applet/NotHelloWorld.class" width="300" height="100">
```

Как упоминалось ранее, атрибут `code` задаёт имя класса. В нём указывается значение, которое представляет собой имя файла, оканчивающееся расширением `.class`. Атрибуты `width` и `height` определяют размеры окна, в котором будет размещаться апплет. Значения в обоих атрибутах задаются в пикселях. Следует не забыть также указать закрывающий дескриптор `</applet>`. Текст, заключённый между дескрипторами `<applet>` и `</applet>`, выводится на экран в том случае, если браузер не в состоянии отобразить апплет. Атрибуты `code`, `width` и `height` обязательны. Если хотя бы один из них будет пропущен, браузер не сможет загрузить апплет.

Вся эта информация обычно встраивается в HTML-страницу и в самом минимальном виде может выглядеть так, как показано ниже.

```
<html>
  <head>
    <title>NotHelloWorldApplet</title>
  </head>
  <body>
    <p>Следующая текстовая строка выводится под управлением системы
Java:</p>
    <applet code="applet/NotHelloWorld.class" width="100" height="100">
      Если в браузере поддерживает Java, то здесь будет отображен апплет
    </applet>
  </body>
</html>
```

Вместе с дескриптором `<applet>` можно использовать описанные ниже атрибуты.

- `width`, `height`
Эти атрибуты обязательны. Они задают ширину и высоту окна апплета в пикселях. В программе `appletviewer` эти параметры определяют первоначальные размеры апплета. Размеры любого окна, созданного утилитой `appletviewer`, можно изменить. А в браузере изменить размеры апплета *нельзя*. Поэтому,

включая апплет в состав HTML-документа, следует заранее продумать, сколько места ему понадобится.

- align

Этот атрибут задает вид выравнивания апплета. Значения этого атрибута те же самые, что и у атрибута align дескриптора ``.

- vspace, hspace

Эти необязательные атрибуты обозначают количество пикселей выше и ниже апплета (vspace), а также по обеим сторонам от него (hspace).

- code

Этот атрибут задает имя файла, содержащего класс апплета. Имя берется относительно пути, заданного в атрибуте `codebase`, или относительно текущей страницы, если атрибут `codebase` не указан. Так, если класс апплета находится в пакете `com.mycompany`, задается атрибут `code="com/mycompany/MyApplet.class"` или `code="com.mycompany.MyApplet.class"`. Но указывать абсолютные пути в этом атрибуте нельзя. Если же файл класса находится в каком-нибудь другом месте, то путь к нему следует указать в атрибуте `codebase`.

Атрибут `code` задает только имя файла, содержащего класс апплета. Разумеется, апплет может содержать и файлы других классов. Загружая класс, содержащий апплет, браузер выясняет, какие еще нужны файлы, и загружает их дополнительно. При разметке апплета в HTML-файле следует непременно задать атрибут `code` или же рассматриваемый далее атрибут `object`.

- codebase

Этот необязательный атрибут уведомляет браузер о том, что файлы классов находятся в каталоге по базовому веб-адресу (URL), указанному в виде значения этого атрибута. По желанию можно указать даже абсолютный URL каталога на другом сервере. Но чаще всего задается относительный URL. Допустим, структура каталогов выглядит так, как показано ниже.

- aDirectory/

```
|__MyPage.html
|__myApplets/
    |__CalculatorApplet.class
```

В таком случае в документе `MyPage.html` может присутствовать следующий дескриптор:

```
<applet code="MyApplet.class" codebase="myApplets" width="100" height="150">
```

- archive

Этот необязательный атрибут перечисляет архивные файлы Java или файлы, содержащие классы и другие ресурсы для апплета. Такие файлы загружаются с сервера перед загрузкой самого апплета. Подобная технология значительно ускоряет работу, поскольку для загрузки JAR-файла, содержащего много мелких файлов, необходим лишь один HTTP-запрос. В списке JAR-файлы разделяются запятыми, как показано в приведенном ниже примере.

```
<applet code="MyApplet.class"
  archive="MyClasses.jar,corejava/CoreJavaClasses.jar"
  width="100" height="150">
```

- object

Этот атрибут позволяет задать имя файла, содержащего сериализованный объект апплета. (Объект называется *сериализованным*, если все его поля данных записаны в отдельном файле. Сериализация подробнее рассматривается в главе 1 второго тома данной книги.) Перед отображением апплета объект десериализуется. Если в дескрипторе присутствует этот атрибут, метод `init()` не вызывается, а выполнение апплета начинается с метода `start()`. Перед сериализацией объекта апплета необходимо вызвать его метод `stop()`. Благодаря сериализации появляется возможность реализовывать браузеры, сохраняющие состояние, а после повторной загрузки восстанавливающие то состояние апплета, которое было в момент завершения предыдущего сеанса работы. К сожалению, эта возможность практически не используется разработчиками веб-страниц. В каждом дескрипторе `<applet>` должен быть непременно задан атрибут `code` или `object`, как показано в приведенном ниже примере.

```
<applet object="MyApplet.ser" width="100" height="150">
```

- name

В атрибуте `name` апплету присваивается имя, которое затем можно использовать в сценарии. Браузеры Netscape и Internet Explorer позволяют вызывать методы апплета, расположенного на веб-странице, языковыми средствами JavaScript.

Для доступа к апплету из сценария на JavaScript нужно прежде всего задать его имя следующим образом:

```
<applet code="MyApplet.class" width="100" height="150" name="mine">
</applet>
```

- И тогда к этому объекту можно обращаться по ссылке `document.applets.имя_апплета`, как показано в приведенном ниже примере.

```
var myApplet = document.applets.mine;
```

Используя преимуществ интеграции языков Java и JavaScript, поддерживаемой в браузерах Netscape и Internet Explorer, методы апплета можно вызывать следующим образом:

```
myApplet.init();
```

Атрибут `name` приобретает большое значение, если требуется разместить два взаимодействующих апплета на одной и той же веб-странице. Сначала нужно присвоить имя каждому апплету, а затем передать символьную строку с именем апплета методу `getApplet()` из класса `AppletComplex`. Механизм взаимодействия апплетов обсуждается далее в этой главе.



НА ЗАМЕТКУ! В документе, доступном по адресу <http://www.javaworld.com/javatips/jw-javatip80.html>, описывается, каким образом средства взаимодействия кода JavaScript и Java могут быть использованы для решения классической задачи: изменения размеров окна апплета, жестко заданных в атрибутах `width` и `height`. Это удачный пример интеграции Java и JavaScript.

- alt

В браузерах некоторых пользователей поддержка Java может быть отключена — возможно, системным администратором ради перестраховки. В таком

случае для вывода сообщения таким пользователям можно воспользоваться атрибутом `alt` следующим образом:

```
<applet code="MyApplet.class" width="100" height="150"  
  alt="Если бы в вашем браузере была активизирована поддержка Java,  
  вы бы увидели здесь мой апплет">
```

Если браузер вообще не в состоянии обрабатывать апплеты, он игнорирует неизвестные ему дескрипторы `<applet>` и `<param>`. В таком случае весь текст, заключенный между дескрипторами `<applet>` и `</applet>`, отображается браузером. А браузеры, поддерживающие Java, не отображают текст, заключенный между дескрипторами `<applet>` и `</applet>`. В этих дескрипторах можно отображать сообщения для тех, кто до сих пор применяет доисторические браузеры, как показано в приведенном ниже примере.

```
<applet code="MyApplet.class" width="100" height="150">  
  Если бы ваш браузер поддерживал программы на Java,  
  вы бы увидели здесь мой апплет.  
</applet>
```

Дескриптор `<object>`

Дескриптор `<object>` определен в спецификации HTML 4.0. Он предложен консорциумом W3C взамен дескриптора `<applet>`. Различные атрибуты, определяющие размеры и местоположение апплета, например `align` и `height`, ничем не отличаются от таких же атрибутов дескриптора `<applet>`. Ключевым для апплетов Java в дескрипторе `<object>` является атрибут `classid`. Этот атрибут обозначает местоположение объекта апплета. Помимо апплетов, с помощью дескриптора `<object>` можно загружать и другие объекты, например, компоненты ActiveX или подключаемый модуль Java Plug-in. Тип загружаемого объекта указывается с атрибуте `codetype`. Апплет Java имеет тип `application/java`. Для его загрузки нужно задать в дескрипторе `<object>` следующие атрибуты:

```
<object  
  codetype="application/java"  
  classid="java:MyApplet.class"  
  width="100" height="150">
```

Следует иметь в виду, что после атрибута `classid` может следовать атрибут `codebase`, точно так же, как это происходит в дескрипторе `<applet>`.

Передача данных апплетам через параметры

Подобно тому, как в приложениях воспринимаются данные, введенные в командной строке, в апплетах могут использоваться параметры, заданные в HTML-файле. Для этой цели служит HTML-дескриптор `<param>`. Допустим, на веб-странице требуется определить стиль шрифта, чтобы применить его в апплете. Для этого необходимо разметить следующие HTML-дескрипторы:

```
<applet code="FontParamApplet.class" width="200" height="200">  
  <param name="font" value="Helvetica"/>  
</applet>
```

В этом случае апплет получает значение параметра с помощью метода `getParameter()`, определенного в классе `Applet`:

```
public class FontParamApplet extends JApplet  
{
```

```
public void init()
{
    String fontName = getParameter("font");
    . . .
}
. . .
}
```



НА ЗАМЕТКУ! Метод `getParameter()` можно вызывать только в методе `init()` апплета, а не в его конструкторе. Ведь когда выполняется конструктор апплета, параметры еще не подготовлены. Компоновка большинства нетривиальных апплетов определяется параметрами, и поэтому снабжать апплеты конструкторами не рекомендуется. Вместо этого весь код инициализации апплета лучше разместить в методе `init()`.

Параметры всегда возвращаются в виде символьных строк. Поэтому символьные строки нужно преобразовать в числовой тип, если они применяются именно для этого. Это делается стандартным способом с помощью соответствующего метода, как, например, `parseInt()` из класса `Integer`. Так, если бы требовалось добавить параметр размера шрифта, можно было бы написать следующий код HTML-разметки:

```
<applet code="FontParamApplet.class" width="200" height="200">
  <param name="font" value="Helvetica"/>
  <param name="size" value="24"/>
</applet>
```

А в приведенном ниже фрагменте кода показано, каким образом считывается целочисленный параметр.

```
public class FontParamApplet extends JApplet
{
    public void init()
    {
        String fontName = getParameter("font");
        int fontSize = Integer.parseInt(getParameter("size"));
        . . .
    }
}
```



НА ЗАМЕТКУ! При совпадении значений атрибута `name` в дескрипторе `<param>` и параметра в методе `getParameter()` сравнение осуществляется без учета регистра.

Кроме того, для гарантии правильности заданных параметров следует проверить, не пропущено ли значение параметра `size`. Для этого достаточно сравнить его с пустым значением `null`, как показано в приведенном ниже примере кода.

```
int fontsize;
String sizeString = getParameter("size");
if (sizeString == null) fontSize = 12;
else fontSize = Integer.parseInt(sizeString);
```

Рассмотрим пример апплета, в котором широко применяются параметры. Этот апплет рисует на экране гистограмму, показанную на рис. 10.14. В качестве параметров этот апплет получает метки и величины высоты столбиков гистограммы.

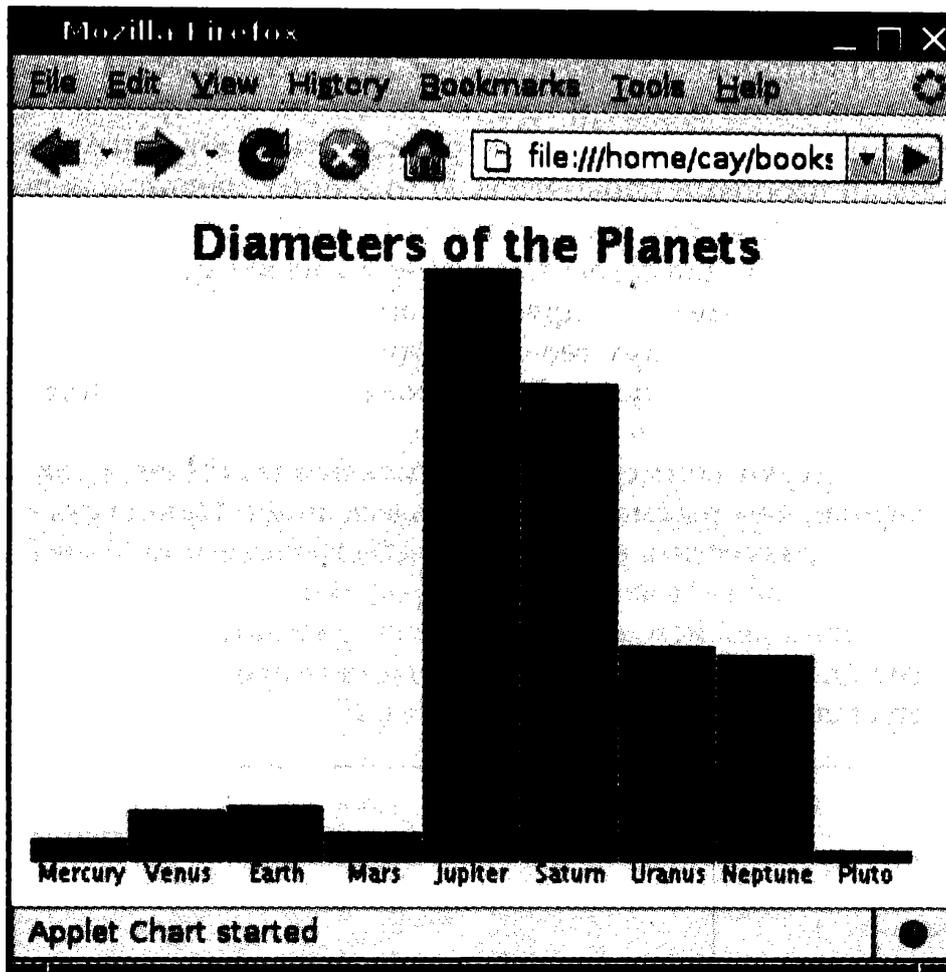


Рис. 10.14. Апплет, рисующий гистограмму

Требуемые значения задаются в HTML-файле с помощью дескриптора `<param>`. Ниже показано, как выглядит HTML-файл разметки гистограммы, приведенной на рис. 10.14.

```
<applet code="Chart.class" width="400" height="300">
  <param name="title" value="Diameters of the Planets"/>
  <param name="values" value="9"/>
  <param name="name.1" value="Mercury"/>
  <param name="name.2" value="Venus"/>
  <param name="name.3" value="Earth"/>
  <param name="name.4" value="Mars"/>
  <param name="name.5" value="Jupiter"/>
  <param name="name.6" value="Saturn"/>
  <param name="name.7" value="Uranus"/>
  <param name="name.8" value="Neptune"/>
  <param name="name.9" value="Pluto"/>
  <param name="value.1" value="3100"/>
  <param name="value.2" value="7500"/>
  <param name="value.3" value="8000"/>
  <param name="value.4" value="4200"/>
  <param name="value.5" value="88000"/>
  <param name="value.6" value="71000"/>
  <param name="value.7" value="32000"/>
  <param name="value.8" value="30600"/>
  <param name="value.9" value="1430"/>
</applet>
```

Эти значения можно было бы задать в виде массива символьных строк и массива чисел, но у дескрипторов `<param>` имеются два преимущества. Во-первых, можно создать несколько копий одного и того же апплета на веб-странице, демонстрируя разные гистограммы. Для этого достаточно разместить на странице два дескриптора `<applet>` с разными наборами параметров. И во-вторых, данные для построения гистограмм можно изменить. Эти преимущества не проявляются в рассматриваемом здесь примере, поскольку диаметры планет не изменяются. Но допустим, что на веб-странице требуется разместить другую гистограмму, отражающую объем еженедельных продаж. Изменить саму веб-страницу нетрудно, поскольку она содержит простой текст. А вот править и перекомпилировать файл с исходным кодом на Java каждую неделю очень неудобно и утомительно.

Разумеется, существуют коммерческие компоненты JavaBeans, которые строят более сложные графики, чем рассматриваемый здесь апплет. Приобретя один из таких компонентов, можно разместить его на своей веб-странице и пользоваться им, даже не зная, каким образом апплет воспроизводит графики.

В листинге 10.4 приведен исходный код апплета, рассматриваемого здесь в качестве примера. Следует заметить, что параметры апплета считываются в методе `init()`, а гистограмма рисуется в методе `paintComponent()`.

Листинг 10.4. Исходный код из файла `chart/Chart.java`

```
1 package chart;
2
3 import java.awt.*;
4 import java.awt.font.*;
5 import java.awt.geom.*;
6 import javax.swing.*;
7
8 /**
9  * @version 1.33 2007-06-12
10  * @author Cay Horstmann
11  */
12 public class Chart extends JApplet
13 {
14     public void init()
15     {
16         EventQueue.invokeLater(new Runnable()
17         {
18             public void run()
19             {
20                 String v = getParameter("values");
21                 if (v == null) return;
22                 int n = Integer.parseInt(v);
23                 double[] values = new double[n];
24                 String[] names = new String[n];
25                 for (int i = 0; i < n; i++)
26                 {
27                     values[i] =
28                         Double.parseDouble(getParameter("value." + (i + 1)));
29                     names[i] = getParameter("name." + (i + 1));
30                 }
31
32                 add(new ChartComponent(values, names, getParameter("title")));
33             }
34         });
35     }
36 }
```

```
34     });
35   }
36 }
37
38 /**
39  * Компонент, рисующий гистограмму
40  */
41 class ChartComponent extends JComponent
42 {
43     private double[] values;
44     private String[] names;
45     private String title;
46
47     /**
48      * Конструирует объект типа ChartComponent
49      * @param v Массив значений для построения гистограммы
50      * @param n Массив меток гистограммы
51      * @param t Заголовок гистограммы
52      */
53     public ChartComponent(double[] v, String[] n, String t)
54     {
55         values = v;
56         names = n;
57         title = t;
58     }
59
60     public void paintComponent(Graphics g)
61     {
62         Graphics2D g2 = (Graphics2D) g;
63
64         // вычислить минимальное и максимальное значения
65         if (values == null) return;
66         double minValue = 0;
67         double maxValue = 0;
68         for (double v : values)
69         {
70             if (minValue > v) minValue = v;
71             if (maxValue < v) maxValue = v;
72         }
73         if (maxValue == minValue) return;
74         int panelWidth = getWidth();
75         int panelHeight = getHeight();
76
77         Font titleFont = new Font("SansSerif", Font.BOLD, 20);
78         Font labelFont = new Font("SansSerif", Font.PLAIN, 10);
79
80         // вычислить протяженность заголовка
81         FontRenderContext context = g2.getFontRenderContext();
82         Rectangle2D titleBounds = titleFont.getStringBounds(title, context);
83         double titleWidth = titleBounds.getWidth();
84         double top = titleBounds.getHeight();
85
86         // воспроизвести заголовок
87         double y = -titleBounds.getY(); // ascent
88         double x = (panelWidth - titleWidth) / 2;
89         g2.setFont(titleFont);
90         g2.drawString(title, (float) x, (float) y);
```

```
91
92 // вычислить протяженность меток гистограммы
93 LineMetrics labelMetrics = labelFont.getLineMetrics("", context);
94 double bottom = labelMetrics.getHeight();
95
96 y = panelHeight - labelMetrics.getDescent();
97 g2.setFont(labelFont);
98
99 // получить масштабный коэффициент и ширину столбиков гистограммы
100 double scale = (panelHeight - top - bottom) / (maxValue - minValue);
101 int barWidth = panelWidth / values.length;
102
103 // нарисовать столбики гистограммы
104 for (int i = 0; i < values.length; i++)
105 {
106     // получить координаты прямоугольного столбика гистограммы
107     double x1 = i * barWidth + 1;
108     double y1 = top;
109     double height = values[i] * scale;
110     if (values[i] >= 0) y1 += (maxValue - values[i]) * scale;
111     else
112     {
113         y1 += maxValue * scale;
114         height = -height;
115     }
116
117     // заполнить столбик гистограммы и обвести его контуром
118     Rectangle2D rect =
119         new Rectangle2D.Double(x1, y1, barWidth - 2, height);
120     g2.setPaint(Color.RED);
121     g2.fill(rect);
122     g2.setPaint(Color.BLACK);
123     g2.draw(rect);
124     // воспроизвести отцентрованную метку под столбиком гистограммы
125     Rectangle2D labelBounds =
126         labelFont.getStringBounds(names[i], context);
127
128     double labelWidth = labelBounds.getWidth();
129     x = x1 + (barWidth - labelWidth) / 2;
130     g2.drawString(names[i], (float) x, (float) y);
131 }
132 }
133 }
```

java.applet.Applet 1.0

- `public String getParameter(String name)`
Получает параметры, которые задаются в дескрипторе `<param>` разметки веб-страницы, загружающей апплет. Строковое значение параметра `name` указывается с учетом регистра.
- `public String getAppletInfo()`
Возвращает символьную строку, содержащую сведения об авторе, версии и авторских правах на данный апплет. Для указания этих сведений данный метод обычно переопределяется.

- `public String[][] getParameterInfo()`

Этот метод переопределяется таким образом, чтобы возвращать двумерный массив параметров из дескриптора `<param>`, поддерживаемых данным апплетом. Каждый ряд такого массива состоит из трех элементов: имени, типа и описания параметра. Ниже в качестве примера приведен результат, возвращаемый данным методом.

```
"fps", "1-10", "frames per second"  
"repeat", "boolean", "repeat image loop?"  
"images", "url", "directory containing images"
```

Обращение к файлам изображений и звуковым файлам

Апплеты способны обращаться как с изображением, так и со звуком. Как известно, файлы изображений должны быть представлены в формате GIF, PNG или JPEG, а звуковые файлы — в формате AU, AIFF, WAV или MIDI. Имеется также возможность создавать анимационные последовательности изображений формата GIF для воспроизведения элементарной анимации.

Местоположение файлов изображений и звуковых файлов указывается по относительным URL. Базовый URL обычно получается при вызове методов `getDocumentBase()` или `getCodeBase()`. Первый из них получает URL HTML-страницы, где находится апплет, а второй — URL каталога с апплетом, указанным в атрибуте `codebase` разметки апплета.



НА ЗАМЕТКУ! В предыдущих версиях Java SE при описании обоих методов была допущена ошибка (зарегистрированная под номером #4456393; <http://bugs.sun.com/bugdatabase/index.jsp>). Эта ошибка была исправлена в версии Java SE 5.0.

Файлы изображений и звуковые файлы можно получать с помощью методов `getImage()` и `getAudioClip()`, указав базовый URL их местоположения, как показано ниже.

```
Image cat = getImage(getCodeBase(), "images/cat.gif");  
AudioClip meow = getAudioClip(getCodeBase(), "audio/meow.au");
```

В главе 7 было показано, каким образом на экран выводится изображение. А для того чтобы воспроизвести звукозапись, достаточно вызвать метод `play()`, как показано в приведенном ниже примере. Такой же метод можно вызвать из класса `Applet`, не загружая предварительно звукозапись:

```
play(getCodeBase(), "audio/meow.au");
```

java.applet.Applet 1.0

- URL `getDocumentBase()`
Возвращает URL веб-страницы, содержащей апплет.
- URL `getCodeBase()`
Возвращает URL каталога, содержащего апплет. Это может быть абсолютный URL, ссылка на каталог, указываемый с помощью атрибута `codebase`, а если данный атрибут отсутствует, то ссылка на каталог, в котором находится HTML-файл.

- `void play(URL url)`
- `void play(URL url, String name)`

Первый метод воспроизводит звуковые файлы, расположенные по указанному URL, а второй интерпретирует строку как адрес, определяемый относительно URL, заданного в первом параметре. Если звуковой файл не найден, то никакие действия не выполняются.

- `AudioClip getAudioClip(URL url)`
- `AudioClip getAudioClip(URL url, String name)`

Первый метод получает звукозапись, расположенную по заданному URL, а второй интерпретирует символьную строку как адрес, определяемый относительно URL, заданного в первом параметре. Если звуковой файл не найден, оба метода возвращают пустое значение `null`.

- `Image getImage(URL url)`
- `Image getImage(URL url, String name)`

Возвращают объект типа `Image`, инкапсулирующий графическое изображение, расположенное по указанному URL. Если изображение по указанному адресу отсутствует, немедленно возвращается пустое значение `null`. В противном случае запускается отдельный поток для загрузки изображения.

Контекст апплета

Апплет запускается на выполнение браузером или утилитой `appletviewer`. Апплет может поручить браузеру выполнить некоторые действия, например, извлечь звукозапись, вывести короткое сообщение в строке состояния или отобразить другую веб-страницу. Браузер или другая программа, загрузившая апплет, может удовлетворить его запросы или проигнорировать их. Так, если апплет выполняется в среде `appletviewer`, его требование отобразить веб-страницу не будет удовлетворено.

Для взаимодействия с браузером апплет вызывает метод `getAppletContext()`. Этот метод возвращает объект, реализующий интерфейс `AppletContext`. Конкретную реализацию интерфейса `AppletContext` можно считать средством взаимодействия апплета и загрузившего его браузера. В дополнение к методам `getAudioClip()` и `getImage()` в интерфейсе `AppletContext` имеется несколько полезных методов, которые обсуждаются ниже.

Взаимодействие апплетов

Веб-страница может содержать несколько апплетов. Если значение атрибута `codebase` одинаково для нескольких апплетов, они могут взаимодействовать друг с другом. Естественно, что взаимодействие апплетов — довольно сложный процесс, необходимость в котором возникает нечасто.

Если задать в HTML-файле атрибут `name` для каждого апплета, то с помощью метода `getApplet(String)`, объявленного в интерфейсе `AppletContext`, можно получить ссылку на любой апплет. Допустим, в HTML-файле содержится следующий дескриптор:

```
<applet code="Chart.class" width="100" height="100" name="Chart1">
```

Тогда в результате приведенного ниже вызова будет получена ссылка на апплет.

```
Applet chart1 = getAppletContext().getApplet("Chart1");
```

Что можно сделать с этой ссылкой? Если в классе `Chart` имеется метод для получения новых данных и обновления гистограммы, его можно вызвать, выполнив соответствующее приведение типов следующим образом:

```
((Chart) chart1).setData(3, "Earth", 9000);
```

Кроме того, можно перечислить все апплеты, расположенные на веб-странице, независимо от наличия у них атрибута `name`. Метод `getApplets()` возвращает *объект перечисления* (подробнее объекты перечислений будут рассматриваться в главе 13). Этот объект, в частности, предоставляет доступ ко всем апплетам. Ниже приведен фрагмент кода, в котором выводятся имена апплетов, расположенных на текущей веб-странице. Апплет на данной веб-странице не может взаимодействовать с апплетами, находящимися на другой веб-странице.

```
Enumeration<Applet> e = getAppletContext().getApplets();
while (e.hasMoreElements())
{
    Applet a = e.nextElement();
    System.out.println(a.getClass().getName());
}
```

Отображение элементов в браузере

У разработчика апплетов имеется доступ к двум областям браузера, загружающего апплет: строке состояния и области отображения веб-страницы. В обоих случаях используются методы из интерфейса `AppletContext`. В частности, с помощью метода `showStatus()` можно вывести строку сообщения в строке состояния следующим образом:

```
showStatus("Loading data . . . please wait");
```



ВНИМАНИЕ! Как показывает опыт, метод `showStatus()` имеет ограниченную область применения. Ведь браузер также использует строку состояния, как правило, заменяя выводимую из апплета символьную строку своими сообщениями вроде `"Applet running"` (Выполняется апплет). Поэтому не пользуйтесь строкой состояния для вывода важных сообщений, которые пользователь не должен пропустить.

С помощью метода `showDocument()` можно отобразить другую веб-страницу. Для этого проще всего вызвать метод `showDocument()` с одним параметром, задающим URL нужного документа, как показано в приведенном ниже примере кода.

```
URL u = new URL("http://horstmann.com/index.html");
getAppletContext().showDocument(u);
```

Но дело в том, что браузер открывает новую веб-страницу в том же самом окне, где отображался текущий документ, заменяя таким образом апплет. Чтобы снова получить возможность работать с апплетом, пользователь должен щелкнуть на кнопке `Back` (Назад), доступной на панели инструментов браузера.

Вызывая метод `showDocument()` с двумя параметрами, можно указать браузеру, что документ должен быть выведен в другом окне. Значения второго параметра описаны в табл. 10.2. Второй параметр представлен символьной строкой. Если задать специальную строку `"_blank"`, браузер откроет новое окно, не изменяя текущее. Но важнее другое: окно браузера можно сначала разделить на несколько фреймов

и присвоить каждому из них имя, а затем разместить апплет во фрейме, отображая остальные документы в других фреймах. В следующем разделе будет показано, как это сделать.

Таблица 10.2. Значения второго параметра метода `showDocument()`

Значение параметра	Место вывода документа
"_self" или отсутствует	Текущий фрейм
"_parent"	Родительский фрейм
"_top"	Фрейм самого верхнего уровня
"_blank"	Новое безымянное окно верхнего уровня
Любая другая символьная строка	Фрейм с заданным именем. Если фрейма с таким именем не существует, открывается новое окно, получающее заданное имя



НА ЗАМЕТКУ! Утилита `appletviewer` не отображает веб-страницы, поэтому игнорирует метод `showDocument()`.

java.applet.Applet 1.2

- `public AppletContext getAppletContext()`
Предоставляет описатель среды браузера для апплета. В большинстве браузеров получаемые в итоге сведения могут быть использованы для управления тем браузером, в котором выполняется апплет.
- `void showStatus(String msg)`
Выводит указанную строку сообщения в строке состояния браузера.

java.applet.AppletContext 1.0

- `Enumeration<Applet> getApplets()`
Возвращает перечисление (см. главу 13) всех апплетов в одном и том же контексте, т.е. на одной веб-странице.
- `Applet getApplet(String name)`
Возвращает апплет из текущего контекста по его имени. Если апплет с таким именем не найден, возвращает пустое значение `null`. Просматривается только текущая веб-страница.
- `void showDocument(URL url)`
- `void showDocument(URL url, String target)`
Отображает новую веб-страницу в том же самом фрейме браузера. В первой форме данного метода новая страница заменяет текущую. А во второй форме для обозначения целевого фрейма служит параметр `target` (см. табл. 10.2).

Сохранение глобальных параметров настройки приложений

Пользователи приложений обычно ожидают, что предпочитаемые ими настройки глобальных параметров будут сохранены, а при последующем запуске приложения восстановлены. Сначала в этом разделе будет рассмотрен простой подход к сохранению конфигурационной информации в файлах свойств, который традиционно применяется в приложениях на Java, а затем прикладной интерфейс API для сохранения глобальных параметров настройки, предоставляющий более надежное решение.

Таблица свойств

Таблица свойств представляет собой структуру данных, в которой хранятся пары “ключ–значение”. Такие таблицы нередко используются для хранения сведений о параметрах настройки приложений и отличаются тремя характеристиками.

- Ключи и значения представлены символьными строками.
- Всю структуру данных легко записать и прочитать из файла.
- Имеется вспомогательная таблица для значений, устанавливаемых по умолчанию.

Таблицы свойств реализуются в классе `Properties`. Они очень удобны для обозначения разных вариантов настройки прикладных программ, как показано в приведенном ниже примере кода.

```
Properties settings = new Properties();
settings.put("width", "200");
settings.put("title", "Hello, World!");
```

Для записи списка свойств в файл служит метод `store()`. В приведенном ниже примере кода таблица свойств просто выводится в файл `program.properties`. А в качестве второго параметра указываются комментарии, которые вводятся в файл.

```
FileOutputStream out = new FileOutputStream("program.properties");
settings.store(out, "Program Properties");
```

В результате выполнения данного фрагмента кода будут выведены следующие свойства с сохраненными параметрами настройки прикладной программы:

```
#Program Properties
#Mon Apr 30 07:22:52 2007
width=200
title=Hello, World!
```

Для загрузки этих свойств из файла служит приведенный ниже фрагмент кода.

```
FileInputStream in = new FileInputStream("program.properties");
settings.load(in);
```

Обычно свойства программы принято сохранять в подкаталоге начального каталога пользователя. Имя такого каталога зачастую начинается с точки — в системе UNIX это соглашение указывает на системный каталог, скрытый от пользователя. Именно такое условное обозначение применяется в программе, рассматриваемой здесь в качестве примера.

Чтобы найти начальный каталог пользователя, можно вызвать метод `System.getProperties()`, в котором, как оказывается, объект типа `Properties` также служит

для описания системной информации. У начального каталога имеется ключ "user.home". Для чтения одиночного ключа предусмотрен следующий служебный метод:

```
String userDir = System.getProperty("user.home");
```

Если пользователь данной программы собирается править файл с таблицей свойств вручную, то целесообразно предоставить устанавливаемые по умолчанию значения ее свойств. Для этой цели в классе `Properties` предусмотрены два механизма. Прежде всего для поиска значения в символьной строке можно указать значение по умолчанию, которое должно выбираться автоматически в отсутствие соответствующего ключа:

```
String title = settings.getProperty("title", "Default title");
```

Если в таблице свойств имеется свойство "title", то переменной `title` присваивается символьная строка с данным свойством. В противном случае переменной `title` присваивается задаваемая по умолчанию символьная строка "Default title".

Если же задавать значения по умолчанию при каждом вызове метода `getProperty()` слишком утомительно, то все значения, устанавливаемые по умолчанию, можно ввести во вспомогательную таблицу свойств и передать ее конструктору основной таблицы свойств, как показано ниже.

```
Properties defaultSettings = new Properties();
defaultSettings.put("width", "300");
defaultSettings.put("height", "200");
defaultSettings.put("title", "Default title");
. . .
Properties settings = new Properties(defaultSettings);
```

Имеется даже возможность указать стандартные значения для настроек по умолчанию, если передать в качестве параметра другую таблицу свойств конструктору `defaultSettings`, но обычно так не делается.

В примере программы из листинга 10.5 показано, как пользоваться свойствами для сохранения и загрузки состояния программы. Эта программа запоминает положение, размеры и заголовок фрейма. У ее пользователя имеется также возможность отредактировать вручную файл `.corejava/program.properties` в своем начальном каталоге, чтобы придать этой программе требуемый внешний вид.



НА ЗАМЕТКУ! Свойства хранятся в простой таблице, не имеющей иерархической структуры. Как правило, используется фиктивная иерархия с именами ключей `window.main.color`, `window.main.title` и т.п. Но в классе `Properties` отсутствуют методы для построения настоящей иерархической структуры. Если в файле хранится сложная информация о конфигурации программы, в таком случае следует использовать класс `Preferences`, который будет описан в следующем разделе.

Листинг 10.5. Исходный код из файла `properties/PropertiesTest.java`

```
1 package properties;
2
3 import java.awt.EventQueue;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.util.Properties;
7
8 import javax.swing.*;
```

```
9
10 /**
11  * В этой программе проверяются свойства.
12  * В ней запоминаются положение, размеры и заголовок фрейма
13  * @version 1.00 2007-04-29
14  * @author Cay Horstmann
15  */
16 public class PropertiesTest
17 {
18     public static void main(String[] args)
19     {
20         EventQueue.invokeLater(new Runnable()
21         {
22             public void run()
23             {
24                 PropertiesFrame frame = new PropertiesFrame();
25                 frame.setVisible(true);
26             }
27         });
28     }
29 }
30
31 /**
32  * Фрейм, восстанавливающий свое положение и размеры и файла свойств
33  * и обновляющий свойства после выхода из программы
34  */
35 class PropertiesFrame extends JFrame
36 {
37     private static final int DEFAULT_WIDTH = 300;
38     private static final int DEFAULT_HEIGHT = 200;
39
40     private File propertiesFile;
41     private Properties settings;
42
43     public PropertiesFrame()
44     {
45         // получить положение, размеры и заголовок фрейма из свойств
46         String userDir = System.getProperty("user.home");
47         File propertiesDir = new File(userDir, ".corejava");
48         if (!propertiesDir.exists()) propertiesDir.mkdir();
49         propertiesFile = new File(propertiesDir, "program.properties");
50
51         Properties defaultSettings = new Properties();
52         defaultSettings.put("left", "0");
53         defaultSettings.put("top", "0");
54         defaultSettings.put("width", "" + DEFAULT_WIDTH);
55         defaultSettings.put("height", "" + DEFAULT_HEIGHT);
56         defaultSettings.put("title", "");
57
58         settings = new Properties(defaultSettings);
59
60         if (propertiesFile.exists()) try
61         {
62             FileInputStream in = new FileInputStream(propertiesFile);
63             settings.load(in);
64         }
65         catch (IOException ex)
66         {
67             ex.printStackTrace();
68         }
69     }

```

```
70 int left = Integer.parseInt(settings.getProperty("left"));
71 int top = Integer.parseInt(settings.getProperty("top"));
72 int width = Integer.parseInt(settings.getProperty("width"));
73 int height = Integer.parseInt(settings.getProperty("height"));
74 setBounds(left, top, width, height);
75
76 // если заголовок не задан, запросить его у пользователя
77
78 String title = settings.getProperty("title");
79 if (title.equals("")) title =
80     JOptionPane.showInputDialog("Please supply a frame title:");
81 if (title == null) title = "";
82 setTitle(title);
83
84 addWindowListener(new WindowAdapter()
85 {
86     public void windowClosing(WindowEvent event)
87     {
88         settings.put("left", "" + getX());
89         settings.put("top", "" + getY());
90         settings.put("width", "" + getWidth());
91         settings.put("height", "" + getHeight());
92         settings.put("title", getTitle());
93         try
94         {
95             FileOutputStream out = new FileOutputStream(propertiesFile);
96             settings.store(out, "Program Properties");
97         }
98         catch (IOException ex)
99         {
100             ex.printStackTrace();
101         }
102         System.exit(0);
103     }
104 });
105 }
```

java.util.Properties 1.0

- `Properties()`
Создает пустую таблицу свойств.
- `Properties(Properties defaults)`
Создает пустую таблицу свойств с заданными по умолчанию значениями.
Параметры: `defaults` Значения по умолчанию, используемые для поиска
- `String getProperty(String key)`
Возвращает символьную строку, связанную с заданным ключом, а если ключ в основной таблице свойств отсутствует, то строку, связанную с ключом из таблицы со значениями, устанавливаемыми по умолчанию. Если же указанный ключ отсутствует в таблице со значениями, устанавливаемыми по умолчанию, то возвращается пустое значение `null`.
Параметры: `key` Ключ, связанный с возвращаемой символьной строкой

- `String getProperty(String key, String defaultValue)` ·

Возвращает символьную строку, связанную с указанным ключом. Если же ключ отсутствует в таблице свойств, то возвращает свойство со значением, устанавливаемым по умолчанию.

Параметры: `key` Ключ, связанный с возвращаемой символьной строкой
`defaultValue` Символьная строка, которая возвращается, если ключ отсутствует в таблице свойств

- `void load(InputStream in) throws IOException`

Загружает таблицу свойств из потока ввода.

Параметры: `key` Поток ввода

- `void store(OutputStream out, String header) 1.2`

Выводит таблицу свойств в поток вывода.

Параметры: `out` Поток вывода
`header` Заголовок, размещаемый в первой строке сохраняемого файла свойств

java.lang.System 1.0

- `Properties getProperties()`

Извлекает все системные свойства. Приложение должно иметь право доступа ко всем свойствам, в противном случае генерируется исключение безопасности.

- `String getProperty(String key)`

Извлекает системное свойство по заданному ключу. Если у приложения отсутствует право доступа к данному свойству, то генерируется исключение безопасности. Следующие свойства всегда доступны для извлечения:

```
java.version
java.vendor
java.vendor.url
java.class.version
os.name
os.version
os.arch
file.separator
path.separator
line.separator
java.specification.version
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
```



НА ЗАМЕТКУ! Все имена свободно доступных системных свойств можно найти в файле `security/java.policy`, доступном из каталога исполняющей системы Java.

Прикладной интерфейс API для сохранения глобальных параметров настройки

Как было показано выше, класс `Properties` позволяет легко загружать и сохранять информацию о конфигурации прикладной программы. Но файлам свойств присущи следующие недостатки.

- В некоторых операционных системах вообще не поддерживаются начальные каталоги, что затрудняет выбор единого места для хранения конфигурационных файлов.
- Стандартные условные обозначения имен конфигурационных файлов отсутствуют, что может привести к конфликтам имен, если пользователь установит несколько приложений.

В некоторых операционных системах конфигурационные данные хранятся в центральном хранилище. Наиболее известным примером такого хранилища является системный реестр Windows. Подобное центральное хранилище поддерживается в классе `Preferences` независимо от операционной системы. Так, в Windows таким хранилищем конфигурационных данных для класса `Preferences` служит системный реестр, а в Linux — локальный системный файл. Разумеется, реализация центрального хранилища абсолютно прозрачна для разработчиков, пользующихся классом `Preferences`.

Хранилище, организуемое в классе `Preferences`, имеет древовидную структуру, в узлах которой содержатся имена путей, например `/com/mycompany/myapp`. Как и при составлении пакетов, во избежание конфликта имен рекомендуется формировать имена на основе доменных имен, записывая их в обратном порядке. При разработке прикладного интерфейса API для сохранения глобальных параметров настройки предполагалось, что пути к узлам должны совпадать с именами пакетов, используемых в прикладной программе.

Каждый узел в хранилище содержит отдельную таблицу пар “ключ–значение”, которую можно использовать для записи чисел, символьных строк или байтовых массивов. Возможность хранить сериализованные объекты не предусмотрена. Разработчики интерфейса API посчитали, что формат сериализации слишком хрупок и обеспечить долговременное хранение данных в нем совсем не просто. Разумеется, если вы не согласны с этим, то можете хранить сериализованные объекты в виде байтовых массивов.

Для дополнительного удобства предусмотрено несколько параллельных деревьев. В каждой программе используется одно дерево. Кроме того, существует дополнительное дерево, называемое системным и предназначенное для хранения настроек, общих для всех пользователей. Для доступа к соответствующему пользовательскому дереву в классе `Preferences` используется понятие “текущего пользователя”.

Поиск требуемого узла в дереве начинается с пользовательского или системного корня:

```
Preferences root = Preferences.userRoot();
```

или

```
Preferences root = Preferences.systemRoot();
```

И тогда для доступа к узлу достаточно указать соответствующий путь:

```
Preferences node = root.node("/com/mycompany/myapp");
```

Для быстрого и удобного доступа к узлу дерева путь к нему приравнивается к имени пакета его класса. Для этого достаточно взять объект данного класса и сделать вызов одним из двух приведенных ниже способов. Как правило, ссылка `obj` равна `this`:

```
Preferences node = Preferences.userNodeForPackage(obj.getClass());
```

или

```
Preferences node = Preferences.systemNodeForPackage(obj.getClass());
```

Получив доступ к узлу, можно обратиться к таблице с парами “ключ–значение”, используя такие методы:

```
String get(String key, String defval)
int getInt(String key, int defval)
long getLong(String key, long defval)
float getFloat(String key, float defval)
double getDouble(String key, double defval)
boolean getBoolean(String key, boolean defval)
byte[] getByteArray(String key, byte[] defval)
```

Но если хранилище недоступно, то при чтении конфигурационных данных придется указывать значения, устанавливаемые по умолчанию. Эти значения требуются по нескольким причинам. Во-первых, данные могут быть неполными, так как пользователи не всегда указывают все параметры. Во-вторых, на некоторых платформах хранилища могут не создаваться из-за нехватки ресурсов. И в-третьих, мобильные устройства могут быть временно отсоединены от хранилища.

С другой стороны, данные в хранилище можно записать, используя методы

```
put(String key, String value)
putInt(String key, int value)
```

и так далее в зависимости от типа данных. А перечислить все ключи, сохраненные в узле, можно с помощью метода `String[] keys()`.

В настоящее время нет способа, позволяющего определить тип значения, связанного с конкретным ключом. Центральным хранилищам, например системному реестру в Windows, традиционно присущи два недостатка.

- Во-первых, в них постепенно накапливается устаревшая информация.
- Во-вторых, данные о конфигурации системы становятся все более запутанными, что затрудняет перенос глобальных параметров настройки на новую платформу.

Класс `Preferences` позволяет устранить второй недостаток. Данные можно экспортировать в поддерево, а в некоторых случаях — в отдельный узел, вызвав один из приведенных ниже методов.

```
void exportSubtree(OutputStream out)
void exportNode(OutputStream out)
```

Данные хранятся в формате XML. Их можно импортировать в другое хранилище, вызвав следующий метод:

```
void importPreferences(InputStream in)
```

В качестве примера ниже приведены конфигурационные данные, размеченные в формате XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM "http://java.sun.com/dtd/preferences.dtd">
<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map/>
    <node name="com">
      <map/>
      <node name="horstmann">
        <map/>
        <node name="corejava">
          <map>
            <entry key="left" value="11"/>
            <entry key="top" value="9"/>
            <entry key="width" value="453"/>
            <entry key="height" value="365"/>
            <entry key="title" value="Hello, World!"/>
          </map>
        </node>
      </node>
    </node>
  </root>
</preferences>
```

Если в прикладной программе используются глобальные параметры настройки, ее пользователям можно предоставить возможность экспортировать и импортировать эти глобальные параметры, чтобы упростить перенос программы с одного компьютера на другой. Такой подход демонстрируется в примере программы, исходный код которой приведен в листинге 10.6. В этой программе сохраняются координаты и размеры главного окна. Попробуйте изменить размеры окна, выйти из программы и запустить ее снова на выполнение. Размеры окна должны совпадать с теми размерами, которые были заданы перед выходом из программы.

Листинг 10.6. Исходный код из файла preferences/PreferencesTest.java

```
1 package preferences;
2
3 import java.awt.EventQueue;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.util.prefs.*;
7 import javax.swing.*;
8
9 /**
10  * В этой программе проверяются глобальные параметры настройки.
11  * В ней запоминаются положение, размеры и заголовок фрейма
12  * @version 1.02 2007-06-12
13  * @author Cay Horstmann
14  */
15 public class PreferencesTest
16 {
17     public static void main(String[] args)
18     {
19         EventQueue.invokeLater(new Runnable()
20         {
```

```
21     public void run()
22     {
23         PreferencesFrame frame = new PreferencesFrame();
24         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25         frame.setVisible(true);
26     }
27     });
28 }
29 }
30
31 /**
32  * Фрейм, восстанавливающий свое положение и размеры и файла свойств
33  * и обновляющий свойства после выхода из программы
34  */
35 class PreferencesFrame extends JFrame
36 {
37     private static final int DEFAULT_WIDTH = 300;
38     private static final int DEFAULT_HEIGHT = 200;
39
40     public PreferencesFrame()
41     {
42         // получить положение, размеры и заголовок фрейма из свойств
43
44         Preferences root = Preferences.userRoot();
45         final Preferences node = root.node("/com/horstmann/corejava");
46         int left = node.getInt("left", 0);
47         int top = node.getInt("top", 0);
48         int width = node.getInt("width", DEFAULT_WIDTH);
49         int height = node.getInt("height", DEFAULT_HEIGHT);
50         setBounds(left, top, width, height);
51
52         // если заголовок не задан, запросить его у пользователя
53
54         String title = node.get("title", "");
55         if (title.equals("")) title =
56             JOptionPane.showInputDialog("Please supply a frame title:");
57         if (title == null) title = "";
58         setTitle(title);
59
60         // установить компонент для выбора и отображения XML-файлов
61         final JFileChooser chooser = new JFileChooser();
62         chooser.setCurrentDirectory(new File("."));
63
64         // принять файлы с расширением .xml
65         chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
66         {
67             public boolean accept(File f)
68             {
69                 return f.getName().toLowerCase().endsWith(".xml") || f.isDirectory();
70             }
71
72             public String getDescription()
73             {
74                 return "XML files";
75             }
76         });
77     }
```

```
78 // установить меню
79 JMenuBar menuBar = new JMenuBar();
80 setJMenuBar(menuBar);
81 JMenu menu = new JMenu("File");
82 menuBar.add(menu);
83
84 JMenuItem exportItem = new JMenuItem("Export preferences");
85 menu.add(exportItem);
86 exportItem.addActionListener(new ActionListener()
87 {
88     public void actionPerformed(ActionEvent event)
89     {
90         if (chooser.showSaveDialog(PreferencesFrame.this) ==
91             JFileChooser.APPROVE_OPTION)
92         {
93             try
94             {
95                 OutputStream out =
96                     new FileOutputStream(chooser.getSelectedFile());
97                 node.exportSubtree(out);
98                 out.close();
99             }
100            catch (Exception e)
101            {
102                e.printStackTrace();
103            }
104        }
105    }
106 });
107
108 JMenuItem importItem = new JMenuItem("Import preferences");
109 menu.add(importItem);
110 importItem.addActionListener(new ActionListener()
111 {
112     public void actionPerformed(ActionEvent event)
113     {
114         if (chooser.showOpenDialog(PreferencesFrame.this) ==
115             JFileChooser.APPROVE_OPTION)
116         {
117             try
118             {
119                 InputStream in =
120                     new FileInputStream(chooser.getSelectedFile());
121                 Preferences.importPreferences(in);
122                 in.close();
123             }
124            catch (Exception e)
125            {
126                e.printStackTrace();
127            }
128        }
129    }
130 });
131
132 JMenuItem exitItem = new JMenuItem("Exit");
133 menu.add(exitItem);
134 exitItem.addActionListener(new ActionListener()
```

```
135     {
136     public void actionPerformed(ActionEvent event)
137     {
138         node.putInt("left", getX());
139         node.putInt("top", getY());
140         node.putInt("width", getWidth());
141         node.putInt("height", getHeight());
142         node.put("title", getTitle());
143         System.exit(0);
144     }
145     });
146 }
147 }
```

java.util.prefs.Preferences 1.4

- Preferences userRoot()
Возвращает корневой узел из дерева глобальных параметров настройки для пользователя вызывающей программы.
- Preferences systemRoot()
Возвращает системный корневой узел из дерева глобальных параметров настройки.
- Preferences node(String path)
Возвращает узел, достигаемый из текущего узла по заданному пути. Если в качестве параметра *path* указан абсолютный путь, который обычно начинается с косой черты (/), то узел достигается из корня дерева глобальных параметров настройки. Если узел отсутствует по заданному пути, он создается.
- Preferences userNodeForPackage(Class cl)
- Preferences systemNodeForPackage(Class cl)
Возвращают узел из дерева текущего пользователя или системного дерева, абсолютный путь к которому соответствует имени пакета, содержащего класс *cl*.
- String[] keys()
Возвращает все ключи, принадлежащие данному узлу.
- String get(String key, String defval)
- int getInt(String key, int defval)
- long getLong(String key, long defval)
- float getFloat(String key, float defval)
- double getDouble(String key, double defval)
- boolean getBoolean(String key, boolean defval)
- byte[] getByteArray(String key, byte[] defval)
Возвращают значение, связанное с заданным ключом. Если значение отсутствует в хранилище глобальных параметров настройки, имеет неверный тип или же само хранилище недоступно, возвращается значение, предусмотренное по умолчанию.
- void put(String key, String value)

- `void putInt(String key, int value)`
- `void putLong(String key, long value)`
- `void putFloat(String key, float value)`
- `void putDouble(String key, double value)`
- `void putBoolean(String key, boolean value)`
- `void putByteArray(String key, byte[] value)`
Сохраняют пару "ключ-значение" в заданном узле дерева.
- `void exportSubtree(OutputStream out)`
- Выводит в указанный поток глобальные параметры настройки, хранящиеся в заданном узле и производных от него узлах.
- `void exportNode(OutputStream out)`
Выводит в указанный поток глобальные параметры настройки, хранящиеся в заданном узле, игнорируя производные от него узлы.
- `void importPreferences(InputStream in)`
Импортирует параметры глобальные настройки из указанного потока.

На этом обсуждение способов развертывания и доставки прикладных программ, написанных на Java завершается. В следующей главе речь пойдет об обработке исключений, возникающих во время работы программы. Попутно будет дан ряд рекомендаций, касающихся способов тестирования и отладки программ, чтобы избежать многих осложнений при их выполнении.

Исключения, утверждения, протоколирование и отладка

В этой главе...

- ▶ Обработка ошибок
- ▶ Перехват исключений
- ▶ Рекомендации по обработке исключений
- ▶ Применение утверждений
- ▶ Протоколирование
- ▶ Рекомендации по отладке программ с ГПИ
- ▶ Применение отладчика

В идеальном случае пользователи никогда не делают ошибок при вводе данных, файлы, которые они пытаются открыть, всегда существуют, а программы всегда работают правильно. Исходный код в приведенных до сих пор примерах был рассчитан именно на такой идеальный случай. Но теперь пришла пора опуститься на землю, где людям свойственно делать ошибки при программировании и вводе данных, и рассмотреть механизмы Java, позволяющие выявлять и обрабатывать неизбежные ошибки в программах и данных.

Иметь дело с ошибками всегда неприятно. Если пользователь однажды потеряет все результаты своей работы из-за скрытой ошибки в программе или непредвиденного стечения обстоятельств, он может навсегда отвернуться от такой программы. В подобной ситуации необходимо сделать, по крайней мере, следующее.

- Сообщить пользователю об обнаруженной программной ошибке.
- Сохранить все результаты его работы.
- Дать ему возможность благополучно завершить программу.

В исключительных ситуациях, например, при вводе данных с ошибками, которые могут привести к нарушению нормальной работы программы, в Java применяется механизм перехвата ошибок, который называется *обработкой исключений*. Этот механизм аналогичен обработке исключений в C++ и Delphi и описывается в первой части этой главы.

Во время тестирования программы реализуется целый ряд проверок, чтобы убедиться в правильности ее работы. Но эти проверки зачастую оказываются слишком дорогостоящими по времени и ресурсам, а по окончании тестирования — излишними. Проверки можно удалить, а когда потребуется дополнительное тестирование, вернуть их обратно, хотя это и потребует определенных затрат труда. Вторая часть главы посвящена применению средств диагностических утверждений для выборочной активизации проверок.

Когда прикладная программа делает что-то не так, не всегда имеется возможность оказать пользователю своевременную помощь или прервать работу программы. Вместо этого можно зарегистрировать возникшую неполадку для последующего анализа. Поэтому третья часть главы посвящена средствам протоколирования.

И в завершение этой главы будут даны некоторые рекомендации по поводу того, как получить полезные сведения из работающей прикладной программы на Java и как пользоваться отладчиком, входящим в состав ИСР.

Обработка ошибок

Допустим, в ходе выполнения программы, написанной на Java, обнаруживается ошибка. Она может быть вызвана неверной информацией в файле, ненадежным сетевым соединением или выходом за допустимые границы массива, как бы неприятно ни было об этом упоминать, а может быть и попыткой использовать ссылку, которая не указывает на какой-то конкретный объект. Вполне естественно, пользователи надеются, что программа самостоятельно справится с возникшими затруднениями. Если из-за ошибки какая-нибудь операция не может быть завершена благополучно, программа должна сделать одно из двух.

- Вернуться в безопасное состояние и разрешить пользователю выполнить другие команды.
- Дать пользователю возможность сохранить результаты своей работы и аккуратно завершить работу.

Добиться этого не так-то просто: фрагмент кода, в котором обнаруживается ошибка (или даже тот фрагмент кода, выполнение которого приводит к ошибке), обычно находится очень далеко от кода, который может восстановить данные и сохранить

результаты, полученные пользователем. Поэтому основное назначение механизма обработки исключений — передать данные обработчику исключений из того места, где возник сбой. Предусматривая обработку исключений в программе, следует предвидеть возможные ошибки и связанные с ними осложнения. Какие же ошибки следует рассмотреть в первую очередь?

- *Ошибки ввода.* В дополнение к неизбежным опечаткам пользователи часто предпочитают действовать так, как они считают нужным, а не следовать инструкциям разработчиков прикладных программ. Допустим, пользователю требуется перейти на веб-сайт, но он допустил синтаксическую ошибку при вводе веб-адреса (URL) этого сайта. Программа должна была бы проверить синтаксис URL, но, предположим, что ее автор забыл это сделать. Тогда сетевое программное обеспечение сообщит об ошибке.
- *Сбои оборудования.* Оборудование не всегда работает должным образом. Принтер может оказаться выключенным, а веб-страница — временно недоступной. Зачастую оборудование перестает нормально работать в ходе выполнения задания, например, бумага в принтере может закончиться на середине печатаемой страницы.
- *Физические ограничения.* Диск может оказаться переполненным, а оперативная память — исчерпанной.
- *Ошибки программирования.* Какой-нибудь метод может работать неправильно. Например, он может возвращать неверный результат или некорректно вызывать другие методы. Выход за допустимые границы массива, попытка найти несуществующий элемент в хеш-таблице, извлечение элемента из пустого стека — все это характерные примеры ошибок программирования.

Обычно метод сообщает об ошибке, возвращая специальный код, который анализируется вызывающим методом. Например, методы, вводящие данные из файлов, обычно возвращают значение `-1` по достижении конца файла. Такой способ обработки ошибок часто оказывается эффективным. В других случаях в качестве признака ошибки возвращается пустое значение `null`.

К сожалению, возможность возвращать код ошибки имеется далеко не всегда. Иногда правильные данные не удается отличить от признаков ошибок. Так, метод, возвращающий целое значение, вряд ли возвратит значение `-1`, обнаружив ошибку, поскольку оно может быть получено в результате вычислений.

Как упоминалось в главе 5, в Java имеется возможность предусмотреть в каждом методе альтернативный выход, которым следует воспользоваться, если нормальное выполнение задания нельзя довести до конца. В этом случае метод не станет возвращать значение, а *сгенерирует* объект, инкапсулирующий сведения о возникшей ошибке. Следует, однако, иметь в виду, что выход из метода происходит незамедлительно, и он не возвращает своего нормального значения. Более того, возобновить выполнение кода, вызвавшего данный метод, невозможно. Вместо этого начинается поиск обработчика исключений, который может справиться с возникшей ошибочной ситуацией.

Исключения имеют свой собственный синтаксис и являются частью особой иерархии наследования. Сначала мы рассмотрим особенности синтаксиса исключений, а затем дадим ряд рекомендаций относительно эффективного применения обработчиков исключений.

Классификация исключений

В Java объект исключения всегда является экземпляром класса, производного от класса `Throwable`. Как станет ясно в дальнейшем, если стандартных классов недостаточно, можно создавать и свои собственные классы исключений. На рис. 11.1 показана в упрощенном виде иерархия наследования исключений в Java.

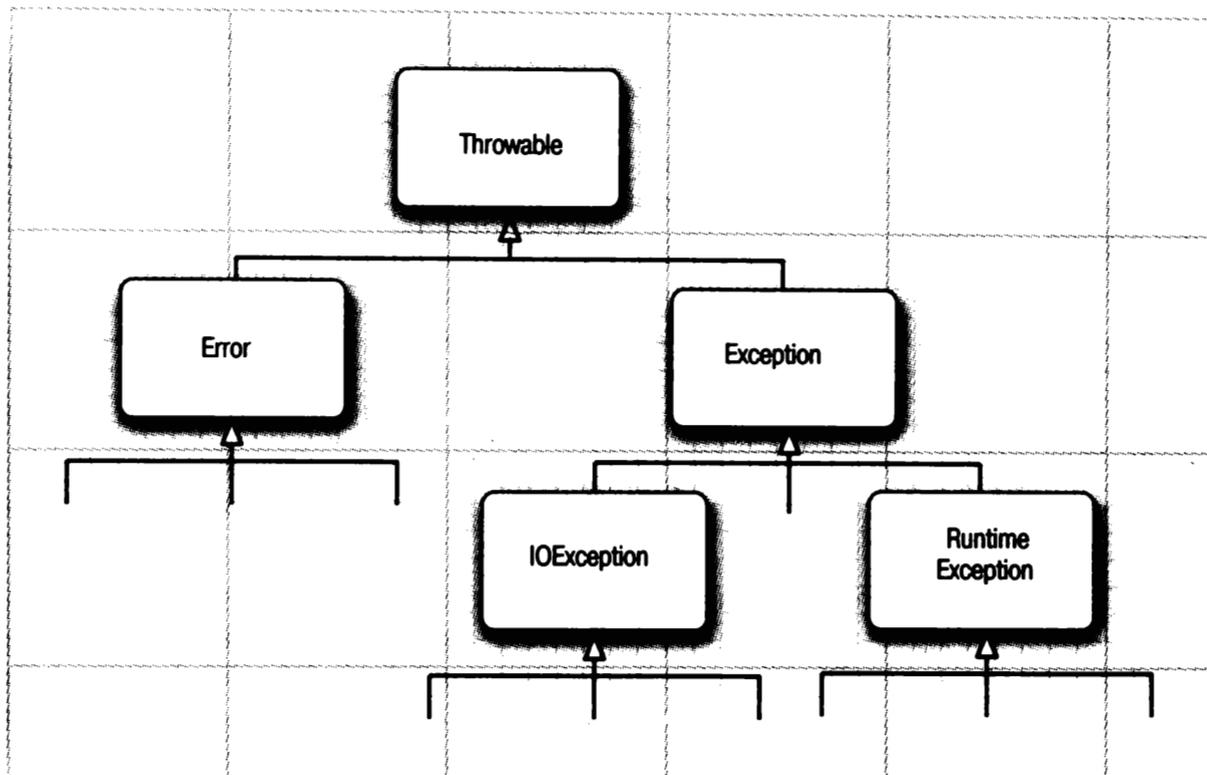


Рис. 11.1. Иерархия наследования исключений в Java

Обратите внимание на то, что иерархия наследования исключений сразу же разделяется на две ветви: `Error` и `Exception`, хотя общим предшественником для всех исключений является класс `Throwable`. Иерархия класса `Error` описывает внутренние ошибки и ситуации, возникающие в связи с нехваткой ресурсов в исполняющей системе Java. Ни один объект этого класса нельзя сгенерировать самостоятельно. При возникновении внутренней ошибки в такой системе возможности разработчика прикладной программы крайне ограничены. Можно лишь уведомить пользователя и попытаться аккуратно прервать выполнение программы, хотя такие ситуации достаточно редки.

При программировании на Java основное внимание следует уделять иерархии класса `Exception`. Эта иерархия также разделяется на две ветви: исключения, производные от класса `RuntimeException`, и остальные. Исключения типа `RuntimeException` возникают вследствие ошибок программирования. Все другие виды исключений являются следствием непредвиденного стечения обстоятельств, например, ошибок ввода-вывода, возникающих при выполнении вполне корректных программ.

Исключения, производные от класса `RuntimeException`, связаны со следующими программными ошибками.

- Неверное приведение типов.
- Выход за пределы массива.
- Попытка обратиться к объекту по пустой ссылке `null`.

Остальные исключения возникают в следующих случаях.

- Попытка чтения по достижении конца файла.
- Попытка открыть несуществующий файл.
- Попытка получить объект типа `Class`, если в символьной строке указан несуществующий класс.

Исключения типа `RuntimeException` практически всегда возникают по вине программиста. Так, исключения типа `ArrayIndexOutOfBoundsException` можно избежать, если всегда проверять индексы массива. А исключение `NullPointerException` никогда не возникнет, если перед тем, как воспользоваться переменной, проверить, не содержит ли она пустое значение `null`.

А как быть, если файл не существует? Нельзя ли сначала проверить, существует ли он вообще? Дело в том, что файл может быть удален сразу же после проверки его существования. Следовательно, понятие существования файла зависит от среды исполнения, а не от кода программы.

В спецификации языка Java любое исключение, производное от класса `Error` или `RuntimeException`, называется *непроверяемым*. Все остальные исключения называются *проверяемыми*. Для всех проверяемых исключений компилятор проверяет наличие соответствующих обработчиков.



НА ЗАМЕТКУ! Имя класса `RuntimeException` иногда вводит в заблуждение. Разумеется, все рассматриваемые здесь ошибки возникают во время выполнения программы.



НА ЗАМЕТКУ C++! Если вы знакомы с иерархией исключений (намного более ограниченной) из стандартной библиотеки C++, то, возможно, заметили некоторое противоречие в именах. В C++ имеются два основных класса, описывающих исключения: `runtime_error` и `logic_error`. Класс `logic_error` равнозначен классу `RuntimeException` в Java. А класс `runtime_error` является базовым для исключений, возникающих при непредвиденном стечении обстоятельств. Он равнозначен всем исключениям, предусмотренным в Java, но не относящимся к типу `RuntimeException`.

Объявление проверяемых исключений

Метод может генерировать исключения, если возникает ситуация, с которой он не в состоянии справиться. Идея проста: метод не только сообщает компилятору, какие значения он может возвращать, но и предсказывает, какие ошибки *могут* возникнуть. Например, в коде, вводящем данные из файла, можно предположить, что такого файла не существует или он пуст. Следовательно, код, предназначенный для ввода из файла, должен сообщить компилятору, что он может сгенерировать исключение типа `IOException`.

Объявление о том, что данный метод может генерировать исключение, делается в его заголовке. Ниже в качестве примера приведено объявление одного из конструкторов класса `FileInputStream` из стандартной библиотеки. (Потоки рассматриваются в главе 12.)

```
public FileInputStream(String name) throws FileNotFoundException
```

Это объявление означает, что данный конструктор создает объект типа `FileInputStream`, исходя из параметра `name` типа `String`, но в определенных случаях, когда что-нибудь пойдет не так, он может также сгенерировать исключение типа `FileNotFoundException`. Если это произойдет, исполняющая система начнет поиск обработчика событий, предназначенного для обработки объектов типа `FileNotFoundException`.

Создавая свой собственный метод, не нужно объявлять все возможные исключения, которые этот метод фактически может сгенерировать. Чтобы лучше понять, когда и что следует описывать с помощью оператора `throws`, имейте в виду, что исключение генерируется в следующих четырех случаях.

- Вызывается метод, генерирующий проверяемое исключение, например конструктор класса `FileInputStream`.
- Обнаружена ошибка, и с помощью оператора `throw` явным образом генерируется проверяемое исключение (оператор `throw` обсуждается в следующем разделе).
- Обнаружена ошибка программирования, например, в программе присутствует выражение `a[-1] = 0`, вследствие чего возникает непроверяемое исключение, например, типа `ArrayIndexOutOfBoundsException`.
- Возникает внутренняя ошибка в виртуальной машине или библиотеке исполняющей системы.

В первых двух случаях нужно сообщить тем, кто будет пользоваться данным методом, что возможно исключение. Зачем? А затем, что любой метод, генерирующий исключение, представляет собой потенциально опасное место в прикладной программе. Если своевременно не предусмотреть в ней обработку данного типа исключения, то выполнение текущего потока прервется.

Объявить, что метод может генерировать исключение, можно, включив в его заголовок *описание исключения*, как показано ниже.

```
class MyAnimation
{
    . . .
    public Image loadImage(String s) throws IOException
    {
        . . .
    }
}
```

Если же метод может генерировать несколько проверяемых исключений, все они должны быть перечислены в его заголовке через запятую следующим образом:

```
class MyAnimation
{
    . . .
    public Image loadImage(String s)
        throws FileNotFoundException, EOFException
    {
        . . .
    }
}
```

Но внутренние ошибки, т.е. исключения, производные от класса `Error`, объявлять не нужно. Такие исключения могут генерироваться любыми методами, а самое главное, что они не поддаются никакому контролю.

Точно так же совсем не обязательно объявлять непроверяемые исключения, производные от класса `RuntimeException`:

```
class MyAnimation
{
    . . .
    void drawImage(int i)
        throws ArrayIndexOutOfBoundsException // не рекомендуется!
    {
        . . .
    }
}
```

Ответственность за подобные ошибки полностью возлагается на разработчика прикладной программы. Так, если вас беспокоит возможность выхода индекса за допустимые границы массива, лучше уделите больше внимания предотвращению этой исключительной ситуации, вместо того чтобы объявлять о потенциальной опасности ее возникновения.

Таким образом, в методе должны быть объявлены все *проверяемые* исключения, которые он может генерировать. А непроверяемые исключения находятся вне контроля разработчика данного метода (класс `Error`) или же являются следствием логических ошибок, которые не следовало допускать (класс `RuntimeException`). Если же в объявлении метода не сообщается обо всех проверяемых исключениях, компилятор выдаст сообщение об ошибке.

Разумеется, вместо объявления исключений, как было показано выше, их можно перехватывать. В этом случае исключение не генерируется, и объявлять его в операторе `throws` не нужно. Далее в этой главе мы обсудим, что лучше: перехватывать исключение самостоятельно или поручить это кому-нибудь другому.



ВНИМАНИЕ! Метод подкласса не может генерировать более общие исключения, чем переопределяемый им метод суперкласса. (Исключения могут только конкретизироваться или не возникать совсем.) Если, например, метод суперкласса вообще не генерирует проверяемые исключения, то и подкласс этого сделать не может. Так, если вы переопределяете метод `JComponent.paintComponent()`, то ваш метод `paintComponent()` не должен генерировать ни одного проверяемого исключения, поскольку соответствующий метод суперкласса этого не делает.

Если в объявлении метода указывается, что он может генерировать исключение определенного класса, то он может также генерировать исключения его подклассов. Например, в конструкторе типа `FileInputStream` можно объявить о возможности генерирования исключения типа `IOException`, причем неизвестно, какого именно. Это может быть, в частности, простое исключение класса `IOException` или же объект одного из производных от него классов, в том числе класса `FileNotFoundException`.



НА ЗАМЕТКУ C++! Спецификатор `throws` в Java совпадает со спецификатором `throws` в C++. Но у них имеется одно существенное отличие. В C++ спецификатор `throws` действует только во время выполнения, а не на стадии компиляции. Это означает, что компилятор C++ игнорирует объявление исключений. Но если исключение генерируется в функции, не включенной в список спецификатора `throws`, то вызывается функция `unexpected()` и по умолчанию выполнение программы прекращается. Кроме того, если спецификатор `throws` в коде C++ не указан, то функция может генерировать любое исключение.

Порядок генерирования исключений

Допустим, в прикладной программе произошло нечто ужасное. Так, в ней имеется метод `readData()`, вводящий данные из файла, в заголовке которого указано следующее:

```
Content-length: 1024
```

Но после ввода **733** символов достигнут конец файла. Такую ситуацию можно посчитать настолько ненормальной, что требуется сгенерировать исключение. Далее нужно решить, какого типа должно быть исключение. В данном случае подходит одно из исключений типа `IOException`. В документации можно найти следующее описание исключения типа `EOFException`: “Сигнализирует о том, что во время ввода данных неожиданно обнаружен признак конца файла **EOF**”. Как раз то, что нужно! Такое исключение можно сгенерировать следующим способом:

```
throw new EOFException();
```

или, если угодно, таким:

```
EOFException e = new EOFException();  
throw e;
```

Ниже показано, как генерирование такого исключения реализуется непосредственно в коде.

```
String readData(Scanner in) throws EOFException  
{  
    . . .  
    while (. . .)  
    {  
        if (!in.hasNext()) // достигнут конец файла (признак EOF)  
        {  
            if (n < len)  
                throw new EOFException();  
        }  
        . . .  
    }  
    return s;  
}
```

В классе `EOFException` имеется второй конструктор, получающий в качестве параметра символьную строку. Его можно использовать для более подробного описания исключения, как показано ниже.

```
String gripe = "Content-length: " + len + ", Received: " + n;  
throw new EOFException(gripe);
```

Как видите, если один из существующих классов исключений подходит для конкретных целей прикладной программы, то сгенерировать в ней исключение не составит большого труда. В этом случае нужно сделать следующее.

1. Найти подходящий класс.
2. Создать экземпляр этого класса.
3. Сгенерировать исключение.

После того как исключение будет сгенерировано в методе, управление уже не возвратится в вызывающую часть программы. Это означает, что беспокоиться о возвращении значения, предусмотренного по умолчанию, или кода ошибки не нужно.



НА ЗАМЕТКУ C++! Генерирование исключений в C++ и Java происходит почти одинаково, за одним незначительным отличием. В Java можно генерировать только объекты классов, производных от класса `Throwable`, а в C++ — объекты любого класса.

Создание классов исключений

В прикладной программе может возникнуть ситуация, не предусмотренная ни в одном из стандартных классов исключений. В этом случае нетрудно создать свой собственный класс исключения. Очевидно, что он должен быть подклассом, производным от класса `Exception` или одного из его подклассов, например, `IOException`, как показано ниже. Этот класс можно снабдить конструктором по умолчанию и конструктором, содержащим подробное сообщение. (Это сообщение, возвращаемое методом `toString()` из суперкласса `Throwable`, может оказаться полезным при отладке программы.)

```
class FileFormatException extends IOException
{
    public FileFormatException() {}
    public FileFormatException(String gripe)
    {
        super(gripe);
    }
}
```

А теперь можно сгенерировать исключение собственного типа следующим образом:

```
String readData(BufferedReader in) throws FileFormatException
{
    . . .
    while (. . .)
    {
        if (ch == -1) // достигнут конец файла (признак EOF)
        {
            if (n < len)
                throw new FileFormatException();
        }
        . . .
    }
    return s;
}
```

java.lang.Throwable 1.0

- `Throwable()`
Создает новый объект типа `Throwable`, не сопровождая его подробным сообщением.
- `Throwable(String message)`
Создает новый объект типа `Throwable`, сопровождая его подробным сообщением. По соглашению все классы исключений должны содержать два конструктора: конструктор по умолчанию и конструктор с детальным сообщением.
- `String getMessage()`
Получает подробное сообщение, предусмотренное в объекте типа `Throwable`.

Перехват исключений

Теперь вы знаете, как генерировать исключения. Как видите, ничего сложного в этом нет: сгенерировав исключение, о нем можно попросту забыть. Разумеется, должен существовать код, позволяющий перехватить и обработать исключение.

Если исключение возникает и нигде не перехватывается, то программа прекращает работу, выводя на консоль сообщение о типе исключения и содержимое стека. Программы с ГПИ (апплеты и приложения) перехватывают исключения и выводят аналогичное сообщение, возвращаясь затем в цикл обработки событий в пользовательском интерфейсе. (Отлаживая программы с ГПИ, лучше держать консольное окно на экране, не сворачивая его.)

Перехват исключения осуществляется в блоке операторов `try/catch`. В простейшем случае этот блок имеет следующий вид:

```
try
{
    код
    дополнительный код
    дополнительный код
}
catch (ТипИсключения e)
{
    обработчик исключений данного типа
}
```

Если фрагмент кода в блоке `try` генерирует исключение типа, указанного в заголовке блока `catch`, то происходит следующее.

1. Программа пропускает оставшуюся часть кода в блоке `try`.
2. Программа выполняет код обработчика в блоке `catch`.

Если код в блоке `try` не генерирует исключение, то программа пропускает блок `catch`. А если какой-нибудь из операторов блока `try` сгенерирует исключение, отличающееся от типа, указанного в блоке `catch`, выполнение данной части программы (в частности, вызываемого метода) немедленно прекращается. (Остается только надеяться, что в вызывающей части программы перехват исключения данного типа все же предусмотрен.)

Чтобы продемонстрировать, как все описанное выше действует на практике, рассмотрим следующий типичный код для ввода данных:

```
public void read(String filename)
{
    try
    {
        InputStream in = new FileInputStream(filename);
        int b;
        while ((b = in.read()) != -1)
        {
            обработать введенные данные
        }
    }
    catch (IOException exception)
    {
        exception.printStackTrace();
    }
}
```

Обратите внимание на то, что большая часть кода в блоке `try` выполняет простейшие действия: вводит и обрабатывает данные побайтно до тех пор, пока не будет достигнут конец файла. Как указано в документации на прикладной интерфейс API, существует вероятность того, что метод `read()` сгенерирует исключение `IOException`. В таком случае пропускается весь остальной цикл `while`, происходит непосредственный переход к блоку `catch` и генерируется трассировка стека. Для небольших программ это вполне благоразумный способ обработки исключений. А какие еще существуют возможности?

Зачастую лучше вообще ничего не делать, а просто передать исключение вызывающей части программы. Если в методе `read()` возникнет ошибка ввода, ответственность за обработку этой исключительной ситуации следует возложить на вызывающую часть программы! Придерживаясь такого подхода, достаточно указать, что метод `read()` может генерировать исключение типа `IOException` следующим образом:

```
public void read(String filename) throws IOException
{
    InputStream in = new FileInputStream(filename);
    int b;
    while ((b = in.read()) != -1)
    {
        обработать введенные данные
    }
}
```

Напомним, что компилятор строго следит за спецификаторами `throws`. Вызывая метод, генерирующий проверяемое исключение, следует обработать его самостоятельно в этом методе или делегировать его обработку вызывающей части программы. Что же лучше? Как правило, следует перехватывать лишь те исключения, которые можно обработать самостоятельно, а остальные передавать дальше. Передавая исключение, следует непременно добавить спецификатор `throws`, чтобы предупредить об этом вызывающую часть программы.

Просматривая документацию на прикладной интерфейс API, обратите внимание на методы, которые способны генерировать исключения. А затем решите, следует ли обрабатывать исключение самостоятельно или включить его в список спецификатора

throws. В последнем варианте нет ничего постыдного: лучше предоставить обработку исключения более компетентному обработчику, чем пытаться организовать ее самостоятельно.

Как упоминалось ранее, из этого правила имеется одно исключение. Если создается метод, переопределяющий метод суперкласса, но не генерирующий ни одного исключения (например, метод `paintComponent()` из класса `JComponent`), каждое проверяемое исключение следует *непрерывно* перехватывать в теле этого метода самостоятельно. В этот метод из подкласса нельзя добавить спецификатор `throws`, отсутствующий в переопределяемом методе из суперкласса.



НА ЗАМЕТКУ C++! Перехват исключений в Java и C++ осуществляется почти одинаково. Следующий оператор в Java:

```
catch (Exception e) // Java
```

является аналогом приведенного ниже оператора в C++:

```
catch (Exception& e) // C++
```

В блоке `try` можно перехватить несколько исключений, обработав их по отдельности. Для каждого типа исключения следует предусмотреть свой блок `catch` следующим образом:

```
try
{
    код, способный генерировать исключения
}
catch (FileNotFoundException e)
{
    чрезвычайные действия, если отсутствуют нужные файлы
}
catch (UnknownHostException e)
{
    чрезвычайные действия, если хосты неизвестны
}
catch (IOException e)
{
    чрезвычайные действия во всех остальных
    случаях появления ошибок ввода-вывода
}
```

Объект исключения содержит сведения о нем. Для того чтобы получить дополнительные сведения об этом объекте из подробного сообщения об ошибке, достаточно сделать вызов `e.getMessage()`, для того чтобы получить конкретный тип объекта исключения, — вызов `e.getClass().getName()`.

Начиная с версии Java SE7, разнотипные исключения можно перехватывать в одном и том же блоке `catch`. Допустим, чрезвычайные действия, если нужные файлы отсутствуют или хосты неизвестны, одинаковы. В таком случае блоки `catch` перехвата обоих исключений можно объединить вместе, как показано ниже. К такому способу перехвата исключений следует прибегать лишь в тех случаях, когда перехватываемые исключения не являются подклассами друг для друга.

```
try
{
    код, способный генерировать исключения
}
```

```
catch (FileNotFoundException | UnknownHostException e)
{
    чрезвычайные действия, если нужные файлы отсутствуют
    или неизвестны хосты
}
catch (IOException e)
{
    чрезвычайные действия во всех остальных
    случаях появления ошибок ввода-вывода
}
```



НА ЗАМЕТКУ! При перехвате нескольких исключений переменная исключения неявно считается конечной (*final*). Например, переменной *e* нельзя присвоить другое значение в следующем блоке:

```
catch (FileNotFoundException | UnknownHostException e) { ... }
```



НА ЗАМЕТКУ! Перехват нескольких исключений не делает код ни проще, ни эффективнее. Формируемые в итоге байт-коды содержат единственный блок общего оператора `catch`.

Повторное генерирование исключений

Исключение можно генерировать и в блоке `catch`, образуя тем самым цепочку исключений. Это обычно делается для того, чтобы изменить тип исключения. Так, если разрабатывается подсистема для применения другими разработчиками, то имеет смысл генерировать такие исключения, которые давали бы возможность сразу определить, что ошибка возникла именно в этой подсистеме. В качестве характерного примера можно привести исключение типа `ServletException`. Возможно, что в коде, где выполняется сервлет, и не обязательно иметь подробные сведения о том, какая именно возникла ошибка, а важно лишь знать, что сервлет работает некорректно. В приведенном ниже фрагменте кода показано, каким образом перехватывается и повторно генерируется исключение.

```
try
{
    получить доступ к базе данных
}
catch (SQLException e)
{
    throw new ServletException("database error: " + e.getMessage());
}
```

В данном случае текст сообщения об исключении формируется в конструкторе класса `ServletException`. Но предыдущее исключение лучше сделать причиной, т.е. источником последующего исключения, как показано ниже.

```
try
{
    получить доступ к базе данных
}
catch (SQLException e)
{
    Throwable se = new ServletException("database error");
    se.initCause(e);
    throw se;
}
```

Теперь при перехвате последующего исключения предыдущее исключение можно извлечь следующим образом:

```
Throwable e = se.getCause();
```

Настоятельно рекомендуется именно такой способ заключения в оболочку. Ведь он позволяет генерировать исключения более высокого уровня, не теряя подробных сведений об исходном исключении.



СОВЕТ. Рассмотренный выше способ заключения в оболочку оказывается удобным в том случае, если перехват исключения осуществляется в методе, которому не разрешается генерировать проверяемые исключения. Проверяемое исключение можно перехватить и заключить его в оболочку исключения времени выполнения.

Иногда требуется зарегистрировать исключение и сгенерировать его повторно без всяких изменений следующим образом:

```
try
{
    получить доступ к базе данных
}
catch (Exception e)
{
    logger.log(level, message, e);
    throw e;
}
```

До версии Java SE 7 такому подходу был присущ существенный недостаток. Допустим, в теле метода имеется следующая строка кода:

```
public void updateRecord() throws SQLException
```

Раньше компилятор анализировал оператор `throw` в блоке `catch`, а затем тип переменной `e` и выдавал предупреждение, что данный метод может сгенерировать любое исключение типа `Exception`, а не только типа `SQLException`. Этот недостаток был устранен в версии Java SE 7. Теперь компилятор проверяет только тот факт, что переменная `e` происходит из блока `try`. Если в данном блоке экземплярами класса `SQLException` являются только проверяемые исключения и содержимое переменной `e` не изменяется в блоке `catch`, то в объявлении объемлющего метода можно с полным основанием указать генерирование исключения следующим образом: `throws SQLException`.

Блок `finally`

Когда в методе генерируется исключение, оставшиеся в нем операторы не выполняются. Если же в методе задействованы какие-нибудь локальные ресурсы, о которых известно лишь ему, то освободить их уже нельзя. Можно, конечно, перехватить и повторно сгенерировать все исключения, но это решение не слишком удачно, поскольку ресурсы нужно освобождать в двух местах: в обычном коде и в коде исключения.

В Java принято лучшее решение — организовать блок `finally`. Ниже приведен пример корректного вызова метода `dispose()` для объекта типа `Graphics`. Работая с базами данных, можно применять тот же самый подход, чтобы правильно разорвать соединение с базой данных. Как будет показано в главе 4 второго тома данной книги, разрывать соединение с базой данных необходимо даже в том случае, если при выполнении программы возникло исключение.

Код в блоке `finally` выполняется независимо от того, возникло исключение или нет. Так, в приведенном ниже примере метод `dispose()` выполняется при любых условиях.

```
InputStream in = new FileInputStream(...);
try
{
    // 1
    код, способный генерировать исключения
    // 2
}
catch (IOException e)
{
    // 3
    вывести сообщение об ошибке
    // 4
}
finally
{
    // 5
    in.close();
}
// 6
```

Рассмотрим три возможных ситуации, в которых программа выполняет блок `finally`.

1. Код не генерирует никаких исключений. В этом случае программа сначала полностью выполняет блок `try`, а затем блок `finally`. Иными словами, выполнение программы последовательно проходит через точки 1, 2, 5 и 6,
2. Код генерирует исключение, которое перехватывается в блоке `catch` (в данном примере это исключение типа `IOException`). В этом случае программа сначала выполняет блок `try` до той точки, в которой возникает исключение, а остальная часть блока `try` пропускается. Затем программа выполняет код из соответствующего блока `catch` и, наконец, код из блока `finally`.
3. Если в блоке `catch` исключения не генерируются, то выполнение программы продолжается с первой строки, следующей после блока `try`. Таким образом, выполнение программы последовательно проходит через точки 1, 3, 4, 5 и 6. Если же исключение генерируется в блоке `catch`, то управление передается вызывающей части программы и выполнение программы проходит только через точки 1, 3 и 5.
4. Код генерирует исключение, которое не обрабатывается в блоке `catch`. В этом случае программа выполняет блок `try` вплоть до той точки, в которой генерируется исключение, а оставшаяся часть блока `try` пропускается. Затем программа выполняет код из блока `finally` и передает исключение обратно вызывающей части программы. Таким образом, выполнение программы проходит только через точки 1 и 5.

Блок `finally` можно использовать и без блока `catch`. Рассмотрим следующий пример кода:

```
InputStream in = ...;
try
```

```

{
    код, способный генерировать исключения
}
finally
{
    in.close();
}

```

Оператор `in.close()` из блока `finally` выполняется независимо от того, возникает ли исключение в блоке `try`. Разумеется, если исключение возникает, оно будет перехвачено в очередном блоке `catch`. Как рекомендуется ниже, конструкцию `finally` на самом деле следует применять всякий раз, когда требуется освободить используемый ресурс.



СОВЕТ. Настоятельно рекомендуется разделять блоки `try/catch` и `try/finally`. В этом случае код программы становится более понятным. Рассмотрим следующий пример кода:

```

InputStream in = ...;
try
{
    try
    {
        код, способный генерировать исключения
    }
    finally
    {
        in.close();
    }
}
catch (IOException e)
{
    вывести сообщение об ошибке
}

```

Здесь внутренний блок `try` отвечает только за закрытие потока ввода, а внешний блок `try` сообщает об ошибках. Такой код не только более понятен, но и более функционален, так как ошибки выявляются и в блоке `finally`.



ВНИМАНИЕ! Если в блоке `finally` присутствует оператор `return`, результаты его выполнения могут быть неожиданными. Допустим, в середине блока `try` происходит возврат из метода с помощью оператора `return`. Перед тем как передать управление вызывающей части программы, следует выполнить блок `finally`. Если и в нем имеется оператор `return`, то первоначально возвращаемое значение будет замаскировано. Рассмотрим следующий пример кода:

```

public static int f(int n)
{
    try
    {
        int r = n * n;
        return r;
    }
    finally
    {
        if (n == 2) return 0;
    }
}

```

Если сделать вызов `f(2)`, то в блоке `try` будет вычислено значение `r = 4` и выполнен оператор `return`. Но на самом деле до оператора `return` будет выполнен код из блока `finally`. В этом блоке из метода принудительно возвращается нулевое значение, игнорируя первоначально вычисленное значение `4`.

Иногда блок `finally` повергает программирующих на Java в полное отчаяние, например, в тех случаях, когда методы очистки памяти генерируют исключения. Предположим, что при возникновении исключения требуется закрыть поток ввода, как показано ниже.

```
InputStream in = ...;
try
{
    код, способный генерировать исключения
}
finally
{
    in.close();
}
```

Допустим также, что код в блоке `try` генерирует некоторое исключение, не относящееся к классу `IOException`. И в этом случае выполняется блок `finally`, в котором вызывается метод `close()`, способный сгенерировать исключение типа `IOException`! В итоге исходное исключение будет потеряно, а вместо него будет сгенерировано исключение в методе `close()`.

Таким образом, при обработке исключений возникает затруднительное положение в связи с тем, что первое из упомянутых выше исключений представляет намного больший интерес. Для того чтобы исправить это положение, придется повторно сгенерировать исходное исключение, усложнив в конечном итоге исходный код. Ниже показано, как лучше всего выйти из столь затруднительного положения.

```
InputStream in = ...;
Exception ex = null;
try
{
    try
    {
        код, способный генерировать исключения
    }
    catch (Exception e)
    {
        ex = e;
        throw e;
    }
}
finally
{
    try
    {
        in.close();
    }
    catch (Exception e)
    {
        if (ex == null) throw e;
    }
}
```

Правда, в версии Java SE 7 внедрено языковое средство, намного упрощающее задачу освобождения используемых ресурсов. Это средство обсуждается в следующем разделе.

Оператор `try` с ресурсами

В Java SE 7 внедрена следующая удобная конструкция, упрощающая код обработки исключений, в котором требуется освобождать используемые ресурсы:

```
открыть ресурс
try
{
    использовать ресурс
}
finally
{
    закрыть ресурс
}
```

Следует, однако, иметь в виду, что эта конструкция оказывается эффективной при одном условии: используемый ресурс принадлежит к классу, реализующему интерфейс `AutoCloseable`. В этом интерфейсе имеется единственный метод, объявляемый следующим образом:

```
void close() throws Exception
```



НА ЗАМЕТКУ! Имеется также интерфейс `Closeable`, производный от интерфейса `AutoCloseable`. В нем также присутствует единственный метод `close()`. Но этот метод объявляется для генерирования исключения типа `IOException`.

В своей простейшей форме оператор `try` с ресурсами выглядит следующим образом:

```
try (Resource res = ...)
{
    использовать ресурс res
}
```

Если в коде присутствует блок `try`, метод `res.close()` вызывается автоматически. Ниже приведен типичный тому пример ввода всего текста из файла и последующего его вывода.

```
try (Scanner in = new Scanner(new FileInputStream("/usr/share/dict/words")))
{
    while (in.hasNext())
        System.out.println(in.next());
}
```

Происходит ли выход из блока `try` нормально, или же в нем возникает исключение, метод `in.close()` вызывается в любом случае, как и при использовании блока `finally`. В блоке `try` можно также указать несколько ресурсов, как в приведенном ниже примере кода.

```
try (Scanner in = new Scanner(new FileInputStream("/usr/share/dict/words")),
    PrintWriter out = new PrintWriter("out.txt"))
{
    while (in.hasNext())
        out.println(in.next().toUpperCase());
}
```

Таким образом, независимо от способа завершения блока `try` оба потока ввода и вывода благополучно закрываются. Если бы такое освобождение ресурсов пришлось программировать вручную, для этого пришлось бы составлять вложенные блоки `try/finally`.

Как было показано ранее, трудности возникают в том случае, если исключение генерируется не только в блоке `try`, но и в методе `close()`. Оператор `try` с ресурсами предоставляет довольно изящный выход из столь затруднительного положения. Исходное исключение генерируется повторно, а любые исключения, генерируемые в методе `close()`, считаются “подавленными”. Они автоматически перехватываются и добавляются к исходному исключению с помощью метода `addSuppressed()`. И если они представляют какой-то интерес с точки зрения обработки, то следует вызвать метод `getSuppressed()`, предоставляющий массив подавленных исключений из метода `close()`.

Но самое главное, что все это не нужно программировать вручную. Всякий раз, когда требуется освободить используемый ресурс, достаточно применить оператор `try` с ресурсами.



НА ЗАМЕТКУ! Оператор `try` с ресурсами может также иметь сопутствующие операторы `catch` и `finally`. Блоки этих операторов выполняются после освобождения используемых ресурсов. Но на практике вряд ли стоит нагромождать столько кода в единственном блоке оператора `try` с ресурсами.

Анализ элементов трассировки стека

Трассировка стека — это список вызовов методов в данной точке выполнения программы. Вам, скорее всего, не раз приходилось видеть подобные списки. Ведь они выводятся всякий раз, когда при выполнении программы на Java возникает непроверяемое или необрабатываемое исключение.

Для получения текстового описания трассировки стека достаточно вызвать метод `printStackTrace()` из класса `Throwable`, как показано ниже.

```
Throwable t = new Throwable();
ByteArrayOutputStream out = new ByteArrayOutputStream();
t.printStackTrace(out);
String description = out.toString();
```

Более удобный способ состоит в вызове метода `getStackTrace()`, возвращающего массив объектов типа `Stack TraceElement`, которые можно проанализировать в программе, как в следующем примере кода:

```
Throwable t = new Throwable();
Stack TraceElement[] frames = t.getStackTrace();
for (Stack TraceElement frame : frames)
    проанализировать фрейм
```

Класс `Stack TraceElement` содержит методы, позволяющие получить имя файла, номер исполняемой строки кода, а также имя класса и метода. Отформатированную строку, содержащую эти сведения, предоставляет метод `toString()`. А метод `Thread.getAllStack Traces()` позволяет получить трассировку стека во всех потоках. Его применение демонстрируется в следующем фрагменте кода:

```
Map<Thread, Stack TraceElement[]> map = Thread.getAllStack Traces();
for (Thread t : map.keySet())
```

```
{
    StackTraceElement[] frames = map.get(t);
    проанализировать фрейм
}
```

Более подробные сведения об интерфейсе Map и потоках приведены в главах 13 и 14 соответственно. В листинге 11.1 представлен исходный код программы, в которой выводится трассировка стека для рекурсивной функции вычисления факториала. Например, при вычислении факториала числа 3 получаются следующие результаты трассировки стека вызова factorial(3):

```
factorial(3):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.main(StackTraceTest.java:34)
factorial(2):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
factorial(1):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
return 1
return 2
return 6
```

Листинг 11.1. Исходный код из файла stackTrace/StackTraceTest.java

```
1 package stackTrace;
2
3 import java.util.*;
4
5 /**
6  * В этой программе выводится трассировка
7  * стека вызовов рекурсивного метода
8  * @version 1.01 2004-05-10
9  * @author Cay Horstmann
10 */
11 public class StackTraceTest
12 {
13     /**
14      * Вычисляет факториал заданного числа
15      * @param n Положительное целое число
16      * @return n! = 1 * 2 * . . . * n
17      */
18     public static int factorial(int n)
19     {
20         System.out.println("factorial(" + n + "):");
21         Throwable t = new Throwable();
22         StackTraceElement[] frames = t.getStackTrace();
23         for (StackTraceElement f : frames)
24             System.out.println(f);
25         int r;
```

```
26     if (n <= 1) r = 1;
27     else r = n * factorial(n - 1);
28     System.out.println("return " + r);
29     return r;
30 }
31 public static void main(String[] args)
32 {
33     Scanner in = new Scanner(System.in);
34     System.out.print("Enter n: ");
35     int n = in.nextInt();
36     factorial(n);
37 }
38 }
```

java.lang.Throwable 1.0

- `Throwable(Throwable cause) 1.4`
- `Throwable(String message, Throwable cause) 1.4`
Создают объект типа `Throwable` по заданному объекту `cause`, который описывает причины возникновения исключения.
- `Throwable initCause(Throwable cause) 1.4`
Указывает причину создания данного объекта. Если причина создания объекта уже указана, генерирует исключение. Возвращает ссылку `this`.
- `Throwable getCause() 1.4`
Возвращает объект исключения, который был указан в качестве причины для создания данного объекта. Если причина создания данного объекта не была указана, возвращается пустое значение `null`.
- `StackTraceElement[] getStackTrace() 1.4`
Возвращает трассировку стека в момент создания объекта.
- `void addSuppressed(Throwable t) 7`
Добавляет к данному исключению "подавленное" исключение. Это происходит в операторе `try` с ресурсами, где `t` — исключение, генерируемое методом `close()`.
- `Throwable[] getSuppressed() 7`
Получает все исключения, "подавленные" данным исключением. Как правило, это исключения, генерируемые методом `close()` в операторе `try` с ресурсами.

java.lang.Exception 1.0

- `Exception(Throwable cause) 1.4`
- `Exception(String message, Throwable cause)`
Создают объект типа `Exception` по заданному объекту `cause`.

java.lang.RuntimeException 1.0

- RuntimeException(Throwable cause) 1.4
 - RuntimeException(String message, Throwable cause) 1.4
- Создают объект типа RuntimeException по заданному объекту cause.

java.lang.StackTraceElement 1.4

- String getFileName()
Получает имя исходного файла, содержащего точку, в которой выполняется данный элемент. Если эти сведения недоступны, возвращается пустое значение null.
- int getLineNumber()
Получает номер строки кода, содержащей точку, в которой выполняется данный элемент. Если эти сведения недоступны, возвращается значение -1.
- String getClassName()
Возвращает полное имя класса, содержащего точку, в которой выполняется данный элемент.
- String getMethodName()
Возвращает имя метода, содержащего точку, в которой выполняется данный элемент. Имя конструктора — <init>, а имя статического инициализатора — <clinit>. Перегружаемые методы с одинаковыми именами не различаются.
- boolean isNativeMethod()
Возвращает логическое значение true, если точка выполнения данного элемента находится в собственном методе.
- String toString()
Возвращает отформатированную символьную строку, содержащую имя класса и метода, а также имя файла и номер строки кода (если эти сведения доступны).

Рекомендации по обработке исключений

Разные специалисты придерживаются различных мнений относительно обработки исключений. Одни считают, что проверяемые исключения — это не более чем досадная помеха, а другие готовы затратить дополнительное время и труд на их обработку. На наш взгляд, исключения (особенно проверяемые) — полезный механизм, но, применяя его, не стоит слишком увлекаться. В этом разделе мы дадим ряд рекомендаций относительно применения и обработки исключений в прикладных программах.

1. *Обработка исключений не может заменить собой простую проверку.*

Для иллюстрации этого положения мы написали фрагмент кода, в котором используется встроенный класс Stack. В этом коде делается 10 млн. попыток извлечь элемент из пустого стека. Сначала в нем выясняется, пуст ли стек:

```
if (!s.empty()) s.pop();
```

Затем элемент принудительно извлекается из стека, независимо от того, пуст ли он или заполнен, как показано ниже. После этого перехватывается исключение типа `EmptyStackException`, которое предупреждает, что так делать нельзя.

```
try()
{
    s.pop();
}
catch (EmptyStackException e)
{
}
```

Время, затраченное на нашей тестовой машине на вызов метода `isEmpty()`, составило **646** миллисекунд, а на перехват исключения типа `EmptyStackException` — **21739** миллисекунд.

Как видите, перехват исключения занял намного больше времени, чем простая проверка. Из этого следует такой вывод: пользуйтесь исключениями только в тех случаях, когда это оправданно, что зачастую бывает лишь в исключительных ситуациях.

2. Не доводите обработку исключений до абсурдных мелочей.

Многие программисты заключают едва ли не каждый оператор в отдельный блок `try`. Ниже показано, к чему приводит подобная мелочность при программировании на Java.

```
PrintStream out;
Stack s;
for (i = 0; i < 100; i++)
{
    try
    {
        n = s.pop();
    }
    catch (EmptyStackException e)
    {
        // стек оказался пустым
    }
    try
    {
        out.writeInt(n);
    }
    catch (IOException e)
    {
        // сбой при записи данных в файл
    }
}
```

При таком подходе объем кода значительно увеличивается. Поэтому хорошенько подумайте о той задаче, которую должна решать ваша программа. В данном случае требуется извлечь из стека **100** чисел и записать их в файл. (Неважно, зачем это нужно. Ведь это всего лишь пример.) Одно только генерирование исключения не является выходом из положения. Ведь если стек пуст, он не заполнится по мановению волшебной палочки. А если при записи числовых данных в файл возникает ошибка, то она не исчезнет сама собой. Следовательно, имеет смысл разместить в блоке `try` *весь* фрагмент кода для решения поставленной задачи, как показано ниже. Если хотя бы одна из операций в этом блоке даст сбой, можно отказаться от решения всей задачи, а не отдельных ее частей.

```
try
{
    for (i = 0; i < 100; i++)
    {
        n = s.pop();
        out.writeInt(n);
    }
}
catch (IOException e)
{
    // сбой при записи данных в файл
}
catch (EmptyStackException e)
{
    // стек оказался пустым
}
```

Этот фрагмент кода уже выглядит намного яснее. Он выполняет одно из своих основных назначений — *отделяет* нормальную обработку данных от обработки исключений.

3. *Правильно пользуйтесь возможностями, которые предоставляет иерархия наследования исключений.*

Не ограничивайтесь генерированием только исключения типа `RuntimeException`. Найдите подходящий подкласс или создайте собственный. Не перехватывайте исключение типа `Throwable`. При таком подходе ваш код становится трудным для понимания и сопровождения.

Правильно различайте проверяемые и непроверяемые исключения. Для обработки проверяемых исключений требуются дополнительные усилия, поэтому не применяйте их для уведомления о логических ошибках. (В библиотеке, поддерживающей механизм рефлексии, это правило не соблюдается. Поэтому ее пользователям часто приходится писать код для перехвата тех исключений, которые могут вообще не возникнуть.)

В случае необходимости смело преобразуйте один тип исключения в другой, более подходящий в данной ситуации. Так, если вы выполняете синтаксический анализ целого значения, вводимого из файла, перехватывайте исключение класса `NumberFormatException` и преобразуйте его в исключение подкласса, производного от класса `IOException` или `MySubsystemException`, для последующего генерирования.

4. *Не подавляйте исключения.*

При программировании на Java существует большой соблазн подавить исключения. Допустим, требуется написать метод, вызывающий другой метод, который может сгенерировать исключение один раз в сто лет. Компилятор предупредит об этом, поскольку исключение не указано в списке оператора `throws` при объявлении данного метода. Но если нет никакого желания указывать его в списке оператора `throws`, чтобы компилятор не выдавал сообщения об ошибках во всех методах, вызывающих данный метод, такое исключение можно просто подавить следующим образом:

```
public Image loadImage(String s)
{
    try
    {
```

```

    код, способный генерировать проверяемые исключения
}
catch (Exception e)
{} // ничего не делать!
}

```

Теперь код будет скомпилирован. Он будет прекрасно работать, но не в исключительной ситуации. А если она возникнет, то будет просто проигнорирована. Но если вы все-таки считаете, что подобные исключения важны, приложите усилия для организации их обработки.

5. *Обнаруживая ошибки, проявляйте необходимую твердость вместо излишней терпимости.*

Некоторые программисты избегают генерировать исключения, обнаруживая ошибки. Так, если методу передаются некорректные параметры, они предпочитают возвращать фиктивное значение. Когда, например, стек пуст, программист может предусмотреть возврат из метода `Stack.pop()` пустого значения `null` вместо того, чтобы генерировать исключение. На наш взгляд, лучше сгенерировать исключение типа `EmptyStackException` в той точке, где возникла ошибка, чем исключение типа `NullPointerException` впоследствии.

6. *Не бойтесь передавать исключения для обработки в коде, разрабатываемом другими.*

Некоторые программисты считают себя просто обязанными перехватывать все исключения. Вызывая метод, генерирующий некоторое исключение, например конструктор класса `FileInputStream` или метод `readLine()`, они инстинктивно перехватывают исключения, которые могут быть при этом сгенерированы. Но зачастую предпочтительнее *передать* исключение другому обработчику, а не обрабатывать его самостоятельно, как показано ниже.

```

public void readStuff(String filename)
    throws IOException // Ничтоже сумняшеся!
{
    InputStream in = new FileInputStream(filename);
    . . .
}

```

Методы более высокого уровня лучше оснащены средствами уведомления пользователей об ошибках или отмены выполнения неверных операций.



НА ЗАМЕТКУ! Рекомендации в пп.5 и 6 можно свести к следующему общему правилу: генерировать исключения раньше, а перехватывать их позже.

Применение утверждений

Утверждения — широко распространенное средство так называемого *безопасного* программирования. Допустим, вы твердо знаете, что конкретное свойство уже задано, и обращаетесь к нему в своей программе. Например, при вычислении следующего выражения вы уверены в том, что значение параметра **x** не является отрицательным:

```
double y = Math.sqrt(x);
```

Возможно, оно получено в результате других вычислений, которые не могут породить отрицательные числовые значения, или же является параметром метода,

которому можно передавать только положительные числовые значения. Но вы все же хотите еще раз проверить свои предположения, чтобы в ходе выполнения программы не возникло ошибки. Разумеется, можно было бы просто сгенерировать исключение следующим образом:

```
if (x < 0) throw new IllegalArgumentException("x < 0");
```

Но у такого подхода имеется следующий существенный недостаток: приведенный выше код остается в программе, даже по завершении ее тестирования. Если в исходном коде программы расставлено много таких проверок, ее выполнение может замедлиться. Механизм утверждений позволяет вводить в исходный код программы проверки для ее тестирования, а затем удалять их из окончательного варианта. Для этой цели имеется ключевое слово `assert`. Ниже приведены две основные его формы.

```
assert условие;
```

и

```
assert условие : выражение;
```

Оба оператора проверяют заданное *условие* и генерируют исключение типа `AssertionError`, если оно не выполняется. Во втором операторе **выражение** передается конструктору объекта типа `AssertionError` и преобразуется в символьную строку сообщения.



НА ЗАМЕТКУ! Единственная цель, которую преследует *выражение* в операторе `assert`, — получить символьную строку сообщения. В объекте типа `AssertionError` конкретное значение выражения не хранится, поэтому его нельзя запросить в дальнейшем. В документации на JDK снисходительно сообщается, что это сделано для того, чтобы “стимулировать программистов устранять ошибки, выявляемые в результате непрохождения тестов на утверждения, которые противоречат основному назначению программных средств”.

Чтобы проверить, является ли числовое значение переменной `x` неотрицательным, достаточно написать следующий оператор:

```
assert x >= 0;
```

Кроме того, можно передать конкретное значение переменной `x` объекту типа `AssertionError`, чтобы впоследствии вывести его, как показано ниже.

```
assert x >= 0 : x;
```



НА ЗАМЕТКУ C++! Макрокоманда `assert` в C преобразует проверяемое условие в символьную строку, которая выводится, если условие не выполняется. Так, если выражение `assert (x>=0)` имеет логическое значение `false`, выводится символьная строка, сообщающая, что условие `x >= 0` не выполняется. В Java условие не включается автоматически в сообщение об ошибке. Если же условие требуется отобразить, его можно передать в виде символьной строки объекту типа `AssertionError` следующим образом: `assert x >= 0 : "x >= 0"`.

Разрешение и запрет утверждений

По умолчанию утверждения запрещены. Разрешить их в ходе выполнения программы можно с помощью параметра `-enableassertions` или `-ea`, указываемого в командной строке следующим образом:

```
java -enableassertions MyApp
```

Следует, однако, иметь в виду, что для разрешения и запрета утверждений компилировать программу заново не нужно. Это задача *загрузчика классов*. Если диагностические утверждения запрещены, загрузчик классов проигнорирует их код, чтобы не замедлять выполнение программы.

Разрешать утверждения можно в отдельных классах или в целых пакетах следующим образом:

```
java -ea:MyClass -ea:com.mycompany.mylib ... MyApp
```

Эта команда разрешает утверждения в классе `MyClass`, а также во всех классах из пакета `com.mycompany.mylib` и его *подчиненных* пакетов. Параметр `-ea...` разрешает утверждения во всех классах из пакета, выбираемого по умолчанию.

Кроме того, утверждения можно запретить в определенных классах и пакетах. Для этого достаточно указать параметр `-disableassertions` или `-da` в командной строке следующим образом:

```
java -ea:... -da:MyClass MyApp
```

Некоторые классы загружаются не загрузчиком классов, а непосредственно виртуальной машиной. Для выборочного разрешения или запрета утверждений, содержащихся в этих классах, можно также указать параметр `-ea` и `-da` в командной строке. Но эти параметры разрешают и запрещают все утверждения одновременно, и поэтому их нельзя применять в "системных классах" без загрузчика классов. Для разрешения утверждений, находящихся в системных классах, следует указать параметр `-enablesystemassertions/-esa`. Имеется также возможность программно управлять состоянием утверждений. Подробнее об этом можно узнать из документации на прикладной интерфейс API.

Проверка параметров с помощью утверждений

В Java предусмотрены следующие три механизма обработки системных сбоев.

- Генерирование исключений.
- Протоколирование.
- Применение утверждений.

Когда же следует применять утверждения? Чтобы правильно ответить на этот вопрос, необходимо учесть следующее.

- Утверждения оказываются ложными, если произошла неустранимая ошибка.
- Утверждения разрешаются только на время отладки и тестирования программ. (Иногда это положение шутливо формулируют так: надевайте спасательный жилет, плавая у берега, а в открытом море он вам все равно не поможет.)

Итак, утверждения не следует применять в качестве индикаторов несущественных и легко исправимых ошибок. Они предназначены для выявления серьезных неполадок во время тестирования.

Рассмотрим типичную ситуацию, возникающую при проверке параметров метода. Следует ли применять утверждения, чтобы проверить, не выходит ли индекс за допустимые пределы, или выявить пустую ссылку? Чтобы ответить на этот вопрос,

нужно обратиться к документации, описывающей данный метод. В качестве примера рассмотрим метод `Array.sort()` из стандартной библиотеки.

```
/**
 * Упорядочивает указанную часть заданного массива по нарастающей.
 * Упорядочиваемая часть массива начинается с индекса fromIndex и
 * заканчивается индексом, предшествующим toIndex, т.е. элемент с
 * индексом toIndex упорядочению не подлежит.
 * @param a Упорядочиваемый массив
 * @param fromIndex Индекс первого элемента упорядочиваемой части
 *                 массива (включительно)
 * @param toIndex Индекс первого элемента, находящегося за пределами
 *                упорядочиваемой части массива
 * @throws IllegalArgumentException Если fromIndex > toIndex,
 *                                генерируется исключение
 * @throws ArrayIndexOutOfBoundsException Если fromIndex < 0
 *                                или toIndex > a.length
 *                                генерируется исключение
 */
static void sort[] a, int fromIndex, int toIndex
```

В документации утверждается, что этот метод генерирует исключение, если значение индекса задано неверно. Это поведение метода является частью контракта, который он заключает с вызывающим методом. Если вы реализуете этот метод, то должны придерживаться данного контракта и генерировать соответствующее исключение. В этом случае утверждения не подходят.

Следует ли выявлять с помощью утверждений пустые ссылки? Как и прежде, ответ отрицательный. В документации на рассматриваемый здесь метод ничего не говорится о его реакции в том случае, если параметр `a` имеет пустое значение `null`. В вызывающей части программы имеются все основания предполагать, что в этом случае метод будет выполнен и не сгенерирует исключение типа `AssertionError`. Допустим, однако, что в контракт метода внесено следующее изменение:

```
@param a Упорядочиваемый массив. (Не должен быть пустым)
```

Теперь в вызывающей части программы известно, что передавать пустой массив рассматриваемому здесь методу запрещено. Поэтому данный метод может начинаться со следующего утверждения:

```
assert(a != null);
```

Специалисты называют подобный контракт *предусловием*. В исходном методе на параметры не накладывалось никаких предусловий, так как предполагалось, что он будет правильно действовать во всех случаях. А в видоизмененном методе накладывается следующее предусловие: параметр `a` не должен принимать пустое значение `null`. Если в вызывающей части программы данное предусловие нарушается, то контракт не соблюдается и метод не обязан его выполнять. На самом деле, имея утверждение, метод может повести себя непредсказуемо, если он вызывается неверно. В одних случаях он может сгенерировать исключение типа `AssertionError`, а в других — исключение типа `NullPointerException`, что зависит от конфигурации загрузчика классов.

Документирование предположений с помощью утверждений

Многие программисты оформляют свои предположения о работе программы в виде комментариев. Рассмотрим следующий пример из документации, доступной

по адресу <http://docs.oracle.com/javase/6/docs/technotes/guides/language/assert.html>:

```
if (i % 3 == 0)
    . . .
else if (i % 3 == 1)
    . . .
else // (i % 3 == 2)
```

В данном случае намного больше оснований использовать утверждения следующим образом:

```
if (i % 3 == 0)
    . . .
else if (i % 3 == 1)
    . . .
else
{
    assert i % 3 == 2;
    . . .
}
```

Конечно, было бы разумно тщательнее сформулировать проверки. В частности, какие значения может принимать выражение $i \% 3$? Если переменная i принимает положительное числовое значение, остаток может быть равным 0, 1 или 2. А если она принимает отрицательное значение, то остаток может быть равным -1 или -2. Таким образом, намного логичнее предположить, что переменная i не содержит отрицательное числовое значение. Для этого можно составить следующее утверждение, введя его перед условным оператором `if`:

```
assert(i >= 0);
```

Во всяком случае, рассмотренный выше пример показывает, что утверждения удобны для самопроверки в процессе программирования. Они являются тактическим средством для тестирования и отладки, тогда как протоколирование — стратегический механизм, сопровождающий программу на протяжении всего срока ее существования. Подробнее о протоколировании речь пойдет в следующем разделе.

java.lang.ClassLoader 1.0

- `void setDefaultAssertionStatus(boolean b) 1.4`
Разрешает или запрещает утверждения во всех классах, загруженных указанным загрузчиком классов, в которых состояние утверждений не задано явным образом.
- `void setClassAssertionStatus(String className, boolean b) 1.4`
Разрешает или запрещает утверждения в заданном классе и его подклассах.
- `void setPackageAssertionStatus(String packageName, boolean b) 1.4`
Разрешает или запрещает утверждения во всех классах заданного пакета и его подчиненных пакетов.
- `void clearAssertionStatus() 1.4`
Удаляет все утверждения, состояние которых задано явным образом, а также запрещает все утверждения в классах, загруженных данным загрузчиком.

Протоколирование

Программирующие на Java часто прибегают к вызовам метода `System.out.println()`, чтобы отладить код и уяснить поведение программы. Разумеется, после устранения ошибки следует убрать промежуточный вывод и ввести его в другом месте программы для выявления очередной неполадки. С целью упростить отладку программ в подобном режиме предусмотрен прикладной интерфейс API для протоколирования. Ниже перечислены основные преимущества его применения.

- Все протокольные записи нетрудно запретить или разрешить.
- Запрещенные протокольные записи отнимают немного ресурсов и не влияют на эффективность работы приложения.
- Протокольные записи можно направить разным обработчикам, вывести на консоль, записать в файл и т.п.
- Регистраторы и обработчики могут фильтровать записи. Фильтры отбрасывают ненужные записи по критериям, предоставляемым теми, кто реализует фильтры.
- Протокольные записи допускают форматирование. Их можно, например, представить в виде простого текста ли в формате XML.
- В приложениях можно использовать несколько протоколов, имеющих иерархические имена, подобные именам пакетов, например `com.mycompany.myapp`.
- По умолчанию параметры настройки протоколирования задаются в конфигурационном файле. Такой способ задания параметров может быть изменен в приложении.

Элементарное протоколирование

Для элементарного протоколирования служит глобальный регистратор. Он вызывается следующим образом:

```
Logger.getGlobal().info("File->Open menu item selected");
```

По умолчанию выводится следующая протокольная запись:

```
May 10, 2013 10:12:15 PM LoggingImageViewer fileOpen  
INFO: File->Open menu item selected
```

Но если в соответствующем месте программы, например, в начале метода `main()`, ввести приведенную ниже строку кода, протокольная запись будет запрещена.

```
Logger.getGlobal().setLevel(Level.OFF)
```



ВНИМАНИЕ! До тех пор, пока не будет устранена программная ошибка за номером **7184195**, придется делать вызов `Logger.getGlobal().setLevel(Level.INFO)`, чтобы активизировать глобальный регистратор.

Усовершенствованное протоколирование

Итак, рассмотрев элементарное протоколирование, перейдем к средствам протоколирования, применяемым при создании программ по промышленным стандартам.

В профессионально разработанном приложении, как правило, все записи не накапливаются в одном глобальном протоколе. Вместо этого можно определить свои собственные средства протоколирования. Так, для создания или извлечения регистратора вызывается метод `getLogger()`, как показано ниже.

```
private static final Logger myLogger =  
Logger.getLogger("com.mycompany.myapp");
```

Как и имена пакетов, имена регистраторов образуют иерархию. На самом деле они являются еще *более* иерархическими, чем имена пакетов. Если между пакетом и его предшественником нет никакой семантической связи, то регистратор и производные от него регистраторы обладают общими свойствами. Так, если в регистраторе `com.mycompany` задан определенный уровень протоколирования, то производный от него регистратор унаследует этот уровень.

Существует семь уровней протоколирования:

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

По умолчанию используются первые три уровня. Остальные уровни нужно задавать, вызывая метод `setLevel()` следующим образом:

```
logger.setLevel(Level.FINE);
```

В итоге будут регистрироваться все сообщения, начиная с уровня `FINE` и выше. Кроме того, для разрешения протоколирования на всех уровнях можно воспользоваться константой `Level.ALL`, а константой `Level.OFF` — для запрета протоколирования. Для всех уровней определены методы протоколирования, как показано в приведенном ниже примере.

```
logger.warning(message);  
logger.fine(message);
```

С другой стороны, можно воспользоваться методом `log()`, чтобы явно указать нужный уровень:

```
logger.log(Level.FINE, message);
```



СОВЕТ. По умолчанию регистрируются все записи, имеющие уровень `INFO` и выше. Следовательно, для отладочных сообщений, требующихся для диагностики программ, но совершенно не нужных пользователям, следует указывать уровни `CONFIG`, `FINE`, `FINER` и `FINEST`.



ВНИМАНИЕ! Если уровень протоколирования превышает величину `INFO`, следует изменить конфигурацию обработчика протоколов. По умолчанию обработчик протоколов блокирует сообщения, имеющие уровень ниже `INFO`.

Протокольная запись, создаваемая по умолчанию, состоит из имени класса и метода, содержащего вызов регистратора. Но если виртуальная машина оптимизирует процесс исполнения кода, то точные сведения о вызываемых методах могут стать недоступными. Для уточнения вызывающего класса и метода следует применить метод `logp()` следующим образом:

```
void logp(Level l, String className, String methodName, String message)
```

Для отслеживания порядка выполнения диагностируемой программы имеются следующие служебные методы:

```
void entering(String className, String methodName)
void entering(String className, String methodName, Object param)
void entering(String className, String methodName, Object[] params)
void exiting(String className, String methodName)
void exiting(String className, String methodName, Object result)
```

Ниже приведен пример применения этих методов непосредственно в коде. При их вызове формируются протокольные записи, имеющие уровень `FINER` и начинающиеся с символьных строк `ENTRY` и `RETURN`.

```
int read(String file, String pattern)
{
    logger.entering("com.mycompany.mylib.Reader", "read",
        new Object[] { file, pattern });
    . . .
    logger.exiting("com.mycompany.mylib.Reader", "read", count);
    return count;
}
```



НА ЗАМЕТКУ! В будущем методы протоколирования будут поддерживать переменное число параметров. Тогда появится возможность делать вызовы вроде следующего: `logger.entering("com.mycompany.mylib.Reader", "read", file, pattern)`.

Обычно протоколирование служит для записи неожиданных исключений. Для этого имеются два удобных метода, позволяющих ввести имя исключения в протокольную запись следующим образом:

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)
```

Ниже приведены типичные примеры применения этих методов в коде. При вызове метода `throwing()` регистрируется протокольная запись, имеющая уровень `FINER`, а также сообщение, начинающееся со строки `THROW`.

```
if (. . .)
{
    IOException exception = new IOException(". . .");
    logger.throwing("com.mycompany.mylib.Reader", "read", exception);
    throw exception;
}
и
try
{
    . . .
}
catch (IOException e)
{
    Logger.getLogger("com.mycompany.myapp").
        log(Level.WARNING, "Reading image", e);
}
```

Смена диспетчера протоколирования

Свойства системы протоколирования можно изменить, редактируя конфигурационный файл. По умолчанию этот файл находится по следующему пути: `jure/lib/logging.properties`

Если требуется сменить конфигурационный файл, при запуске приложения нужно установить свойство `java.util.logging.config.file` следующим образом:

```
java -Djava.util.logging.config.file=конфигурационный_файл ГлавныйКласс
```



ВНИМАНИЕ! Диспетчер протоколирования инициализируется во время запуска виртуальной машины и перед выполнением метода `main()`. Если же сделать вызов `System.setProperty("java.util.logging.config.file", file)` в методе `main()`, то для повторной инициализации диспетчера протоколирования придется также вызвать метод `LogManager.readConfiguration()`.

Чтобы изменить уровень протоколирования, принятый по умолчанию, нужно отредактировать конфигурационный файл, изменив в нем следующую строку:

```
.level=INFO
```

А в собственных регистраторах уровни протоколирования задаются с помощью строк вроде приведенной ниже. Иными словами, нужно добавить к имени регистратора суффикс `.level`.

```
com.mycompany.myapp.level=FINE
```

Как станет ясно в дальнейшем, регистраторы на самом деле не направляют сообщения на консоль, поскольку это задача обработчиков протоколов. Для обработчиков протоколов также определены уровни. Чтобы вывести на консоль сообщения, имеющие уровень `FINE`, необходимо сделать следующую установку:

```
java.util.logging.ConsoleHandler.level=FINE
```



ВНИМАНИЕ! Параметры настройки диспетчера протоколирования не являются системными свойствами. Запуск программы с параметром `-Dcom.mycompany.myapp.level=FINE` никак не отражается на действиях регистратора.



ВНИМАНИЕ! До версии Java SE 7 в документации на прикладной интерфейс API для класса `LogManager` требовалась установка свойств `java.util.logging.config.class` и `java.util.logging.config.file` средствами прикладного интерфейса API для сохранения глобальных параметров настройки. На самом деле это не так — см. описание программной ошибки за номером **4691587** по адресу <http://bugs.sun.com/bugdatabase>.



НА ЗАМЕТКУ! Файл, содержащий свойства системы протоколирования, обрабатывается классом `java.util.logging.LogManager`. Задав имя подкласса в качестве системного свойства `java.util.logging.manager`, можно указать другой диспетчер протоколирования. Кроме того, можно оставить стандартный диспетчер протоколирования, пропустив инициализационные установки в файле свойств. Подробное описание класса `LogManager` содержится в документации на прикладной интерфейс API.

Уровни протоколирования в работающей программе можно изменять, используя утилиту `jconsole`. Подробнее об этом можно узнать по адресу www.oracle.com/technetwork/articles/java/jconsole-1564139.html#LoggingControl.

Интернационализация

Нередко возникает потребность обеспечить вывод протокольных сообщений на разных языках, чтобы они стали понятны пользователям из других стран. Интернационализация прикладных программ обсуждается в главе 5 второго тома данной книги. Ниже вкратце поясняется, что следует иметь в виду при интернационализации протокольных сообщений.

Данные, требующиеся для интернационализации приложения, содержатся в комплектах ресурсов. Комплект ресурсов представляет собой набор сопоставлений для различных региональных настроек (например, в США или Германии). Например, в комплекте ресурсов можно сопоставить символьную строку `"readingFile"` с текстовой строкой `"Reading file"` на английском языке или со строкой `«Achtung! Datei wird eingelesen»` на немецком языке.

В состав прикладной программы может входить несколько комплектов ресурсов, например, один — для меню, другой — для протокольных сообщений. У каждого комплекта ресурсов имеется свое имя (например, `"com.mycompany.logmessages"`). Для того чтобы ввести сопоставление в комплект ресурсов, следует предоставить файл для региональных настроек на соответствующем языке. Сопоставления сообщений на английском языке находятся в файле `com/mycompany/logmessages_en.properties`, а сопоставления сообщений на немецком языке — в файле `com/mycompany/logmessages_de.properties`. (Здесь пары символов **en** и **de** обозначают стандартные коды языков.) Объединение файлов осуществляется с помощью классов прикладной программы. В частности, класс `ResourceBundle` обнаруживает их автоматически. Содержимое этих файлов состоит из записей простым текстом, подобных приведенным ниже.

```
readingFile=Achtung! Datei wird eingelesen
renamingFile=Datei wird umbenannt
. . .
```

При вызове регистратора сначала указывается конкретный комплект ресурсов:

```
Logger logger = Logger.getLogger(loggerName, "com.mycompany.logmessages");
```

Затем для составления протокольного сообщения указывается ключ из комплекта ресурсов, но не строка самого сообщения, как показано ниже.

```
logger.info("readingFile");
```

Нередко возникает потребность включать какие-нибудь аргументы в интернационализированные сообщения. В таком случае сообщение должно иметь заполнители `{0}`, `{1}` и т.д. Допустим, в протокольное сообщение требуется ввести имя файла. С этой целью заполнитель используется следующим образом:

```
Reading file {0}.
Achtung! Datei {0} wird eingelesen.
```

Заполнители заменяются соответствующими значениями при вызове одного из следующих методов:

```
logger.log(Level.INFO, "readingFile", fileName);
logger.log(Level.INFO, "renamingFile", new Object[] { oldName, newName });
```

Обработчики протоколов

По умолчанию регистраторы посылают протокольные записи объекту класса `ConsoleHandler`, который выводит их в поток сообщений об ошибках `System.err`. Точнее говоря, регистраторы посылают протокольные записи родительскому обработчику протоколов, а у его окончательного предшественника (под именем `""`) имеется объект типа `ConsoleHandler`.

Подобно регистраторам, у обработчиков протоколов имеются свои уровни. Уровень протоколирования записи должен превышать порог как для регистратора, так и для обработчика. Уровень консольного обработчика по умолчанию задается в файле, содержащем параметры настройки диспетчера протоколирования, как показано ниже.

```
java.util.logging.ConsoleHandler.level=INFO
```

Для регистрации записи, имеющей уровень `FINE`, в конфигурационном файле следует изменить исходный уровень протоколирования как регистратора, так и обработчика. С другой стороны, можно вообще пренебречь файлом конфигурации и установить свой собственный обработчик протоколов следующим образом:

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

По умолчанию регистратор посылает протокольные записи как своим, так и родительским обработчикам. Рассматриваемый здесь регистратор наследует от начального регистратора (под именем `""`), посылающего на консоль все протокольные записи уровня `INFO` и выше. Но вряд ли нужно, чтобы все протокольные записи выводились дважды, поэтому в свойстве `useParentHandlers` следует установить логическое значение `false`.

Для того чтобы послать протокольную запись еще куда-нибудь, следует ввести другой обработчик протоколов. С этой целью в прикладном интерфейсе API для протоколирования предоставляются два класса: `FileHandler` и `SocketHandler`. Обработчик типа `SocketHandler` посылает протокольные записи на указанный хост и в порт. Наибольший интерес представляет обработчик типа `FileHandler`, выводящий протокольные записи в файл. Эти записи можно просто передать обработчику типа `FileHandler`, устанавливаемому по умолчанию:

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

Протокольные записи выводятся в файл `java.n.log`, который находится в начальном каталоге пользователя, где `n` — однозначный номер, отличающий этот файл от других аналогичных файлов. Если на платформе начальные каталоги пользователей не поддерживаются (например, в `Windows 95/98/ME`), файл записывается в каталог, предусмотренный по умолчанию, например `C:\Windows`. Протокольные записи хранятся в формате XML. Обычная протокольная запись имеет следующий вид:

```
<record>
  <date>2002-02-04T07:45:15</date>
  <millis>1012837515710</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
```

```

<level>INFO</level>
<class>com.mycompany.mylib.Reader</class>
<method>read</method>
<thread>10</thread>
<message>Reading file corejava.gif</message>
</record>

```

Поведение исходного обработчика протоколов типа `FileHandler` можно изменить, задав другие параметры настройки в диспетчере протоколирования (табл. 11.1) или воспользовавшись другим конструктором (см. краткое описание прикладного интерфейса API для протоколирования в конце этого раздела). Как правило, имя файла протокола, предлагаемое по умолчанию, не используется. Следовательно, для него нужно задать другой шаблон, например `%h/myapp.log` (переменные шаблона описаны в табл. 11.2).

Если в нескольких приложениях (или нескольких копиях одного и того же приложения) используется тот же самый файл протокола, следует установить признак `append`, определяющий режим ввода данных в конце файла. Кроме того, можно воспользоваться шаблоном имени файла `%u` таким образом, чтобы каждое приложение создавало свою особую копию файла протокола.

Целесообразно также установить режим ротации файлов протоколов. В этом случае имена файлов протоколов формируются следующим образом: `myapp.log.0`, `myapp.log.1`, `myapp.log.2` и т.д. Как только размер файла превысит допустимый предел, самый старый файл протокола удаляется, остальные переименовываются, а новый файл получает имя с номером 0.

Таблица 11.1. Параметры настройки обработчика протоколов типа **FileHandler**

Настраиваемое свойство	Описание	Значение по умолчанию
<code>java.util.logging. FileHandler.level</code>	Уровень обработчика	<code>Level.ALL</code>
<code>java.util.logging. FileHandler.append</code>	Определяет, должен ли обработчик добавлять записи в существующий файл или же открывать новый файл при очередном запуске программы	<code>false</code>
<code>java.util.logging. FileHandler.limit</code>	Приблизительная оценка максимального размера файла. При превышении этого размера открывается новый файл (0 — размер файла не ограничен)	0 (т.е. без ограничений) в классе <code>FileHandler</code> ; 50000 в исходной конфигурации диспетчера протоколирования
<code>java.util.logging. FileHandler.pattern</code>	Шаблон имени файла протокола (см. табл. 11.3)	<code>%h/java%u.log</code>
<code>java.util.logging. FileHandler.count</code>	Количество файлов протокола, участвующих в ротации	1 (ротация не производится)
<code>java.util.logging. FileHandler.filter</code>	Класс, используемый для фильтрации	Фильтрация отсутствует
<code>java.util.logging. FileHandler.encoding</code>	Применяемая кодировка	Кодировка, принятая на текущей платформе
<code>java.util.logging. FileHandler.formatter</code>	Средство форматирования протокольных записей	<code>java.util.logging. XMLFormatter</code>

Таблица 11.2. Переменные шаблона для имени файла протокола

Переменная	Описание
%h	Значение системного свойства user.home
%t	Временный системный каталог
%u	Однозначный номер, позволяющий избежать конфликта имен
%g	Определяет режим формирования номеров для подвергаемых ротации файлов протоколов. (Если ротация производится, а в шаблоне отсутствует переменная %g, то используется суффикс .%g.)
%%	Знак процента



СОВЕТ. Многие программисты пользуются средствами протоколирования для упрощения технической поддержки своих прикладных программ. Если в какой-то момент программа начинает работать некорректно, пользователь может обратиться к файлу протокола. Чтобы это стало возможным, следует установить признак append, организовать ротацию протоколов или сделать и то и другое.

Имеется также возможность определить свои собственные обработчики протоколов, расширив класс Handler или StreamHandler. Такой обработчик будет определен в программе, рассматриваемой далее в качестве примера. Этот обработчик отображает протокольные записи в окне (рис. 11.2).

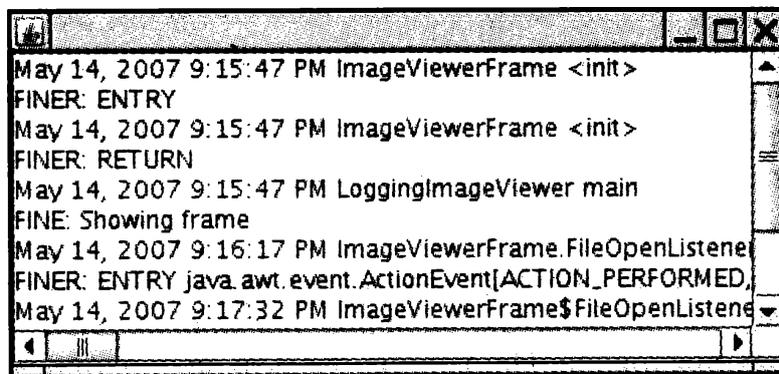


Рис. 11.2. Обработчик, отображающий протокольные записи в окне

Данный обработчик протоколов расширяет класс StreamHandler и устанавливает поток вывода с методами write() для отображения в текстовой области данных, выводимых в этот поток, как показано ниже.

```
class WindowHandler extends StreamHandler
{
    public WindowHandler()
    {
        . . . .
        final JTextArea output = new JTextArea();
        setOutputStream(new
            OutputStream()
            {
                public void write(int b) {} // не вызывается!
                public void write(byte[] b, int off, int len)
                {
```

```

        output.append(new String(b, off, len));
    }
    });
}
. . .
}

```

При таком подходе возникает лишь одно затруднение: обработчик размещает протокольные записи в буфере и направляет их в поток вывода только тогда, когда буфер заполнен. Следовательно, нужно переопределить метод `publish()`, чтобы выводить содержимое буфера после каждой протокольной записи. Это делается следующим образом:

```

class WindowHandler extends StreamHandler
{
    . . .
    public void publish(LogRecord record)
    {
        super.publish(record);
        flush();
    }
}

```

По желанию можно написать и более изощренные потоковые обработчики протоколов. Для этого достаточно расширить класс `Handler` и определить методы `publish()`, `flush()` и `close()`.

Фильтры

По умолчанию записи фильтруются в соответствии с уровнями протоколирования. У каждого регистратора и обработчика протоколов может быть свой фильтр, выполняющий дополнительную фильтрацию. Для этого достаточно реализовать интерфейс `Filter` и определить следующий метод:

```
boolean isLoggable(LogRecord record)
```

Для анализа протокольных записей можно выбрать любой критерий, а метод должен возвращать логическое значение `true` для тех протокольных записей, которые нужно ввести в файл протокола. Например, фильтр может пропускать только протокольные записи, созданные в начале выполнения метода и при возврате из него. Фильтр должен вызвать метод `record.getMessage()` и проверить, начинается ли запись со строки `ENTRY` или `RETURN`.

Для того чтобы задать фильтр в регистраторе или обработчике протоколов, следует вызвать метод `setFilter()`. Следует, однако, иметь в виду, что фильтры можно применять только по очереди.

Средства форматирования

Классы `ConsoleHandler` и `FileHandler` порождают протокольные записи в текстовом виде и в формате XML. Но для них можно определить свой собственный формат. Для этого необходимо расширить класс `Formatter` и переопределить следующий метод:

```
String format(LogRecord record)
```

Далее протокольная запись форматируется избранным способом и возвращается получаемая в итоге символьная строка. В методе `format()` может потребоваться

вызов приведенного ниже метода. Этот метод форматирует сообщение, составляющее часть протокольной записи, подставляя параметры и выполняя интернационализацию.

```
String formatMessage(LogRecord record)
```

Во многих форматах файлов (например, XML) предполагается существование начальной и конечной частей файла, заключающих в себе отформатированные протокольные записи. В таком случае нужно переопределить следующие методы:

```
String getHead(Handler h)
String getTail(Handler h)
```

И наконец, вызывается метод `setFormatter()`, устанавливающий средства форматирования в обработчике протоколов. Имея столько возможностей для протоколирования, в них легко запутаться. Поэтому ниже приведены краткие рекомендации по выполнению основных операций протоколирования.

“Рецепт” протоколирования

Ниже приведен “рецепт” по организации протоколирования, в котором сведены наиболее употребительные операции.

1. Для простых приложений выбирайте один регистратор. Желательно, чтобы имя регистратора совпадало с именем основного пакета приложения, например `com.mycompany.myprog`. Создать регистратор можно, сделав следующий вызов:

```
Logger logger = Logger.getLogger("com.mycompany.myprog");
```

Для удобства в те классы, где интенсивно используется протоколирование, можно добавить статические поля следующим образом:

```
private static final Logger logger = Logger.getLogger("com.mycompany.myprog");
```

2. По умолчанию все сообщения, имеющие уровень INFO и выше, выводятся на консоль. Пользователи могут изменить конфигурацию, предусмотренную по умолчанию, но, как пояснялось ранее, это довольно сложный процесс. Следовательно, лучше задать более оправданные настройки приложения по умолчанию. Приведенный ниже код гарантирует, что все сообщения будут зарегистрированы в файле протокола, связанном с конкретным приложением. Введите этот код в тело метода `main()` своего приложения.

```
if (System.getProperty("java.util.logging.config.class") == null
    && System.getProperty("java.util.logging.config.file") == null)
{
    try
    {
        Logger.getLogger("").setLevel(Level.ALL);
        final int LOG_ROTATION_COUNT = 10;
        Handler handler =
            new FileHandler("%h/myapp.log", 0, LOG_ROTATION_COUNT);
        Logger.getLogger("").addHandler(handler);
    }
    catch (IOException e)
    {
        logger.log(Level.SEVERE, "Can't create log file handler", e);
    }
}
```

3. Теперь все готово для протоколирования. Помните, что все сообщения, имеющие уровень INFO, WARNING и SEVERE, выводятся на консоль. Следовательно, эти уровни нужно зарезервировать для сообщений, представляющих ценность для пользователей вашей программы. Уровень FINE лучше выделить для сообщений, предназначенных для программистов. В тех местах кода, где вы обычно вызывали метод `System.out.println()`, регистрируйте сообщения следующим образом:

```
logger.fine("File open dialog canceled");
```

Рекомендуется также регистрировать неожиданные исключения, например, так, как показано ниже.

```
try
{
    . . .
}
catch (SomeException e)
{
    logger.log(Level.FINE, "explanation", e);
}
```

В листинге 11.2 приведен исходный код программы, составленной по описанному выше “рецепту” протоколирования, но с одним существенным дополнением: сообщения не только регистрируются, но и отображаются в протокольном окне.

Листинг 11.2. Исходный код из файла `logging/LoggingImageViewer.java`

```
1 package logging;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.util.logging.*;
7 import javax.swing.*;
8 /**
9  * Это видоизмененный вариант программы просмотра, в которой
10  * регистрируются различные события
11  * @version 1.02 2007-05-31
12  * @author Cay Horstmann
13  */
14 public class LoggingImageViewer
15 {
16     public static void main(String[] args)
17     {
18         if (System.getProperty("java.util.logging.config.class") == null
19             && System.getProperty("java.util.logging.config.file") == null)
20         {
21             try
22             {
23                 Logger.getLogger("com.horstmann.corejava").setLevel(Level.ALL);
24                 final int LOG_ROTATION_COUNT = 10;
25                 Handler handler = new FileHandler(
26                     "%h/LoggingImageViewer.log", 0, LOG_ROTATION_COUNT);
27                 Logger.getLogger("com.horstmann.corejava").addHandler(handler);
28             }
29             catch (IOException e)
30             {
```

```
31     {
32         Logger.getLogger("com.horstmann.corejava").log(Level.SEVERE,
33             "Can't create log file handler", e);
34     }
35 }
36 EventQueue.invokeLater(new Runnable()
37 {
38     public void run()
39     {
40         Handler windowHandler = new WindowHandler();
41         windowHandler.setLevel(Level.ALL);
42         Logger.getLogger("com.horstmann.corejava").
43             addHandler(windowHandler);
44
45         JFrame frame = new ImageViewerFrame();
46         frame.setTitle("LoggingImageViewer");
47         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
48
49         Logger.getLogger("com.horstmann.corejava").
50             fine("Showing frame");
51         frame.setVisible(true);
52     }
53 });
54 }
55 }
56
57 /**
58  * Фрейм, в котором показывается изображение
59  */
60 class ImageViewerFrame extends JFrame
61 {
62     private static final int DEFAULT_WIDTH = 300;
63     private static final int DEFAULT_HEIGHT = 400;
64
65     private JLabel label;
66     private static Logger logger =
67         Logger.getLogger("com.horstmann.corejava");
68
69     public ImageViewerFrame()
70     {
71         logger.entering("ImageViewerFrame", "<init>");
72         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
73
74         // установить строку меню
75         JMenuBar menuBar = new JMenuBar();
76         setJMenuBar(menuBar);
77
78         JMenu menu = new JMenu("File");
79         menuBar.add(menu);
80
81         JMenuItem openItem = new JMenuItem("Open");
82         menu.add(openItem);
83         openItem.addActionListener(new FileOpenListener());
84         JMenuItem exitItem = new JMenuItem("Exit");
85         menu.add(exitItem);
86         exitItem.addActionListener(new ActionListener()
87     {
```

```
88     public void actionPerformed(ActionEvent event)
89     {
90         logger.fine("Exiting.");
91         System.exit(0);
92     }
93     });
94
95
96     // использовать метку для обозначения изображений
97     label = new JLabel();
98     add(label);
99     logger.exiting("ImageViewerFrame", "<init>");
100 }
101
102 private class FileOpenListener implements ActionListener
103 {
104     public void actionPerformed(ActionEvent event)
105     {
106         logger.entering("ImageViewerFrame.FileOpenListener",
107             "actionPerformed", event);
108
109         // установить компонент для выбора файлов
110         JFileChooser chooser = new JFileChooser();
111         chooser.setCurrentDirectory(new File("."));
112
113         // принять все файлы с расширением .gif
114         chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
115             {
116             public boolean accept(File f)
117             {
118                 return f.getName().toLowerCase().endsWith(".gif") ||
119                     f.isDirectory();
120             }
121
122             public String getDescription()
123             {
124                 return "GIF Images";
125             }
126         });
127
128         // показать диалоговое окно для выбора файлов
129         int r = chooser.showOpenDialog(ImageViewerFrame.this);
130
131         // если файл изображения подходит, выбрать
132         // его в качестве пиктограммы для метки
133         if (r == JFileChooser.APPROVE_OPTION)
134         {
135             String name = chooser.getSelectedFile().getPath();
136             logger.log(Level.FINE, "Reading file {0}", name);
137             label.setIcon(new ImageIcon(name));
138         }
139         else logger.fine("File open dialog canceled.");
140         logger.exiting("ImageViewerFrame.FileOpenListener",
141             "actionPerformed");
142     }
143 }
144 }
145
```

```

146 /**
147  * Обработчик для отображения протокольных записей в окне
148  */
149 class WindowHandler extends StreamHandler
150 {
151     private JFrame frame;
152
153     public WindowHandler()
154     {
155         frame = new JFrame();
156         final JTextArea output = new JTextArea();
157         output.setEditable(false);
158         frame.setSize(200, 200);
159         frame.add(new JScrollPane(output));
160         frame.setFocusableWindowState(false);
161         frame.setVisible(true);
162         setOutputStream(new OutputStream()
163             {
164             public void write(int b)
165             {
166                 } // не вызывается!
167
168             public void write(byte[] b, int off, int len)
169             {
170                 output.append(new String(b, off, len));
171             }
172             });
173     }
174
175
176     public void publish(LogRecord record)
177     {
178         if (!frame.isVisible()) return;
179         super.publish(record);
180         flush();
181     }
182 }

```

java.util.logging.Logger 1.4

- `Logger getLogger(String loggerName)`
- `Logger getLogger(String loggerName, String bundleName)`

Возвращают регистратор протоколов с указанным именем. Если регистратор не существует, он создается.

Параметры:

<i>loggerName</i>	Иерархическое имя регистратора, например <code>com.mycompany.myapp</code> .
<i>bundleName</i>	Имя комплекта ресурсов для поиска интернационализированных сообщений

- `void severe(String message)`
- `void warning(String message)`

- `void info(String message)`
- `void config(String message)`
- `void fine(String message)`
- `void finer(String message)`
- `void finest(String message)`

Протоколируют запись на уровне, соответствующем имени метода, вместе с заданным сообщением.

- `void entering(String className, String methodName)`
- `void entering(String className, String methodName, Object param)`
- `void entering(String className, String methodName, Object[] param)`
- `void exiting(String className, String methodName)`
- `void exiting(String className, String methodName, Object result)`

Протоколируют запись, описывающую вход и выход из метода с заданными параметрами или возвращаемым значением.

- `void throwing(String className, String methodName, Throwable t)`
- Протоколирует запись, описывающую процесс генерирования объекта исключения.

- `void log(Level level, String message)`
- `void log(Level level, String message, Object obj)`
- `void log(Level level, String message, Object[] objs)`
- `void log(Level level, String message, Throwable t)`

Протоколируют запись на заданном уровне вместе с указанным сообщением. Запись может содержать обычные объекты или объект исключения. Для этой цели в сообщении предусмотрены заполнители {0}, {1} и т.д.

- `void logp(Level level, String className, String methodName, String message)`
- `void logp(Level level, String className, String methodName, String message, Object obj)`
- `void logp(Level level, String className, String methodName, String message, Object[] objs)`
- `void logp(Level level, String className, String methodName, String message, Throwable t)`

Протоколируют запись на заданном уровне с данными о вызове и указанным сообщением. Запись может включать обычные объекты или объекты исключений.

- `void logrb(Level level, String className, String methodName, String bundleName, String message)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object obj)`

- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object[] objs)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Throwable t)`

Протоколируют запись на заданном уровне с данными о вызове, указанным сообщением и именем комплекта ресурсов. Запись может включать обычные объекты или объекты исключений.

- `Level getLevel()`
- `void setLevel(Level l)`
Получают и устанавливают уровень данного регистратора.
- `Logger getParent()`
- `void setParent(Logger l)`
Получают и устанавливают родительский регистратор.
- `Handler[] getHandlers()`
Получает все обработчики протоколов для данного регистратора.
- `void addHandler(Handler h)`
- `void removeHandler(Handler h)`
Добавляют и удаляют обработчик протоколов для данного регистратора.
- `boolean getUseParentHandlers()`
- `void setUseParentHandlers(boolean b)`
Получает и устанавливает свойство, определяющее режим использования родительского обработчика протоколов. Если это свойство принимает логическое значение `true`, регистратор направляет все протоколируемые записи обработчикам своего родительского регистратора.
- `Filter getFilter()`
- `void setFilter(Filter f)`
Получают и устанавливают фильтр для данного регистратора.

java.util.logging.Handler 1.4

- `abstract void publish(LogRecord record)`
Посылает протокольную запись по указанному адресу.
- `abstract void flush()`
Выводит данные из буфера.
- `abstract void close()`
Выводит данные из буфера и освобождает все ресурсы.
- `Filter getFilter()`
- `void setFilter(Filter f)`
Получает и устанавливает фильтр для данного обработчика протоколов.
- `Formatter getFormatter()`

- `void setFormatter(Formatter f)`
Получают и устанавливают средство форматирования для данного обработчика протоколов.
- `Level getLevel()`
- `void setLevel(Level l)`
Получают и устанавливают уровень данного обработчика протоколов.

java.util.logging.ConsoleHandler 1.4

- `ConsoleHandler()`
Создает новый консольный обработчик протоколов.

java.util.logging.FileHandler 1.4

- `FileHandler(String pattern)`
- `FileHandler(String pattern, boolean append)`
- `FileHandler(String pattern, int limit, int count)`
- `FileHandler(String pattern, int limit, int count, boolean append)`
Создают файловый обработчик протоколов.

<i>Параметры:</i>	<i>pattern</i>	Шаблон для создания имени файла протокола (см. табл. 11.2)
	<i>limit</i>	Приблизительная оценка максимального размера файла протокола. При превышении этого размера открывается новый файл
	<i>count</i>	Количество файлов, участвующих в ротации
	<i>append</i>	Принимает логическое значение <code>true</code> , если вновь созданный объект файлового обработчика должен выводить данные в существующий файл протокола

java.util.logging.LogRecord 1.4

- `Level getLevel()`
Получает уровень протоколирования для данной записи.
- `String getLoggerName()`
Получает имя регистратора, создавшего данную запись.
- `ResourceBundle getResourceBundle()`
- `String getResourceBundleName()`
Получает комплект ресурсов, предназначенный для интернационализации сообщения, или его имя. Если комплект ресурсов не предусмотрен, возвращается пустое значение `null`.

- `String getMessage()`
Получает исходное сообщение, не прошедшее интернационализацию и форматирование.
- `Object[] getParameters()`
Получает параметры или пустое значение `null`, если они отсутствуют.
- `Throwable getThrown()`
Получает объект сгенерированного исключения или пустое значение `null`, если объект не существует.
- `String getSourceClassName()`
- `String getSourceMethodName()`
Определяют местоположение кода, зарегистрировавшего данную запись. Эти сведения могут быть предоставлены кодом регистрации или автоматически выведены, исходя из анализа стека выполнения программы. Если код регистрации содержит неверное значение или программа была оптимизирована таким образом, что определить местоположение кода невозможно, эти сведения могут оказаться неточными.
- `long getMillis()`
Определяет время создания данной протокольной записи в миллисекундах, отсчитывая его от 1 января 1970 г.
- `long getSequenceNumber()`
Получает однозначный порядковый номер записи.
- `int getThreadID()`
Получает однозначный идентификатор потока, в котором была создана данная протокольная запись. Такие идентификаторы присваиваются классом `LogRecord` и не имеют отношения к другим идентификаторам потоков.

java.util.logging.Filter 1.4

- `boolean isLoggable(LogRecord record)`
Возвращает логическое значение `true`, если указанная запись должна быть зарегистрирована.

java.util.logging.Formatter 1.4

- `abstract String format(LogRecord record)`
Возвращает символьную строку, полученную в результате форматирования заданной протокольной записи.
- `String getHead(Handler h)`
- `String getTail(Handler h)`
Возвращают символьные строки, которые должны появиться в начале и в конце документа, содержащего данную протокольную запись. Эти методы определены в суперклассе `Formatter` и по умолчанию возвращают пустую символьную строку. При необходимости их следует переопределить.

- `String formatMessage(LogRecord record)`

Возвращает интернационализированное и форматированное сообщение, являющееся частью протокольной записи.

Рекомендации по отладке программ

Допустим, вы написали программу и оснастили ее средствами для перехвата и соответствующей обработки исключений. Затем вы запускаете ее на выполнение, и она работает неправильно. Что же делать дальше? (Если у вас никогда не возникал такой вопрос, можете не читать оставшуюся часть этой главы.)

Разумеется, лучше всего было бы иметь под рукой удобный и эффективный отладчик. Такие отладчики являются частью профессиональных ИСР, например Eclipse или NetBeans. Мы еще вернемся далее в этой главе к отладчикам, а в этом разделе дадим несколько рекомендаций, которые стоит взять на вооружение, прежде чем запускать отладчик.

1. Значение любой переменной можно вывести с помощью выражения

```
System.out.println("x=" + x);
```

или

```
Logger.getGlobal().info("x=" + x);
```

Если в переменной `x` содержится числовое значение, оно преобразуется в символьную строку. Если же переменная `x` ссылается на объект, то вызывается метод `toString()`. Состояние объекта, используемого в качестве неявного параметра, поможет определить следующий вызов:

```
Logger.getGlobal().info("this=" + this);
```

В большинстве классов Java метод `toString()` переопределяется таким образом, чтобы предоставить пользователю как можно больше полезной информации. Это очень удобно для отладки. Об этом следует позаботиться и в своих классах.

2. Один малоизвестный, но очень полезный прием состоит в том, чтобы включить в каждый класс отдельный метод `main()` и разместить в нем код, позволяющий протестировать этот класс отдельно от других, как показано ниже.

```
public class MyClass
{
    методы и поля данного класса
    . . .
    public static void main(String[] args)
    {
        тестовый код
    }
}
```

Создайте несколько объектов, вызовите все методы и проверьте, правильно ли они работают. Все это делается в теле методов `main()`, а интерпретатор вызывается для каждого файла класса по отдельности. Если выполняется апплет, то ни один из этих методов `main()` вообще не будет вызван. А если выполняется приложение, то интерпретатор вызовет только метод `main()` из запускающего класса.

3. Если предыдущая рекомендация пришлась вам по душе, попробуйте поработать в среде JUnit, которая доступна по адресу <http://junit.org>. JUnit представляет собой широко распространенную среду блочного тестирования, которая существенно упрощает работу по созданию наборов контрольных примеров. Запускайте тесты после каждого видоизменения класса. При обнаружении программных ошибок добавляйте новый контрольный пример для последующего тестирования.
4. *Протоколирующий прокси-объект* представляет собой экземпляр подкласса, который перехватывает вызовы методов, протоколирует их и обращается к суперклассу. Так, если у вас возникли трудности при вызове метода `nextDouble()` из класса `Random`, вы можете создать прокси-объект в виде экземпляра анонимного подкласса следующим образом:

```
Random generator = new
Random()
{
    public double nextDouble()
    {
        double result = super.nextDouble();
        Logger.getGlobal().info("nextDouble: " + result);
        return result;
    }
};
```

При каждом вызове метода `nextDouble()` будет формироваться протокольное сообщение. Ниже поясняется, как выявить ту часть кода, из которой вызывается данный метод, и как произвести трассировку стека.

5. Вызвав метод `printStackTrace()` из класса `Throwable`, можно получить трассировку стека из любого объекта исключения. В приведенном ниже фрагменте кода перехватывается любое исключение, выводится объект исключения и трассировка стека, а затем повторно генерируется исключение, чтобы найти предназначенный для него обработчик.

```
try
{
    . . .
}
catch (Throwable t)
{
    t.printStackTrace();
    throw t;
}
```

Для трассировки стека не нужно даже перехватывать исключение. Просто введите следующую строку в любом месте кода:

```
Thread.dumpStack();
```

6. Как правило, результаты трассировки стека выводятся в поток сообщений об ошибках `System.err`. Если же требуется протолировать или отобразить результаты трассировки стека, то ниже показано, как заключить их в символьную строку.

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
new Throwable().printStackTrace(out);
String description = out.toString();
```

7. Ошибки, возникающие при выполнении программы, удобно записывать в файл. Но обычно ошибки посылаются в поток сообщений об ошибках `System.err`, а не в поток вывода `System.out`. Поэтому ошибки нельзя вывести в файл по обычной команде следующим образом:

```
java MyProgram > errors.txt
```

Ошибки лучше перехватывать по такой команде:

```
java MyProgram 2> errors.txt
```

А для того чтобы направить оба потока, `Stream.err` и `Stream.out`, в один и тот же файл, воспользуйтесь командой

```
java MyProgram >& errors.txt
```

Эта команда действует в командной оболочке `bash` и в `Windows`.

8. Результаты трассировки стека необрабатываемых исключений свидетельствуют о том, что поток сообщений об ошибках `System.err` далек от идеала. Выводимые в него сообщения смущают конечных пользователей, если им случается их увидеть, и они не доступны для диагностических целей, когда в этом возникает потребность. Поэтому более правильный подход состоит в том, чтобы протолировать их в файл. Обработчик для необрабатываемых исключений можно заменить статическим методом `Thread.setDefaultUncaughtExceptionHandler()` следующим образом:

```
Thread.setDefaultUncaughtExceptionHandler(
    new Thread.UncaughtExceptionHandler()
    {
        public void uncaughtException(Thread t, Throwable e)
        {
            сохранить данные в файле протокола
        }
    });
```

9. Для того чтобы отследить загрузку класса, запустите виртуальную машину `Java` с параметром `-verbose`. На экране появятся строки, подобные следующим:

```
[Opened /usr/local/jdk5.0/jre/lib/rt.jar]
[Opened /usr/local/jdk5.0/jre/lib/jsse.jar]
[Opened /usr/local/jdk5.0/jre/lib/jce.jar]
[Opened /usr/local/jdk5.0/jre/lib/charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
```

Этот способ оказывается полезным для диагностики ошибок, связанных с путями к классам.

10. Параметр `-Xlint` указывает компилятору выявлять типичные ошибки в исходном коде программы. Если, например, вызвать компилятор так, как показано ниже, он уведомит о пропущенных операторах `break` в ветвях оператора `switch`.

```
javac -Xlint:fallthrough
```

Термин *lint* применялся ранее для описания инструмента, предназначенного для обнаружения потенциальных ошибок в программах, написанных на С. Теперь он обозначает все инструменты, отмечающие языковые конструкции, которые могут быть некорректными. Ниже описаны допустимые значения параметра **-Xlint**.

<code>-Xlint</code> или <code>-Xlint:all</code>	Выполняет все проверки
<code>-Xlint:deprecation</code>	То же, что и <code>-deprecation</code> , т.е. выполняется проверка на наличие методов, не рекомендованных к применению
<code>-Xlint:fallthrough</code>	Проверяет наличие операторов <code>break</code> в ветвях оператора <code>switch</code>
<code>-Xlint:none</code>	Проверки не выполняются
<code>-Xlint:path</code>	Проверяет, существуют ли каталоги, указанные в пути к классам и исходным файлам
<code>-Xlint:serial</code>	Предупреждает о сериализуемых классах без идентификатора <code>serialVersionUID</code> (см. главу 1 во втором томе данной книги)
<code>-Xlint:unchecked</code>	Предупреждает о сомнительных взаимных преобразованиях базовых и обобщенных типов (подробнее об этом — в главе 12)

11. В виртуальной машине Java реализована поддержка *контроля и управления* приложениями, написанными на Java. Это позволяет установить в ней агенты, которые будут отслеживать расходование памяти, использование потоков, загрузку классов и т.п. Такая поддержка важна для работы с крупномасштабными прикладными программами, работающими в течение длительного времени, например, с серверами приложений. Для демонстрации новых возможностей в комплект поставки JDK включена графическая утилита `jconsole`, которая отображает статистические данные о производительности виртуальной машины (рис. 11.3). Для того чтобы активизировать контроль и управление приложениями на Java, следует сначала выяснить идентификатор процесса операционной системы, выделяемого для работы виртуальной машины. В Unix/Linux для этой цели нужно воспользоваться утилитой `ps`, а в Windows — диспетчером задач. Затем можно запустить утилиту `jconsole` следующим образом:

```
jconsole идентификатор_процесса
```

На консоль будет выведено немало полезных сведений о запущенной на выполнение программе. Подробнее об этом можно узнать по адресу www.oracle.com/technetwork/articles/java/jconsole-1564139.html.

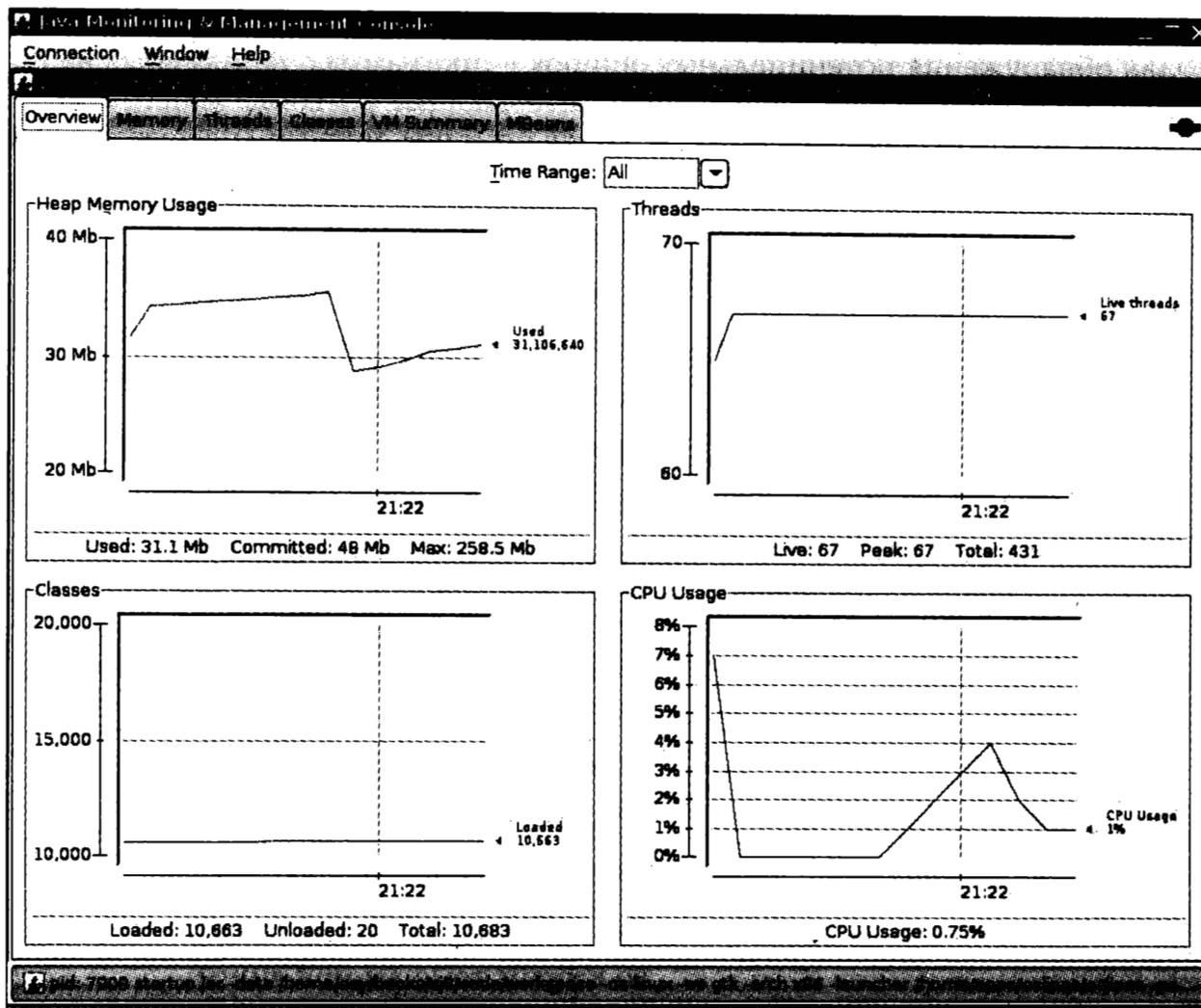


Рис. 11.3. Окно утилиты jconsole

12. Для вывода содержимого памяти, выделенной под “кучу” (так называемого дампа), можно воспользоваться утилитой jmap. Для этого введите следующие команды:

```
jmap -dump:format=b,file=имя_файла_дампа идентификатор_процесса
jhat имя_файла_дампа
```

Затем введите в окне своего браузера адрес **localhost:7000**. Там вы найдете веб-приложение, которое поможет вам углубиться в анализ содержимого “кучи” на момент его вывода из памяти.

13. Если запустить виртуальную машину Java с параметром **-Xprof**, она, в свою очередь, запустит устаревший профилировщик, определяющий наиболее часто используемые методы. Этот профилировщик направляет данные в поток вывода System.out. Туда же направляются данные о методах, которые были скомпилированы динамическим компилятором.



ВНИМАНИЕ! Параметры компилятора, начинающиеся с **-X**, официально не поддерживаются и в некоторых версиях JDK могут отсутствовать. Получить список нестандартных параметров можно по команде **java -X**.

Рекомендации по отладке программ с ГПИ

В этом разделе сначала даются дополнительные рекомендации по отладке программ с ГПИ. А затем будет показано, как пользоваться роботом AWT для автоматизации процесса тестирования ГПИ.

1. Любуясь окнами, созданными средствами библиотеки Swing, и удивляясь, как красиво и ровно расположены в них компоненты, вы, естественно, заинтересуетесь, как именно они устроены. Для этого достаточно нажать комбинацию клавиш <Ctrl+Shift+F1>, и все компоненты будут выведены по порядку, как показано ниже.

```
FontDialog[frame0,0,0,300x200,layout=java.awt.BorderLayout,...
javax.swing.JRootPane[,4,23,292x173,layout=javax.swing.JRootPane$RootLayout;...
javax.swing.JPanel[null.glassPane,0,0,292x173,hidden,layout=java.awt.FlowLayout,...
javax.swing.JLayeredPane[null.layeredPane,0,0,292x173,...
javax.swing.JPanel[null.contentPane,0,0,292x173,layout=java.awt.GridBagLayout,...
javax.swing.JList[,0,0,73x152,alignmentX=null,alignmentY=null,...
    javax.swing.CellRendererPane[,0,0,0x0,hidden]
        javax.swing.DefaultListCellRenderer$UIResource[,-73,-19,0x0,...
            javax.swing.JCheckBox[,157,13,50x25,layout=javax.swing.OverlayLayout,...
            javax.swing.JCheckBox[,156,65,52x25,layout=javax.swing.OverlayLayout,...
            javax.swing.JLabel[,114,119,30x17,alignmentX=0.0,alignmentY=null,...
            javax.swing.JTextField[,186,117,105x21,alignmentX=null,alignmentY=null,...
            javax.swing.JTextField[,0,152,291x21,alignmentX=null,alignmentY=nul
```

2. Если разработанный вами собственный компонент пользовательского интерфейса Swing на экране отображается неверно, вы по достоинству оцените удобство *графического отладчика* Swing. Даже если вы никогда не создавали свои собственные компоненты, очень поучительно проследить, как именно компоненты отображаются на экране. Чтобы переключиться на отладку компонента библиотеки Swing, следует вызвать метод `setDebugGraphicsOptions()` из класса `JComponent`. Ниже вкратце описаны параметры этого метода.

<code>DebugGraphics.FLASH_OPTION</code>	Выделяет красным линией, прямоугольник или текст, прежде чем отобразить их
<code>DebugGraphics.LOG_OPTION</code>	Выводит на экран сообщение о каждой графической операции
<code>DebugGraphics.BUFFERED_OPTION</code>	Выводит операции, которые выполняются в неотображаемом буфере
<code>DebugGraphics.NONE_OPTION</code>	Отключает отладку графики

Для того чтобы выделение красным действовало правильно, нужно отключить режим двойной буферизации, которая служит для подавления мерцания экрана при обновлении окна. Это можно сделать с помощью приведенного ниже фрагмента кода.

```
RepaintManager.currentManager(getRootPane())
    .setDoubleBufferingEnabled(false);
((JComponent) getContentPane()).setDebugGraphicsOptions(
    DebugGraphics.FLASH_OPTION);
```

Введите эти строки кода в конце конструктора создаваемого вами фрейма. При выполнении программы вы увидите панель, которая медленно заполняется содержимым. Для локальной отладки каждого компонента воспользуйтесь методом `setDebugGraphicsOptions()`. Продолжительность, количество и цвет выделения можно регулировать (подробнее об этом — в описании класса `DebugGraphics` непосредственно в документации).

3. Если требуется получить запись о каждом событии AWT, инициируемом в ГПИ вашей прикладной программы, установите в каждом компоненте приемник этих событий. Эту процедуру нетрудно автоматизировать с помощью механизма рефлексии. В качестве примера в листинге 11.3 приведен исходный код класса `EventTracer`, отслеживающего события.

Для отслеживания событий в проверяемом компоненте введите в него объект класса `EventTracer` следующим образом:

```
EventTracer tracer = new EventTracer();
tracer.add(frame);
```

Это даст возможность получить текстовое описание всех событий в ГПИ вашей прикладной программы, как показано на рис. 11.4.

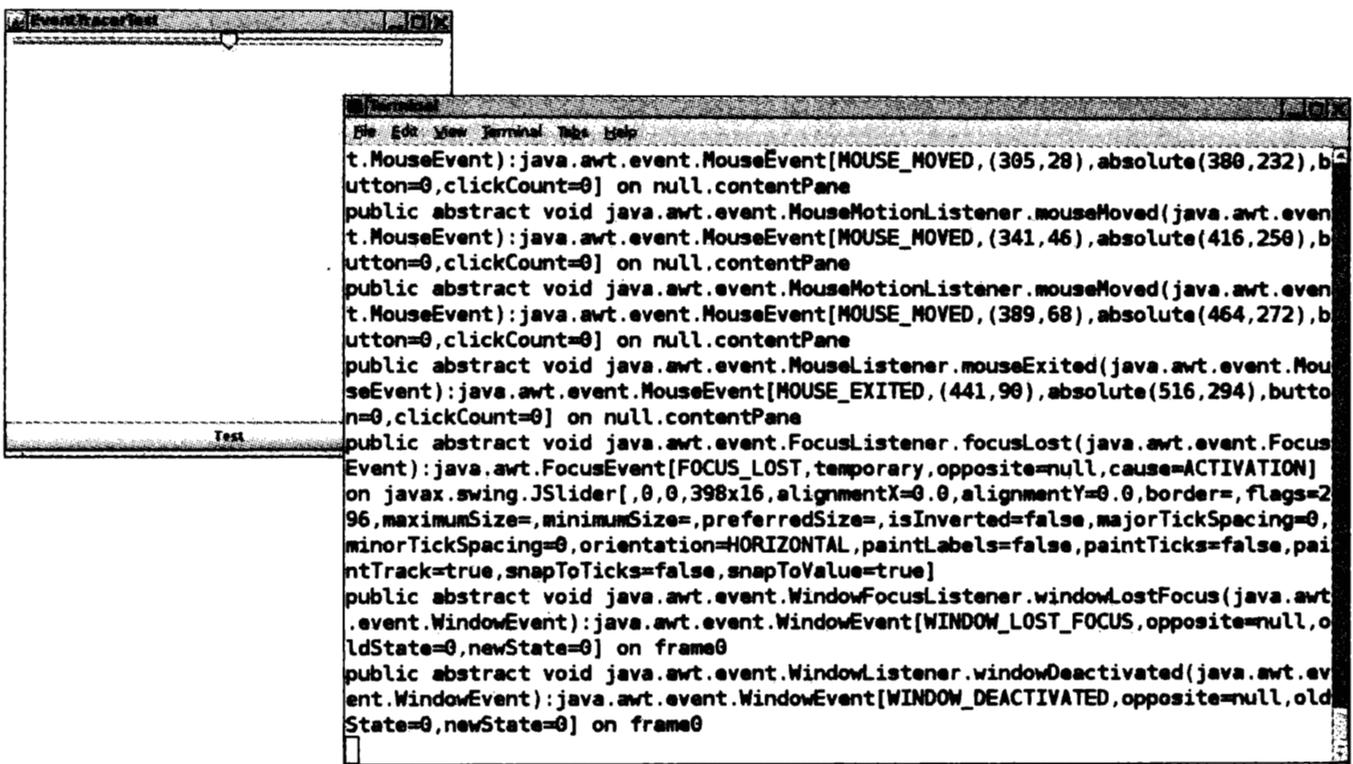


Рис. 11.4. Результат работы класса `EventTracer`

Листинг 11.3. Исходный код из файла `eventTracer/EventTracer.java`

```
1 package eventTracer;
2
3 import java.awt.*;
4 import java.beans.*;
5 import java.lang.reflect.*;
6
7 /**
```

```
8  * @version 1.31 2004-05-10
9  * @author Cay Horstmann
10 */
11 public class EventTracer
12 {
13     private InvocationHandler handler;
14
15     public EventTracer()
16     {
17         // обработчик всех событий в виде прокси-объектов
18         handler = new InvocationHandler()
19         {
20             public Object invoke(Object proxy, Method method, Object[] args)
21             {
22                 System.out.println(method + ":" + args[0]);
23                 return null;
24             }
25         };
26     }
27
28     /**
29     * Добавляет объекты для отслеживания всех событий, которые может
30     * принимать данный компонент и производные от него компоненты
31     * @param c Компонент
32     */
33     public void add(Component c)
34     {
35         try
36         {
37             // получить все события, которые может принимать данный компонент
38             BeanInfo info = Introspector.getBeanInfo(c.getClass());
39
40             EventSetDescriptor[] eventSets = info.getEventSetDescriptors();
41             for (EventSetDescriptor eventSet : eventSets)
42                 addListener(c, eventSet);
43         }
44         catch (IntrospectionException e)
45         {
46         }
47         // если генерируется исключение, приемники событий не вводятся
48
49         if (c instanceof Container)
50         {
51             // получить все производные объекты и организовать
52             // рекурсивный вызов метода add()
53             for (Component comp : ((Container) c).getComponents())
54                 add(comp);
55         }
56     }
57     /**
58     * Добавляет приемник заданного множества событий
59     * @param c Компонент
60     * @param eventSet Описатель интерфейса приемника событий
61     */
62     public void addListener(Component c, EventSetDescriptor eventSet)
63     {
64         // создать прокси-объект для данного типа приемника событий и
65         // направить все вызовы этому обработчику
```

```
66     Object proxy = Proxy.newProxyInstance(null, new Class[]
67         { eventSet.getListenerType() }, handler);
68
69     // добавить прокси-объект как приемник событий в компонент
70     Method addListenerMethod = eventSet.getAddListenerMethod();
71     try
72     {
73         addListenerMethod.invoke(c, proxy);
74     }
75     catch (ReflectiveOperationException e)
76     {
77     }
78     // если генерируется исключение, приемник событий не вводится
79 }
80 }
```

Применение робота AWT

Класс `Robot` можно использовать для передачи нажатий клавиш и щелчков кнопками мыши в любую AWT-программу. Этот класс служит для автоматической проверки ГПИ. Для того чтобы создать объект типа `Robot`, нужно сначала получить объект типа `GraphicsDevice`. Выбираемое по умолчанию устройство вывода на экран определяется с помощью следующей последовательности вызовов:

```
GraphicsEnvironment environment =
    GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice screen = environment.getDefaultScreenDevice();
```

Затем конструируется объект типа `Robot`:

```
Robot robot = new Robot(screen);
```

Чтобы передать нажатия клавиш программе, объект типа `Robot` должен симулировать нажатие и отпускание клавиш. Ниже показано, каким образом он на это настраивается.

```
robot.keyPress(KeyEvent.VK_TAB);
robot.keyRelease(KeyEvent.VK_TAB);
```

Для имитации щелчка кнопкой мыши сначала нужно переместить ее курсор, а затем нажать и отпустить кнопку. Эти действия воспроизводятся в следующем фрагменте кода:

```
robot.mouseMove(x, y); // абсолютные координаты положения курсора в пикселях
robot.mousePress(InputEvent.BUTTON1_MASK);
robot.mouseRelease(InputEvent.BUTTON1_MASK);
```

Основной замысел такой проверки состоит в том, что, имитируя нажатия клавиш и щелчки кнопками мыши, можно увидеть, правильно ли прикладная программа реагирует на них. Для того чтобы захватить экран, вызывается метод `createScreenCapture()`, как показано ниже. Координаты, задаваемые в конструкторе класса `Rectangle`, также являются абсолютными координатами положения на экране и выражаются в пикселях.

```
Rectangle rect = new Rectangle(x, y, width, height);
BufferedImage image = robot.createScreenCapture(rect);
```

И наконец, чтобы в прикладной программе нормально перехватывались команды, выполняемые объектами типа `Robot`, между ними должны быть небольшие задержки. Для этой цели служит метод `delay()`, при вызове которого величина задержки указывается в миллисекундах следующим образом:

```
robot.delay(1000); // задержка на 1000 миллисекунд
```

В примере программы, приведенной в листинге 11.4, демонстрируется применение объектов типа `Robot`. Эти объекты тестируют программу, демонстрирующую обращение с кнопками на панели и рассматривавшуюся в главе 8. Сначала в ответ на нажатие клавиши пробела активизируется левая кнопка. Затем объект типа `Robot` ожидает две секунды, чтобы дать пользователю возможность увидеть состояние экрана. По истечении этого промежутка времени объект типа `Robot` имитирует нажатие клавиши `<Tab>` и клавиши пробела для активизации следующей кнопки. После этого имитируется щелчок на третьей кнопке. (Для того чтобы действительно щелкнуть на этой кнопке, в программе нужно правильно задать параметры `x` и `y`.) И наконец, экран захватывается и отображается в отдельном фрейме (рис. 11.5).

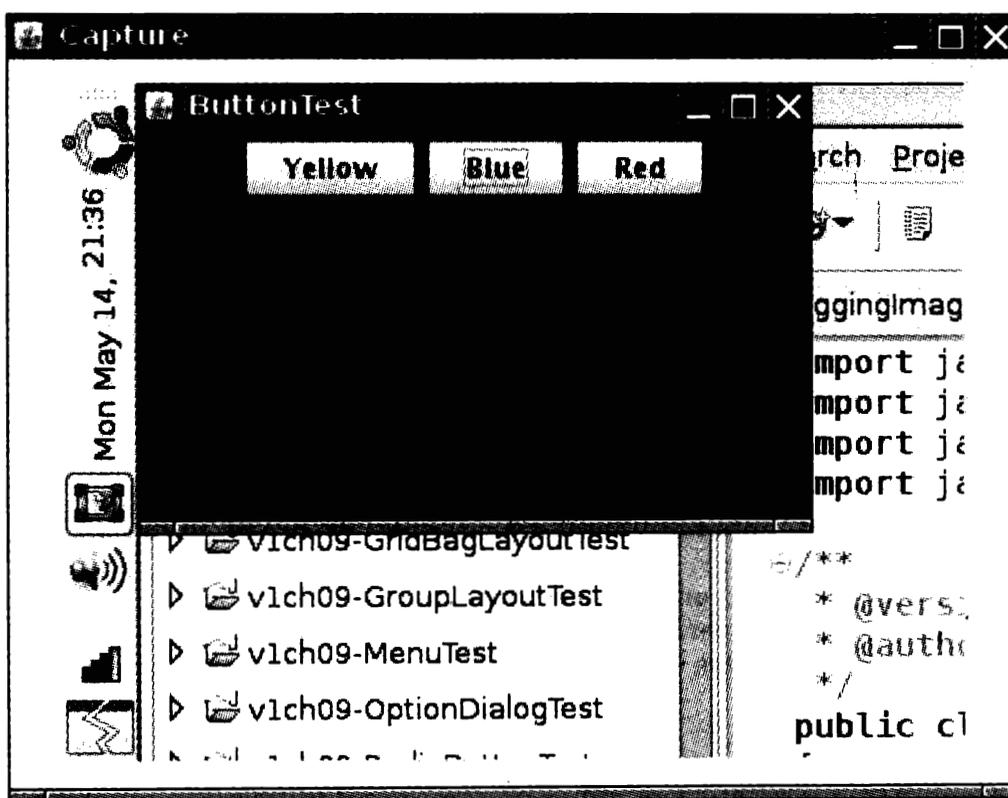


Рис. 11.5. Захват экрана роботом AWT



НА ЗАМЕТКУ! Робот AWT следует непременно выполнять в отдельном потоке, как показано в примере кода из листинга 11.4. Подробнее о потоках речь пойдет в главе 14.

Рассматриваемый здесь пример показывает, что сам класс `Robot` не совсем удобен для тестирования ГПИ. Он скорее предоставляет основные стандартные блоки, из которых можно построить средство для тестирования программ. Профессиональные средства тестирования способны фиксировать, сохранять и воспроизводить

сценарии взаимодействия тестируемых прикладных программ с пользователями, а также определять расположение компонентов на экране, чтобы не щелкать кнопками мыши наугад.

Листинг 11.4. Исходный код из файла `robot/RobotTest.java`

```
1 package robot;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.image.*;
6 import javax.swing.*;
7 /**
8  * @version 1.04 2012-05-17
9  * @author Cay Horstmann
10 */
11 public class RobotTest
12 {
13     public static void main(String[] args)
14     {
15         EventQueue.invokeLater(new Runnable()
16         {
17             public void run()
18             {
19                 // создать фрейм с панелью кнопок
20
21                 ButtonFrame frame = new ButtonFrame();
22                 frame.setTitle("ButtonTest");
23                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24                 frame.setVisible(true);
25             }
26         });
27
28         // присоединить робот к устройству вывода на экран
29
30         GraphicsEnvironment environment =
31             GraphicsEnvironment.getLocalGraphicsEnvironment();
32         GraphicsDevice screen = environment.getDefaultScreenDevice();
33
34         try
35         {
36             final Robot robot = new Rrobot(screen);
37             robot.waitForIdle();
38             new Thread()
39             {
40                 public void run()
41                 {
42                     runTest(robot);
43                 };
44             }.start();
45         }
46         catch (AWTException e)
47         {
48             e.printStackTrace();
49         }
```

```
50     }
51
52     /**
53     * Выполняет простую тестовую процедуру
54     * @param robot Робот, присоединяемый к устройству вывода на экран
55     */
56     public static void runTest(Robot robot)
57     {
58         // симитировать нажатие клавиши пробела
59         robot.keyPress(' ');
60         robot.keyRelease(' ');
61
62         // симитировать нажатие клавиш табуляции и пробела
63         robot.delay(2000);
64         robot.keyPress(KeyEvent.VK_TAB);
65         robot.keyRelease(KeyEvent.VK_TAB);
66         robot.keyPress(' ');
67         robot.keyRelease(' ');
68
69         // симитировать щелчок кнопкой мыши на крайней справа кнопке
70         robot.delay(2000);
71         robot.mouseMove(220, 40);
72         robot.mousePress(InputEvent.BUTTON1_MASK);
73         robot.mouseRelease(InputEvent.BUTTON1_MASK);
74
75         // захватить экран и показать полученное в итоге изображение
76         robot.delay(2000);
77         BufferedImage image =
78             robot.createScreenCapture(new Rectangle(0, 0, 400, 300));
79
80         ImageFrame frame = new ImageFrame(image);
81         frame.setVisible(true);
82     }
83 }
84
85 /**
86 * Фрейм для показа захваченного изображения экрана
87 */
88
89 class ImageFrame extends JFrame
90 {
91     private static final int DEFAULT_WIDTH = 450;
92     private static final int DEFAULT_HEIGHT = 350;
93
94     /**
95     * @param image Показываемое изображение
96     */
97     public ImageFrame(Image image)
98     {
99         setTitle("Capture");
100        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
101
102        JLabel label = new JLabel(new ImageIcon(image));
103        add(label);
104    }
105 }
```

java.awt.GraphicsEnvironment 1.2

- `static GraphicsEnvironment getLocalGraphicsEnvironment()`
Возвращает локальное графическое окружение.
- `GraphicsDevice getDefaultScreenDevice()`
Возвращает устройство вывода на экран, заданное по умолчанию. Однако компьютеры с несколькими мониторами имеют по одному устройству вывода на каждый экран, поэтому для получения массива, содержащего все устройства, нужно вызвать метод `getScreenDevices()`.

java.awt.Robot 1.3

- `Robot(GraphicsDevice device)`
Создает объект типа `Robot`, способный взаимодействовать с данным устройством вывода.
- `void keyPress(int key)`
- `void keyRelease(int key)`
Имитируют нажатие и отпускание клавиши.
Параметры: `key` Код клавиши. Подробные сведения о кодах клавиш можно найти в документации на класс `KeyStroke`
- `void mouseMove(int x, int y)`
Имитирует перемещение курсора мыши.
Параметры: `x, y` Абсолютные координаты положения курсора мыши, выраженные в пикселях
- `void mousePress(int eventMask)`
- `void mouseRelease(int eventMask)`
Имитируют нажатие и отпускание кнопки мыши.
Параметры: `eventMask` Маска события, описывающая кнопку мыши. Дополнительные сведения о масках событий см. в документации на класс `InputEvent`
- `void delay(int milliseconds)`
Приостанавливает работу объекта типа `Robot` на заданное количество миллисекунд.
- `BufferedImage createScreenCapture(Rectangle rect)`
Захватывает часть экрана.
Параметры: `rect` Захватываемая часть экрана. Указывается в абсолютных координатах

Применение отладчика

Отладка программы путем вывода на экран промежуточных результатов — не самое приятное занятие. Ведь в этом случае приходится постоянно добавлять и удалять операторы в разных местах исходного кода программы, а затем компилировать ее заново. Отладчик намного облегчает эту задачу, поскольку в нем можно установить *точку прерывания*. Вплоть до этой точки программа выполняется без задержек, а после того, как выполнение программы приостанавливается, можно получить интересные сведения о состоянии исполняемого кода.

В листинге 11.7 приведен исходный код программы `ButtonTest` из главы 8, куда были намеренно внесены некоторые ошибки. Если щелкнуть на любой кнопке, то ничего не произойдет. Проанализировав исходный код, вы обнаружите, что после щелчка на кнопке фон панели должен окраситься в соответствующий цвет.

Для обнаружения ошибок в такой короткой программе достаточно внимательно просмотреть ее исходный код. Но ради примера в данном случае предполагается, что такой способ обнаружения программных ошибок непрактичен. Далее в этой главе будет показано, как пользоваться отладчиком ИСР Eclipse для эффективного обнаружения ошибок в крупных программах.



НА ЗАМЕТКУ! Если вы пользуетесь таким автономным отладчиком, как, например, JSwat (<http://code.google.com/p/jswat>) или старым и неуклюжим отладчиком вроде `jdb`, то должны сначала скомпилировать рассматриваемую здесь программу с параметром `-g`, введя следующую команду:

```
javac -g BuggyButtonTest.java
```

А в ИСР это делается автоматически.

Листинг 11.5. Исходный код из файла `debugger/BuggyButtonTest.java`

```
1 package debugger;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6 /**
7  * @version 1.22 2007-05-14
8  * @author Cay Horstmann
9  */
10 public class BuggyButtonTest
11 {
12     public static void main(String[] args)
13     {
14         EventQueue.invokeLater(new Runnable()
15         {
16             public void run()
17             {
18                 JFrame frame = new BuggyButtonFrame();
19                 frame.setTitle("BuggyButtonTest");
20                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21                 frame.setVisible(true);
22             }
14 }
```

```
23     });
24 }
25 }
26 class BuggyButtonFrame extends JFrame
27 {
28     private static final int DEFAULT_WIDTH = 300;
29     private static final int DEFAULT_HEIGHT = 200;
30
31     public BuggyButtonFrame()
32     {
33         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34
35         // ввести панель в фрейм
36
37         BuggyButtonPanel panel = new BuggyButtonPanel();
38         add(panel);
39     }
40 }
41
42 class BuggyButtonPanel extends JPanel
43 {
44     public BuggyButtonPanel()
45     {
46         ActionListener listener = new ButtonListener();
47
48         JButton yellowButton = new JButton("Yellow");
49         add(yellowButton);
50         yellowButton.addActionListener(listener);
51
52         JButton blueButton = new JButton("Blue");
53         add(blueButton);
54         blueButton.addActionListener(listener);
55
56         JButton redButton = new JButton("Red");
57         add(redButton);
58         redButton.addActionListener(listener);
59     }
60
61     private class ButtonListener implements ActionListener
62     {
63         public void actionPerformed(ActionEvent event)
64         {
65             String arg = event.getActionCommand();
66             if (arg.equals("yellow")) setBackground(Color.yellow);
67             else if (arg.equals("blue")) setBackground(Color.blue);
68             else if (arg.equals("red")) setBackground(Color.red);
69         }
70     }
71 }
```

Для запуска отладчика в Eclipse выберите команду меню Run⇒Debug As⇒Java Application (Выполнить⇒Отладить как⇒Приложение на Java). Отлаживаемая программа запустится. Установите точку прерывания на первой строке кода в теле метода actionPerformed(), щелкнув правой кнопкой мыши на левом поле напротив

этой строки кода и выбрав команду Toggle Breakpoint (Включить/Выключить точку прерывания) из контекстного меню.

Точка прерывания будет достигнута, как только виртуальная машина Java начнет выполнять код метода `actionPerformed()`. Для этого щелкните в окне отлаживаемой программы на кнопке Yellow (Желтый). Отладчик остановится в начале метода `actionPerformed()`, как показано на рис. 11.6.

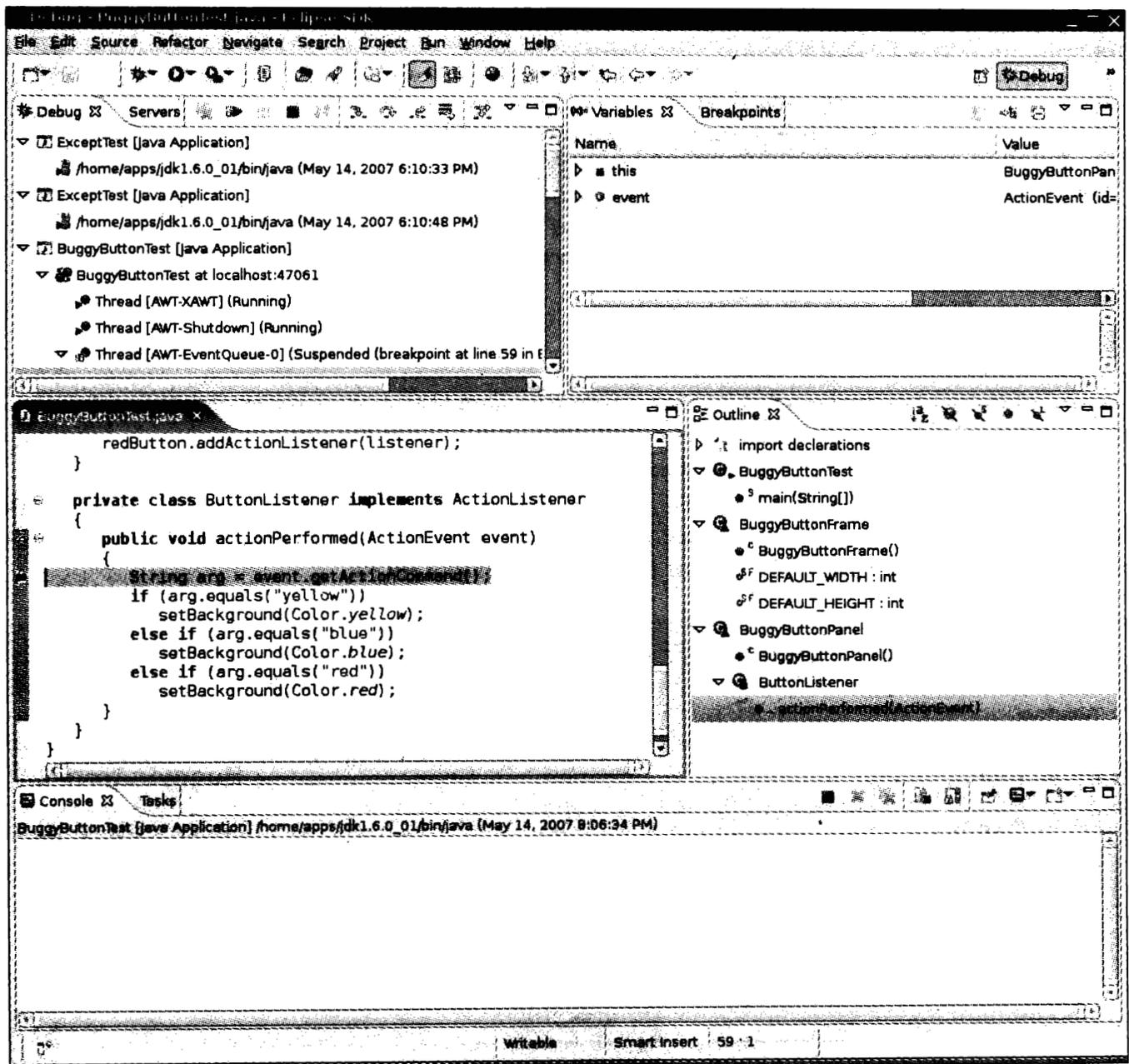


Рис. 11.6. Точка прерывания в отладчике

Для пошагового исполнения программы предусмотрено две основные команды. По команде `Step Into` осуществляется вход в каждый вызов метода, а по команде `Step Over` — переходит к следующей строке кода, не входя в тело вызываемых методов. Для выбора этих команд в Eclipse предоставляются пункты меню `Run` ⇒ `Step Into` и `Run` ⇒ `Step Over` или оперативные клавиши `<F5>` и `<F6>` соответственно. Итак, выберите команду `Step Over` два раза подряд и посмотрите, к чему это приведет. Как следует из приведенного ниже рисунка, это явно не то, что должно было бы произойти.

```
BuggyButtonTest.java ✕
    redButton.addActionListener(listener);
}

private class ButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        String arg = event.getActionCommand();
        if (arg.equals("yellow"))
            setBackground(Color.yellow);
        // setBackground(Color.blue);
        else if (arg.equals("red"))
            setBackground(Color.red);
    }
}
}
```

Предполагалось, что в программе должен быть вызван метод `setColor(Color.yellow)`, по завершении которого должен произойти возврат в вызывающую часть программы. В поисках ошибки исследуйте содержимое локальных переменных, чтобы выяснить значение переменной `arg`, как показано ниже.

Name	Value
▶ ■ this	BuggyButtonPan
▶ ● event	ActionEvent (id=
▶ ● arg	"Yellow"

Теперь вы можете ясно видеть, что случилось. В переменной `arg` было строковое значение "Yellow", начинающее с прописной буквы **Y**, а сравнение осуществляется со строкой "yellow", начинающейся со строчной буквы **y**, как показано ниже.

```
if (arg.equals("yellow"))
```

Чтобы выйти из отладчика, выберите команду меню `Run ⇒ Terminate` (Выполнить ⇒ Завершить). В отладчике Eclipse предусмотрено немало других дополнительных команд, но, ограничиваясь даже самой простой методикой отладки, которая была продемонстрирована выше, можно продвинуться достаточно далеко. У других отладчиков, как, например, в ИСР NetBeans, имеется аналогичный набор команд.

В этой главе вы ознакомились с обработкой исключений, а также получили полезные рекомендации по поводу тестирования и отладки прикладных программ. А в двух последующих главах речь пойдет об обобщенном программировании и его наиболее важном применении: архитектуре коллекций Java.

Обобщенное программирование

В этой главе...

- ▶ Назначение обобщенного программирования
- ▶ Определение простого обобщенного класса
- ▶ Обобщенные методы
- ▶ Ограничения на переменные типа
- ▶ Обобщенный код и виртуальная машина
- ▶ Ограничения и пределы обобщений
- ▶ Правила наследования обобщенных типов
- ▶ Подстановочные типы
- ▶ Рефлексия и обобщения

Обобщенные классы представляют собой наиболее значительное изменение в программировании на Java с момента выпуска версии 1.0. Появление обобщений в версии Java 5.0 SE стало результатом реализации самых первых требований к спецификации Java (Java Specification Requests — JSR 14), которые были сформулированы еще в 1999 г. Группа экспертов потратила около пяти лет на разработку спецификаций и тестовые реализации.

Обобщения понадобились потому, что они позволяют писать более безопасный код, который легче читается, чем код, перегруженный переменными типа `Object` и приведениями типов. Обобщения особенно полезны для классов коллекций вроде вездесущего класса `ArrayList`.

Обобщения похожи, по крайней мере, внешне, на шаблоны в C++. В языке C++, как и в Java, шаблоны впервые были внедрены для поддержки строго типизированных

коллекций. Но с годами им были найдены и другие применения. Проработав материал этой главы, вы, возможно, найдете новые, ранее неизвестные применения обобщениям в своих программах.

Назначение обобщенного программирования

Обобщенное программирование означает написание кода, который может быть неоднократно использован с объектами самых разных типов. Так, если нет желания программировать отдельные классы для составления коллекций из объектов типа `String` и `File`, достаточно собрать эти объекты в коллекцию, воспользовавшись единственным обобщенным классом `ArrayList`. И это всего лишь один простой пример обобщенного программирования.

До появления версии Java SE 5.0 обобщенное программирование на Java всегда реализовывалось посредством *наследования*. Так, в классе `ArrayList` просто поддерживался массив ссылок на класс `Object` следующим образом:

```
public class ArrayList // до появления обобщенных классов
{
    private Object[] elementData;
    . . .
    public Object get(int i) { . . . }
    public void add(Object o) { . . . }
}
```

Такой подход порождает две серьезные проблемы. Всякий раз, когда извлекается значение, необходимо выполнить приведение типа, как показано ниже.

```
ArrayList files = new ArrayList();
. . .
String filename = (String) files.get(0);
```

Более того, в таком коде отсутствует проверка ошибок. Ничто не мешает добавить значения любого класса следующим образом:

```
files.add(new File(". . ."));
```

Этот вызов компилируется и выполняется без ошибок. Но затем попытка привести результат выполнения метода `get()` к типу `String` приведет к ошибке.

Обобщения предлагают лучшее решение: *параметры типа*. Класс `ArrayList` теперь принимает параметр типа, обозначающий тип элементов коллекции, как показано ниже. Благодаря этому код получается более удобочитаемым. Теперь становится сразу понятно, что этот конкретный списочный массив содержит объекты типа `String`.

```
ArrayList<String> files = new ArrayList<String>();
```



НА ЗАМЕТКУ! Как упоминалось ранее, начиная с версии Java SE 7, обобщенный тип можно опустить в объявлении конструктора класса, как показано ниже.

```
ArrayList<String> files = new ArrayList<>();
```

Обобщенный тип выводится из типа переменной.

Данные о типах могут быть полезными и для компилятора. Так, в приведенном ниже вызове метода `get()` никакого приведения типов не требуется. Ведь компилятору известно, что возвращаемым типом является `String`, а не `Object`.

```
String filename = files.get(0);
```

Компилятору также известно, что у метода `add()` из класса `ArrayList<String>` имеется параметр типа `String`. Это намного безопаснее, чем иметь дело с параметром типа `Object`. Теперь компилятор может проконтролировать, не подставлен ли объект неверного типа в исходном коде. Например, следующая строка кода не скомпилируется:

```
files.add(new File(". . .")); // в коллекцию типа ArrayList<String>
                               // можно вводить только объекты типа String
```

Ошибка компиляции — это намного лучше, чем исключение в связи с неправильным приведением типов во время выполнения. Привлекательность параметров типа в том и состоит, что они делают исходный код программы более удобочитаемым и безопасным.

На кого рассчитано обобщенное программирование

Пользоваться такими обобщенными классами, как `ArrayList`, очень просто. И большинство программирующих на Java просто применяют типы вроде `ArrayList<String>`, как будто они являются такой же неотъемлемой частью языка Java, как и массивы типа `String[]`. (Разумеется, списочные массивы лучше простых массивов, поскольку они допускают автоматическое расширение.)

Но реализовать обобщенный класс не так-то просто. Те, кто будет пользоваться обобщенным кодом, попытаются подставлять всевозможные классы вместо предусмотренных параметров типа. Они надеются, что все будет работать без досадных ограничений и запутанных сообщений об ошибках. Поэтому главная задача обобщенно программирующего — предвидеть все возможные в дальнейшем применения разработанного им обобщенного класса.

Насколько трудно этого добиться? Приведем пример типичного затруднения, которое пришлось преодолевать разработчикам стандартной библиотеки классов. Класс `ArrayList` содержит метод `addAll()`, предназначенный для добавления элементов из другой коллекции. Так, у программиста может возникнуть потребность добавить все элементы из коллекции типа `ArrayList<Manager>` в коллекцию типа `ArrayList<Employee>`. Но обратное, разумеется, недопустимо. Как же разрешить один вызов и запретить другой? Создатели Java нашли искусный выход из этого затруднительного положения, внедрив понятие *подстановочного типа*. Подстановочные типы довольно абстрактны, но они позволяют разработчику библиотеки сделать методы как можно более гибкими.

Обобщенное программирование разделяется на три уровня квалификации. На элементарном уровне можно просто пользоваться обобщенными классами (как правило, коллекциями типа `ArrayList`, даже не задумываясь, как они работают). Большинство прикладных программистов предпочитают оставаться на этом уровне до тех пор, пока дело не заладится. Сочетая разные обобщенные классы или же имея дело с унаследованным кодом, в котором ничего неизвестно о параметрах типа, можно столкнуться с непонятным сообщением об ошибках. В подобной ситуации нужно иметь достаточно ясное представление об обобщениях Java, чтобы устранить неполадку системно, а не “методом тыка”. И, наконец, можно решиться на реализацию своих собственных обобщенных классов и методов.

Прикладным программистам, вероятно, не придется писать много обобщенного кода. Разработчики JDK уже выполнили самую трудную работу и предусмотрели параметры типа для всех классов коллекций. В связи с этим можно сформулировать следующее эмпирическое правило: от применения параметров типа выигрывает

только тот код, в котором традиционно присутствует много операций приведения от самых общих типов, как, например, класс `Object` или интерфейс `Comparable`.

Из этой главы вы узнаете все, что следует знать для реализации вашего собственного обобщенного кода. Но мы надеемся, что наши читатели используют полученные знания прежде всего для отыскания причин неполадок в своих программах, а также для удовлетворения любопытства относительно внутреннего устройства параметризованных классов коллекций.

Определение простого обобщенного класса

Обобщенным называется класс с одной или более переменной типа. В этой главе в качестве примера рассматривается простой обобщенный класс `Pair`. Этот класс выбран для примера потому, что он позволяет сосредоточить основное внимание на механизме обобщений, не вдаваясь в подробности сохранения данных. Ниже приведен исходный код обобщенного класса `Pair`.

```
public class Pair<T>
{
    private T first;
    private T second;

    public Pair() { first = null; second = null; }
    public Pair(T first, T second)
        { this.first = first; this.second = second; }

    public T getFirst() { return first; }
    public T getSecond() { return second; }

    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }
}
```

В классе `Pair` вводится переменная типа `T`, заключенная в угловые скобки (`<>`) после имени самого класса. У обобщенного класса может быть больше одной переменной типа. Например, класс `Pair` можно было бы определить с разными типами для первого и второго поля следующим образом:

```
public class Pair<T, U> { . . . }
```

Переменные типа используются повсюду в определении класса для обозначения типов, возвращаемых методами, а также типов полей и локальных переменных. Ниже приведен пример объявления переменной типа.

```
private T first; // использовать переменную типа
```



НА ЗАМЕТКУ! В именах переменных типа принято употреблять заглавные буквы и стремиться к их краткости. Так, в библиотеке Java буквой **E** обозначается тип элемента коллекции, буквами **K** и **V** — типы ключей и значений в таблице, а буквой **T** (и соседними буквами **S** и **U** при необходимости) — “любой тип вообще”, как поясняется в документации.

Экземпляр обобщенного типа создается путем подстановки имени типа вместо переменной типа следующим образом:

```
Pair<String>
```

Результат такой подстановки следует рассматривать как обычный класс с конструкторами:

```
Pair<String>()
Pair<String>(String, String)
```

и методами:

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

Иными словами, обобщенный класс действует как фабрика обычных классов. В примере программы из листинга 12.1 применяется класс `Pair`. В статическом методе `minmax()` осуществляется перебор элементов массива с одновременным вычислением минимального и максимального значений. Для возврата обоих значений используется объект типа `Pair`. Напомним, что в методе `compareTo()` сравниваются две строки и возвращается нулевое значение, если символьные строки одинаковы, отрицательное числовое значение — если первая символьная строка следует прежде второй в лексикографическом порядке, в противном случае — положительное числовое значение.



НА ЗАМЕТКУ C++! На первый взгляд обобщенные классы в Java похожи на шаблонные классы в C++. Единственное очевидное отличие состоит в том, что в Java не предусмотрено специальное ключевое слово `template`. Но, как будет показано далее в главе, у этих двух механизмов обобщений имеются существенные отличия.

Листинг 12.1. Исходный код из файла `pair1/PairTest1.java`

```
1 package pair1;
2
3 /**
4  * @version 1.01 2012-01-26
5  * @author Cay Horstmann
6  */
7 public class PairTest1
8 {
9     public static void main(String[] args)
10    {
11        String[] words = { "Mary", "had", "a", "little", "lamb" };
12        Pair<String> mm = ArrayAlg.minmax(words);
13        System.out.println("min = " + mm.getFirst());
14        System.out.println("max = " + mm.getSecond());
15    }
16 }
17
18 class ArrayAlg
19 {
20     /**
21     * Получает символьные строки с минимальным и
22     * максимальным значениями среди элементов массива
23     * @param a Массив символьных строк
24     * @return Пара минимального и максимального значений или
25     * пустое значение, если параметр a имеет пустое значение
26     */
```

```
27 public static Pair<String> minmax(String[] a)
28 {
29     if (a == null || a.length == 0) return null;
30     String min = a[0];
31     String max = a[0];
32     for (int i = 1; i < a.length; i++)
33     {
34         if (min.compareTo(a[i]) > 0) min = a[i];
35         if (max.compareTo(a[i]) < 0) max = a[i];
36     }
37     return new Pair<>(min, max);
38 }
39 }
```

Обобщенные методы

В предыдущем разделе было показано, как определяется обобщенный класс. Имеется также возможность определить отдельный метод с параметрами (или переменными) типа следующим образом:

```
class ArrayAlg
{
    public static <T> T getMiddle(T... a)
    {
        return a[a.length / 2];
    }
}
```

Этот метод определен в обычном, а не в обобщенном классе. Тем не менее это обобщенный метод, на что указывают угловые скобки и переменная типа. Обратите внимание на то, что переменная типа вводится после модификаторов (в данном случае `public static`) и перед возвращаемым типом.

Обобщенные методы можно определять как в обычных классах, так и в обобщенных. Когда вызывается обобщенный метод, ему можно передать конкретные типы данных, заключая их в угловые скобки перед именем метода, как показано ниже.

```
String middle = ArrayAlg.<String>getMiddle("John", "Q.", "Public");
```

В данном случае (как и вообще) при вызове метода можно пропустить параметр типа `String`. У компилятора имеется достаточно информации, чтобы вывести из такого обобщения именно тот метод, который требуется вызвать. Он сравнивает тип параметра метода (т.е. `String[]`) с обобщенным типом `T[]`) и приходит к выводу, что вместо обобщенного типа `T` следует подставить тип `String`, что равнозначно следующему вызову:

```
String middle = ArrayAlg.getMiddle("John", "Q.", "Public");
```

Выводимость типов в обобщенных методах практически всегда действует исправно. Но иногда компилятор ошибается, и тогда приходится разбираться в тех ошибках, о которых он сообщает. Рассмотрим следующий пример:

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

При выполнении этой строки кода компилятор выведет сообщение об ошибке загадочного содержания, но суть его в том, что данный код можно интерпретировать двояко и в обоих случаях — правильно. По существу, компилятор выполняет

автоупаковку параметра в один объект типа `Double` и два объекта типа `Integer`, а затем пытается найти общий супертип для этих классов. И таких типов два: класс `Number` и интерфейс `Comparable`, который сам является обобщенным. В этом случае для устранения ошибки методу следует передать все параметры со значениями типа `double`.



СОВЕТ. Петер Ван Дер Ахе (Peter von der Ahe) рекомендует следующий прием, если требуется выяснить, какой тип компилятор выводит при вызове обобщенного метода: намеренно допустите ошибку и изучите полученное в итоге сообщение об ошибке. В качестве примера рассмотрим следующий вызов: `ArrayAlg.getMiddle("Hello", 0, null)`. Если присвоить полученный результат переменной ссылки на объект типа `JButton`, что заведомо неверно, то в конечном итоге будет получено следующее сообщение об ошибке:

```
found:
java.lang.Object&java.io.Serializable&java.lang.Comparable<? extends
java.lang.Object&java.io.Serializable&java.lang.Comparable<?>>
```

Это означает, что результат вызова данного метода можно присвоить переменным ссылки на объекты типа `Object`, `Serializable` или `Comparable`.



НА ЗАМЕТКУ C++! В C++ параметр типа указывается после имени метода. Это может привести к неприятной неоднозначности при синтаксическом анализе кода. Например, выражение `g(f<a,b>(c))` может означать следующее: "вызвать метод `g()` с результатом вызова `f<a,b>(c)`" или же "вызвать метод `g()` с двумя логическими значениями `f<a и b>(c)`".

Ограничения на переменные типа

Иногда класс или метод нуждается в наложении ограничений на переменные типа. Приведем характерный тому пример, в котором требуется вычислить наименьший элемент массива следующим образом:

```
class ArrayAlg
{
    public static <T> T min(T[] a) // почти верно
    {
        if (a == null || a.length == 0) return null;
        T smallest = a[0];
        for (int i = 1; i < a.length; i++)
            if (smallest.compareTo(a[i]) > 0) smallest = a[i];
        return smallest;
    }
}
```

Но здесь возникает затруднение. Обратите внимание на тело метода `min()`. Переменная `smallest` относится к типу `T`, а это означает, что она может быть объектом произвольного класса. Но откуда известно, имеется ли в классе типа `T` метод `compareTo()`?

Выход из этого затруднительного положения состоит в том, чтобы наложить ограничение на тип `T` и вместо него подставлять только класс, реализующий `Comparable` — стандартный интерфейс с единственным методом `compareTo()`. Для этого достаточно наложить *ограничение* на переменную типа `T` следующим образом:

```
public static <T extends Comparable> T min(T[] a) . . .
```

В действительности интерфейс `Comparable` сам является обобщенным. Но мы не будем пока что принимать во внимание это обстоятельство и соответствующие предупреждения компилятора. В разделе “Подстановочные типы” далее в этой главе мы обсудим, как правильно пользоваться параметрами типа вместе с интерфейсом `Comparable`.

Теперь обобщенный метод `min()` может вызываться только с массивами классов, реализующих интерфейс `Comparable`, в том числе `String`, `Date` и т.п. А вызов `min()` с массивом типа `Rectangle` приведет к ошибке во время компиляции, поскольку класс `Rectangle` не реализует интерфейс `Comparable`.



НА ЗАМЕТКУ C++! В C++ нельзя ограничивать типы в параметрах шаблонов. Если создать экземпляр шаблона с неправильным типом, появится (зачастую невнятное) сообщение об ошибке в коде шаблона.

Вас может удивить, почему здесь используется ключевое слово `extends` вместо `implements`, ведь `Comparable` — это интерфейс? Так, обозначение

```
<T extends ограничивающий_тип>
```

означает, что тип `T` должен быть *подтипом* ограничивающего типа, причем к типу `T` и ограничивающему типу может относиться как интерфейс, так и класс. Ключевое слово `extends` выбрано потому, что это вполне благоразумное приближение понятия подтипа, и создатели Java не сочли нужным вместо этого вводить в язык новое ключевое слово.

Переменная типа или подстановки может иметь несколько ограничений, как показано в приведенном ниже примере. Ограничивающие типы разделяются знаком `&`, потому что запятые служат для разделения переменных типа.

```
T extends Comparable & Serializable
```

Как и в механизме наследования в Java, подтипов интерфейсов может быть сколько угодно, но только один из ограничивающих типов может быть классом. Если для ограничения служит класс, он должен быть первым в списке накладываемых ограничений.

В следующем примере программы из листинга 12.2 метод `minmax()` переделан на обобщенный. В этом методе вычисляется минимальное и максимальное значения в обобщенном массиве и возвращается объект обобщенного типа `Pair<T>`.

Листинг 12.2. Исходный код из файла `pair2/PairTest2.java`

```
1 package pair2;
2
3 import java.util.*;
4
5 /**
6  * @version 1.01 2012-01-26
7  * @author Cay Horstmann
8  */
9 public class PairTest2
10 {
11     public static void main(String[] args)
12     {
13         GregorianCalendar[] birthdays =
14         {
```

```
15     new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
16     new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A. Lovelace
17     new GregorianCalendar(1903, Calendar.DECEMBER, 3), // J. von Neumann
18     new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
19     };
20     Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
21     System.out.println("min = " + mm.getFirst().getTime());
22     System.out.println("max = " + mm.getSecond().getTime());
23 }
24 }
25
26 class ArrayAlg
27 {
28     /**
29     * Получает минимальное и максимальное значения
30     * из массива объектов типа T
31     * @param a Массив объектов типа T
32     * @return Пара минимального и максимального значений или
33     * пустое значение, если параметр a имеет пустое значение
34     */
35     public static <T extends Comparable> Pair<T> minmax(T[] a)
36     {
37         if (a == null || a.length == 0) return null;
38         T min = a[0];
39         T max = a[0];
40         for (int i = 1; i < a.length; i++)
41         {
42             if (min.compareTo(a[i]) > 0) min = a[i];
43             if (max.compareTo(a[i]) < 0) max = a[i];
44         }
45         return new Pair<>(min, max);
46     }
47 }
```

Обобщенный код и виртуальная машина

Виртуальная машина не имеет дела с объектами обобщенных типов — все объекты принадлежат обычным классам. В первоначальных вариантах реализаций обобщений можно было даже компилировать программу с обобщениями в файлы классов, которые способна исполнять виртуальная машина версии 1.0! Такая обратная совместимость при последующем усовершенствовании механизма обобщений Java была отменена.

Всякий раз, когда определяется обобщенный тип, автоматически создается соответствующий ему базовый (так называемый “сырой”) тип. Имя этого типа совпадает с именем обобщенного типа с удаленными параметрами типа. Переменные типа *стираются* и заменяются ограничивающими типами (или типом `Object`, если переменная не имеет ограничений). Например, базовый тип для обобщенного типа `Pair<T>` выглядит следующим образом:

```
public class Pair
{
    private Object first;
    private Object second;
    public Pair(Object first, Object second)
    {
```

```

    this.first = first;
    this.second = second;
}

public Object getFirst() { return first; }
public Object getSecond() { return second; }
public void setFirst(Object newValue) { first = newValue; }
public void setSecond(Object newValue) { second = newValue; }
}

```

Если `T` — неограниченная переменная типа, то ее тип просто заменяется на `Object`. В итоге получается обычный класс вроде тех, что реализовывались до появления обобщений в Java. Прикладные программы могут содержать разные варианты обобщенного класса `Pair`, в том числе `Pair<String>` или `Pair<GregorianCalendar>`, но в результате стирания все они приводятся к базовым типам `Pair`.



НА ЗАМЕТКУ C++! В этом отношении обобщения в Java заметно отличаются от шаблонов в C++. Для каждого экземпляра шаблона в C++ получается свой тип. Это неприятное явление называется "раздуванием кода шаблона". А Java этим недостатком не страдает.

Базовый тип заменяет тип переменных первым накладываемым на них ограничением или же типом `Object`, если никаких ограничений не предусмотрено. Например, у переменной типа в обобщенном классе `Pair<T>` отсутствуют явные ограничения, и поэтому базовый тип заменяет обобщенный тип `T` на `Object`. Допустим, однако, что объявлен несколько иной обобщенный тип, как показано ниже.

```

public class Interval<T extends Comparable &
    Serializable> implements Serializable
{
    private T lower;
    private T upper;
    . . .
    public Interval(T first, T second)
    {
        if (first.compareTo(second) <= 0) { lower = first; upper = second; }
        else { lower = second; upper = first; }
    }
}

```

Соответствующий ему базовый тип выглядит следующим образом:

```

public class Interval implements Serializable
{
    private Comparable lower;
    private Comparable upper;
    . . .
    public Interval(Comparable first, Comparable second) { . . . }
}

```



НА ЗАМЕТКУ! А что, если поменять местами ограничения: `class Interval<Serializable & Comparable>`? В этом случае базовый тип заменит обобщенный `T` на `Serializable`, а компилятор произведет там, где требуется, приведение к типу `Comparable`. Поэтому ради эффективности в конце списка ограничений следует размещать отмечающие интерфейсы (т.е. интерфейсы без методов).

Преобразование обобщенных выражений

Когда в программе вызывается обобщенный метод, компилятор вводит операции приведения типов при стирании возвращаемого типа. Рассмотрим в качестве примера следующую последовательность операторов:

```
Pair<Employee> buddies = . . . ;
Employee buddy = buddies.getFirst();
```

В результате стирания из метода `getFirst()` возвращается тип `Object`. Поэтому компилятор автоматически вводит приведение к типу `Employee`. Это означает, что компилятор преобразует вызов данного метода в следующие две команды для виртуальной машины:

- вызвать метод базового типа `Pair.getFirst()`;
- привести тип `Object` возвращаемого объекта к типу `Employee`.

Операции приведения типов вводятся также при обращении к обобщенному полю. Допустим, что поля `first` и `second` открыты, т.е. `public`. (Возможно, это и не самый лучший стиль программирования, но вполне допустимый в Java.) Тогда при преобразовании приведенного ниже выражения в получаемый в итоге байт-код также будут введены операции приведения типов.

```
Employee buddy = buddies.first;
```

Преобразование обобщенных методов

Стирание типов происходит и в обобщенных методах. Программисты обычно воспринимают обобщенные методы как целое семейство методов вроде следующего:

```
public static <T extends Comparable> T min(T[] a)
```

Но после стирания типов остается только один приведенный ниже метод. Обратите внимание на то, что параметр обобщенного типа `T` стирается, а остается только ограничивающий тип `Comparable`.

```
public static Comparable min(Comparable[] a)
```

Стирание типов в обобщенных методах приводит к некоторым осложнениям. Рассмотрим следующий пример кода:

```
class DateInterval extends Pair<Date>
{
    public void setSecond(Date second)
    {
        if (second.compareTo(getFirst()) >= 0)
            super.setSecond(second);
    }
    . . .
}
```

В этом фрагменте кода интервал дат задается парой объектов типа `Date`, и поэтому соответствующие методы требуется переопределить, чтобы второе сравниваемое значение не было меньше первого. В результате стирания данный класс преобразуется в следующий:

```
class DateInterval extends Pair // после стирания
{
    public void setSecond(Date second) { . . . }
    . . .
}
```

Как ни странно, но имеется и другой метод `setSecond()`, унаследованный от класса `Pair`, а именно

```
public void setSecond(Object second)
```

И это совершенно другой метод, поскольку он имеет параметр другого типа: `Object` вместо `Date`. Но он *не* должен быть другим. Рассмотрим следующую последовательность операторов:

```
DateInterval interval = new DateInterval(. . .);
Pair<Date> pair = interval; // допускается присваивание суперклассу
pair.setSecond(aDate);
```

Предполагается, что вызов метода `setSecond()` является полиморфным, и поэтому вызывается соответствующий метод. А поскольку переменная `pair` ссылается на объект типа `DateInterval`, это должен быть вызов `DateInterval.setSecond()`. Но дело в том, что стирание типов мешает соблюдению принципа полиморфизма. В качестве выхода из этого затруднительного положения компилятор формирует следующий *мостовой метод* в классе `DateInterval`:

```
public void setSecond(Object second) { setSecond((Date) second); }
```

Для того чтобы стал понятнее этот механизм, проанализируем выполнение приведенного ниже оператора.

```
pair.setSecond(aDate)
```

В объявлении переменной `pair` указан тип `Pair<Date>`, и к этому типу относится только один метод под именем `setSecond`, а именно `setSecond(Object)`. Виртуальная машина вызывает этот метод для того объекта, на который ссылается переменная `pair`. Этот объект относится к типу `DateInterval`, и поэтому вызывается метод `DateInterval.setSecond(Object)`. Именно он и является синтезированным мостовым методом. Ведь он, в свою очередь, вызывает метод `DateInterval.setSecond(Date)`, что, собственно говоря, и требуется.

Мостовые методы могут быть еще более необычными. Допустим, метод из класса `DateInterval` также переопределяет метод `getSecond()`, как показано ниже.

```
class DateInterval extends Pair<Date>
{
    public Date getSecond() { return (Date) super.getSecond().clone(); }
    . . .
}
```

В классе `DateInterval` имеются следующие два метода под именем `getSecond`:

```
Date getSecond() // определен в классе DateInterval
Object getSecond() // переопределяет метод из класса Pair
// для вызова первого метода
```

Написать такой код на Java без параметров нельзя. Ведь было бы неверно иметь два метода с одинаковыми типами параметров. Но в виртуальной машине типы параметров и *возвращаемый тип* определяют метод. Поэтому компилятор может сформировать байт-код для двух методов, отличающихся только возвращаемым типом, и виртуальная машина правильно ведет себя в подобной ситуации.



НА ЗАМЕТКУ! Применение мостовых методов не ограничивается только обобщенными типами. Как упоминалось в главе 5, вполне допустимо определять в методе более ограниченный возвращаемый тип, когда он переопределяет другой метод. Так, в приведенном ниже фрагменте кода

методы `Object.clone()` и `Employee.clone()` имеют так называемые *ковариантные возвращаемые типы*.

```
public class Employee implements Cloneable
{
    public Employee clone() throws CloneNotSupportedException { ... }
}
```

На самом деле в классе `Object` имеются два таких метода:

```
Employee clone() // определен выше
Object clone() // синтезированный мостовой метод, переопределяющий
                // метод Object.clone()
```

При этом синтезированный мостовой метод вызывает вновь определенный метод.

Таким образом, о преобразовании обобщений в Java нужно запомнить следующее.

- Для виртуальной машины обобщений не существует, но имеются только обычные классы и методы.
- Все параметры типа заменяются ограничивающими типами.
- Мостовые методы синтезируются для соблюдения принципа полиморфизма.
- Операции приведения типов вводятся по мере надобности для обеспечения типовой безопасности.

Вызов унаследованного кода

Внедрением обобщений в Java преследовалась главная цель: обеспечить совместимость обобщенного и унаследованного кода. Обратимся к конкретному примеру. Для установки меток в компоненте типа `JSlider` используется следующий метод:

```
void setLabelTable(Dictionary table)
```

где `Dictionary` является базовым типом, поскольку класс `JSlider` был реализован до внедрения обобщений в Java. Но для заполнения словаря следует использовать базовый тип, как показано ниже.

```
Dictionary<Integer, Component> labelTable = new Hashtable<>();
labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
. . .
```

При попытке передать объект обобщенного типа `Dictionary<Integer, Component>` методу `setLabelTable()` компилятор выдает предупреждение, как показано в приведенной ниже строке кода.

```
slider.setLabelTable(labelTable); // ПРЕДУПРЕЖДЕНИЕ!
```

Ведь компилятору неизвестно наверное, что именно метод `setLabelTable()` может сделать с объектом типа `Dictionary`. Так, этот метод может заменить все ключи символьными строками, нарушив гарантию того, что ключи должны иметь тип `Integer`. Поэтому при выполнении последующих операций приведения типов могут возникнуть неприятные исключения.

С таким предупреждением вряд можно что-нибудь поделать, кроме его осмысления и попытки предугадать, что же компонент типа `JSlider` собирается сделать

с объектом типа `Dictionary` В данном случае совершенно ясно, что компонент типа `JSlider` только вводит информацию, так что предупреждением компилятора можно пренебречь.

А теперь рассмотрим противоположный случай, когда объект базового типа получается от унаследованного класса. Его можно присвоить переменной обобщенного типа, но тогда будет выдано предупреждение, как показано в приведенной ниже строке кода.

```
Dictionary<Integer, Components> labelTable  
    = slider.getLabelTable(); // ПРЕДУПРЕЖДЕНИЕ!
```

И в этом случае следует проанализировать предупреждение и убедиться в том, что таблица меток действительно содержит объекты типа `Integer` и `Component`, хотя нет никакой гарантии, что они там присутствуют. В частности, злоумышленник может установить другой объект типа `Dictionary` в компоненте типа `JSlider`. Но опять же эта ситуация не хуже той, что была до внедрения обобщений. В худшем случае ваша программа сгенерирует исключение.

Итак, обратив внимание на предупреждение компилятора, можно воспользоваться аннотацией для того, чтобы оно исчезло. Такую аннотацию следует разместить перед локальной переменной следующим образом:

```
@SuppressWarnings("unchecked")  
Dictionary<Integer, Components> labelTable  
    = slider.getLabelTable(); // без предупреждения
```

Эта аннотация отменяет проверку всего кода в теле метода.

Ограничения и пределы обобщений

В последующих разделах будет рассмотрен целый ряд ограничений, которые следует учитывать, обращаясь с обобщениями в Java. Большинство этих ограничений являются следствием стирания типов.

Параметрам типа нельзя приписывать простые типы

Примитивный тип нельзя подставить вместо типа параметра. Это означает, что не бывает объекта типа `Pair<double>`, а только объект типа `Pair<Double>`. Причину, конечно, следует искать в стирании типов. После такого стирания в классе `Pair` отсутствуют поля типа `Object`, и поэтому их нельзя использовать для хранения значений типа `double`.

Это досадно, но такое ограничение согласуется с особым положением примитивных типов в Java. Этот недостаток не носит фатального характера. Ведь существует всего восемь простых типов данных, а обработать их всегда можно с помощью отдельных классов и методов, когда нельзя подставить вместо них типы-оболочки.

Во время выполнения можно запрашивать только базовые типы

В виртуальной машине объекты всегда имеют определенный необобщенный тип. Поэтому все запросы типов во время выполнения дают только базовый тип. Например, следующий оператор на самом деле только проверяет, является ли а экземпляром `Pair` любого типа:

```
if (a instanceof Pair<String>) // ОШИБКА!
```

Это же справедливо и в отношении следующего оператора:

```
if (a instanceof Pair<T>) // ОШИБКА!
```

или такого оператора приведения типов:

```
Pair<String> p = (Pair<String>) a; // ПРЕДУПРЕЖДЕНИЕ! Проверить можно только
// принадлежность переменной a к типу Pair
```

Чтобы напомнить о возможной опасности, компилятор выдает ошибку (при операции `instanceof`) или предупреждение (при приведении типов) всякий раз, когда делается запрос, принадлежит ли объект к обобщенному типу. Аналогично метод `getClass()` всегда возвращает базовый тип, как показано в приведенном ниже примере кода. Результатом сравнения оказывается логическое значение `true`, потому что при обоих вызовах метода `getClass()` возвращается объект типа `Pair.class`.

```
Pair<String> stringPair = . . . ;
Pair<Employee> employeePair = . . . ;
if (stringPair.getClass() == employeePair.getClass()) // равны!
```

Массивы параметризованных типов недопустимы

Нельзя объявить массив параметризованных типов вроде следующего:

```
Pair<String>[] table = new Pair<String>[10]; // ОШИБКА!
```

Что же здесь не так? После стирания типом таблицы становится `Pair[]`. Но его можно преобразовать в тип `Object[]` следующим образом:

```
Object[] objarray = table;
```

В массиве запоминается тип его элементов и генерируется исключение типа `ArrayStoreException`, если попытаться сохранить в нем элемент неверного типа, как показано ниже.

```
objarray[0] = "Hello"; // ОШИБКА! Типом компонента является Pair
```

Но стирание делает этот механизм неэффективным для обобщенных типов. Приведенное ниже присваивание пройдет проверку на сохранение в массиве, но выдаст ошибку соблюдения типов. Именно поэтому массивы параметризованных типов не допускаются.

```
objarray[0] = new Pair<Employee>();
```

Следует, однако, иметь в виду, что не допускается только создание подобных массивов. И хотя разрешается, например, объявить переменную типа `Pair<String>[]`, тем не менее ее нельзя инициализировать с помощью оператора `Pair<String>[10]`.



НА ЗАМЕТКУ! Допускается объявлять массивы элементов подстановочных типов, а затем приводить их к соответствующим типам следующим образом:

```
Pair<String>[] table = (Pair<String>[]) new Pair<?>[10];
```

Хотя результат такой операции не гарантирует полную безопасность. Так, если сначала сохранить объект типа `Pair<Employee>` в элементе массива `table[0]`, а затем вызвать для него метод `table[0].getFirst()` из класса `String`, то будет сгенерировано исключение типа `ClassCastException`.



СОВЕТ. Если объекты параметризованных типов требуется хранить в коллекциях, пользуйтесь обобщенным классом `ArrayList`, например `ArrayList<Pair<String>>`. Это безопасно и эффективно.

Предупреждения о переменном числе аргументов

Как было показано в предыдущем разделе, в Java не поддерживаются массивы обобщенных типов. А в этом разделе будет рассмотрен следующий связанный с этим вопрос: передача экземпляров обобщенных типов методу с переменным числом аргументов.

Рассмотрим в качестве примера следующий простой метод с переменным числом аргументов:

```
public static <T> void addAll(Collection<T> coll, T... ts)
{
    for (t : ts) coll.add(t);
}
```

Напомним, что параметр `ts` на самом деле является массивом, содержащим все предоставляемые аргументы. А теперь рассмотрим вызов этого метода в приведенном ниже фрагменте кода.

```
Collection<Pair<String>> table = ...;
Pair<String> pair1 = ...;
Pair<String> pair2 = ...;
addAll(table, pair1, pair2);
```

Для вызова этого метода в виртуальной машине Java придется сформировать массив объектов типа `Pair<String>`, что не по правилам. Но эти правила были ослаблены, чтобы уведомлять в подобных случаях только о предупреждении.

Подавить выдачу такого предупреждения можно двумя способами. Во-первых, ввести аннотацию `@SuppressWarnings("unchecked")` в тело метода, содержащего вызов метода `addAll()`. И, во-вторых, начиная с версии Java SE 7, ввести аннотацию `@SafeVarargs` в тело самого метода `addAll()`, как показано ниже.

```
@SafeVarargs
public static <T> void addAll(Collection<T> coll, T... ts)
```

Теперь этот метод можно вызывать с аргументами обобщенных типов. Приведенную выше аннотацию можно ввести в любые методы, выбирающие элементы из массива параметров — наиболее характерного примера употребления переменного числа аргументов.



НА ЗАМЕТКУ! С помощью аннотации `@SafeVarargs` можно преодолеть ограничение на создание обобщенного массива. Для этой цели служит следующий метод:

```
@SafeVarargs static <E> E[] array(E... array) { return array; }
```

Этот метод можно далее вызвать следующим образом:

```
Pair<String>[] table = array(pair1, pair2);
```

На первый взгляд такой прием удобен, но он чреват следующей опасностью:

```
Object[] objarray = table;
objarray[0] = new Pair<Employe>();
```

Этот фрагмент кода будет выполняться без исключения типа `ArrayStoreException`, поскольку в массиве сохраняются данные только стертого типа. Но если обратиться к элементу массива `table[0]` в другом месте кода, такое исключение будет сгенерировано.

Нельзя создавать экземпляры переменных типа

Переменные типа нельзя использовать в выражениях вроде `new T(...)`, `new T[...]` или `T.class`. Например, следующий конструктор `Pair<T>` недопустим:

```
public Pair() { first = new T(); second = new T(); } // ОШИБКА!
```

Стирание типов может изменить обобщенный тип `T` на `Object`, а вызывать конструктор `new Object()`, конечно, не стоит. В качестве обходного приема можно конструировать обобщенные объекты через рефлексию, вызывая метод `Class.newInstance()`. К сожалению, это совсем не простой прием. Так, сделать приведенный ниже вызов нельзя, поскольку выражение `.class` недопустимо.

```
first = T.class.newInstance(); // ОШИБКА!
```

Вместо этого придется разработать прикладной интерфейс API, чтобы обращаться с объектом типа `Class`, как показано ниже.

```
public static <T> Pair<T> makePair(Class<T> cl)
{
    try { return new Pair<>(cl.newInstance(), cl.newInstance()) }
    catch (Exception ex) { return null; }
}
```

Этот метод может быть вызван следующим образом:

```
Pair<String> p = Pair.makePair(String.class);
```

Следует, однако, иметь в виду, что класс `Class` сам является обобщенным. Например, `String.class` — это экземпляр (на самом деле единственный) типа `Class<String>`. Поэтому метод `makePair()` может выводить тип создаваемой пары. Но построить обобщенный массив нельзя, как показано ниже. Стирание типов может вынудить этот метод всегда строить массив `Comparable[2]`.

```
public static <T extends Comparable> T[] minmax(T[] a) { T[] mm =
    new T[2]; . . . } // ОШИБКА!
```

Если массив используется только как закрытое поле экземпляра класса, его можно объявить как `Object[]` и прибегнуть к приведению типов при извлечении из него элементов. Например, класс `ArrayList` может быть реализован следующим образом:

```
public class ArrayList<E>
{
    private Object[] elements;
    . . .
    @SuppressWarnings("unchecked") public E get(int n)
    { return (E) elements[n]; }
    public void set(int n, E e) { elements[n] = e; } // приведение типов
                                                // не требуется
}
```

Но конкретная его реализация не настолько ясна, как показано ниже. В этом коде приведение к типу `E[]` совершенно ложно, но стирание типов делает это незаметным.

```
public class ArrayList<E>
{
    private E[] elements;
    . . .
    public ArrayList() { elements = (E[]) new Object[10]; }
}
```

Такой прием не подходит для метода `minmax()`, поскольку он возвращает массив обобщенного типа `T[]`. И если сделать ложное заключение о типе, то во время выполнения кода возникнет ошибка. Допустим, обобщенный метод `minmax()` реализуется следующим образом:

```
public static <T extends Comparable> T[] minmax(T... a)
{
    Object[] mm = new Object[2];
    . . .
    return (T[]) mm; // компилируется без предупреждения
}
```

Тогда приведенный ниже вызов данного метода компилируется без всяких предупреждений.

```
String[] ss = minmax("Tom", "Dick", "Harry");
```

Исключение типа `ClassCastException` возникает, когда ссылка на объект типа `Object[]` присваивается переменной типа `String[]`. В таком случае можно воспользоваться рефлексией и сделать вызов `Array.newInstance()` следующим образом:

```
public static <T extends Comparable> T[] minmax(T... a)
{
    T[] mm = (T[]) Array.newInstance(a.getClass().getComponentType(), 2);
    . . .
}
```

Методу `toArray()` из класса `ArrayList` повезло в меньшей степени. Ему нужно построить массив типа `T[]`, но у него нет типа элементов. Поэтому возможны два варианта:

```
Object[] toArray()
T[] toArray(T[] result)
```

Второй метод принимает параметр в виде массива. Если этот массив достаточно велик, он используется. В противном случае создается новый массив подходящего размера с элементами типа `result`.

Переменные типа в статическом контексте обобщенных классов недействительны

На переменные типа нельзя ссылаться в статических полях или методах. Например, следующая замечательная идея не сработает:

```
public class Singleton<T>
{
    private static T singleInstance; // ОШИБКА!

    public static T getSingleInstance() // ОШИБКА!
    {
        if (singleInstance == null) конструировать новый экземпляр типа T
            return singleInstance;
    }
}
```

Если бы такое было возможно, то программа должна была бы сконструировать экземпляр типа `Singleton<Random>` для общего генератора случайных чисел и экземпляр типа `Singleton <JFileChooser>` для общего диалогового окна выбора файлов. Но такая уловка не сработает. Ведь после стирания типов остается только один класс `Singleton` и только одно поле `singleInstance`. Именно по этой причине статические поля и методы с переменными типа просто объявлены как недопустимые.

Нельзя генерировать или перехватывать экземпляры обобщенного класса в виде исключений

Генерировать или перехватывать объекты обобщенного класса в виде исключений не допускается. На самом деле обобщенный класс не может расширять класс `Throwable`. Например, приведенное ниже определение обобщенного класса не будет скомпилировано.

```
public class Problem<T> extends Throwable
    { /* . . . */ } // ОШИБКА! Класс Throwable расширить нельзя
```

Кроме того, переменную типа нельзя использовать в блоке `catch`. Например, следующий метод не будет скомпилирован:

```
public static <T extends Throwable> void doWork(Class<T> t)
{
    try
    {
        выполнить нужные действия
    }
    catch (T e) // ОШИБКА! Перехватывать переменную типа нельзя
    {
        Logger.global.info(...)
    }
}
```

Но в то же время переменные типа можно использовать в определениях исключений. Так, приведенный ниже метод вполне допустим.

```
public static <T extends Throwable> void doWork(T t) throws T //
Допустимо!
{
    try
    {
        выполнить нужные действия
    }
    catch (Throwable realCause)
    {
        t.initCause(realCause);
        throw t;
    }
}
```

Преодоление ограничения на обработку проверяемых исключений

Как гласит основополагающий принцип обработки исключений в Java, для всех проверяемых исключений должны быть непременно предоставлены обработчики. Но это ограничение можно преодолеть, используя обобщения. И главным средством для достижения этой цели служит следующий метод:

```
@SuppressWarnings("unchecked")
public static <T extends Throwable> void throwAs(Throwable e) throws T
{
    throw (T) e;
}
```

Допустим, этот метод содержится в классе `Block`. Если сделать приведенный ниже вызов, то компилятор посчитает `t` непроверяемым исключением.

```
Block.<RuntimeException>throwAs(t);
```

В следующем фрагменте кода все исключения будут расценены компилятором как непроверяемые:

```
try
{
    ВЫПОЛНИТЬ НУЖНЫЕ ДЕЙСТВИЯ
}
catch (Throwable t)
{
    Block.<RuntimeException>throwAs(t);
}
```

Этот фрагмент кода можно ввести в абстрактный класс. Его пользователю достаточно переопределить тело данного метода, чтобы выполнить в нем нужные действия. А в результате вызова метода `toThread()` он может получить объект класса `Thread`, в методе `run()` которого игнорируются проверяемые исключения, как показано ниже.

```
public abstract class Block
{
    public abstract void body() throws Exception;
    public Thread toThread()
    {
        return new Thread()
        {
            public void run()
            {
                try
                {
                    body();
                }
                catch (Throwable t)
                {
                    Block.<RuntimeException>throwAs(t);
                }
            }
        };
    }
}

@SuppressWarnings("unchecked")
public static <T extends Throwable> void throwAs(Throwable e) throws T
{
    throw (T) e;
}
```

Например, в следующем примере кода выполняется поток, в котором генерируется проверяемое исключение.

```
public class Test
{
```

```
public static void main(String[] args)
{
    new Block()
    {
        public void body() throws Exception
        {
            Scanner in = new Scanner(new File("ququx"));
            while (in.hasNext())
                System.out.println(in.next());
        }
    }
    .toThread().start();
}
```

При выполнении этого фрагмента кода выводится результат трассировки стека с сообщением об исключении типа `FileNotFoundException`, если, конечно, не предоставить файл под именем `ququx`. Что же в этом такого особенного? Как правило, все исключения, возникающие в методе `run()` исполняемого потока, приходится перехватывать и заключать в оболочку непроверяемых исключений. Ведь метод `run()` объявляется без генерирования проверяемых исключений.

Но в данном примере эти исключения не заключаются в оболочку. Вместо этого просто генерируется исключение, которое компилятор не воспринимает как проверяемое. С помощью обобщенных классов, стирания типов и аннотации `@SuppressWarnings` в данном примере удалось преодолеть весьма существенное ограничение, накладываемое системой типов в Java.

Остерегайтесь конфликтов после стирания типов

Не допускается создавать условия, приводящие к конфликтам после стирания обобщенных типов. Допустим, в обобщенный класс `Pair` вводится метод `equals()` следующим образом:

```
public class Pair<T>
{
    public boolean equals(T value)
        { return first.equals(value) && second.equals(value); }
    . . .
}
```

Рассмотрим далее класс `Pair<String>`. По существу, в нем имеются два метода `equals()`:

```
boolean equals(String) // определен в обобщенном классе Pair<T>
boolean equals(Object) // унаследован от класса Object
```

Но интуиция приводит к заблуждению. В результате стирания типов метод `boolean equals(T)` становится методом `boolean equals(Object)`, который вступает в конфликт с методом `Object.equals()`. Разумеется, для разрешения подобного конфликта следует переименовать метод, из-за которого возникает конфликт.

В описании обобщений приводится другое правило: для поддержки преобразования путем стирания типов налагается ограничение, в соответствии с которым класс или переменная типа не могут одновременно быть подтипами двух типов, представляющих собой разные виды параметризации одного и того же интерфейса. Например, следующий код недопустим:

```
class Calendar implements Comparable<Calendar> { . . . }
class GregorianCalendar extends Calendar implements
    Comparable<GregorianCalendar> { . . . } // ОШИБКА!
```

В этом коде класс `GregorianCalendar` должен одновременно реализовать оба типа, `Comparable<Calendar>` и `Comparable<GregorianCalendar>`, представляющие собой разные виды параметризации одного и того же интерфейса. Непонятно, каким образом это ограничение согласуется со стиранием типов. Ведь вполне допустим следующий необобщенный вариант реализации интерфейса `Comparable`:

```
class Calendar implements Comparable { . . . }
class GregorianCalendar extends Calendar implements Comparable { . . . }
```

Причина недоразумения намного уточненнее, поскольку может случиться конфликт с синтезированными мостовыми методами. Класс, реализующий обобщенный интерфейс `Comparable<X>`, получает приведенный ниже мостовой метод. Но дело в том, что наличие двух таких методов с разными обобщенными типами `X` не допускается.

```
public int compareTo(Object other) { return compareTo((X) other); }
```

Правила наследования обобщенных типов

Для правильного обращения с обобщенными классами необходимо усвоить ряд правил, касающихся наследования и подтипов. Начнем с ситуации, которую многие программисты считают интуитивно понятной. Рассмотрим в качестве примера класс `Employee` и подкласс `Manager`. Является ли обобщенный класс `Pair<Manager>` подклассом, производным от обобщенного класса `Pair<Employee>`? Как ни странно, не является. Например, следующий код не подлежит компиляции:

```
Manager[] topHonchos = . . . ;
Pair<Employee> result = ArrayAlg.minmax(topHonchos); // ОШИБКА!
```

Метод `minmax()` возвращает объект типа `Pair<Manager>`, но не тип `Pair<Employee>`, а присваивать их друг другу недопустимо. В общем случае между классами `Pair<S>` и `Pair<T>` нет никаких отношений наследования, как бы ни соотносились друг с другом обобщенные типы `S` и `T` (рис. 12.1).

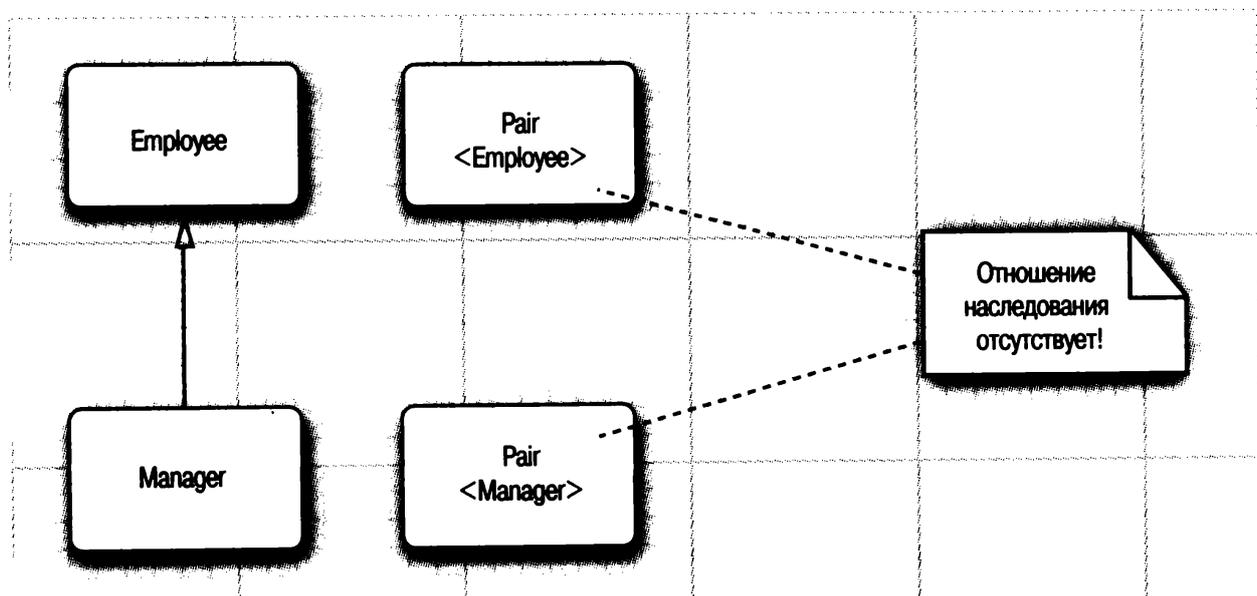


Рис. 12.1. Отношение наследования между обобщенными классами `Pair` отсутствует

Такое ограничение может показаться слишком строгим, но оно необходимо для соблюдения типовой безопасности. Предположим, что объект класса `Pair<Manager>` все-таки разрешается преобразовать в объект класса `Pair<Employee>`, как показано в приведенном ниже фрагменте кода.

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair<Employee> employeeBuddies =
    managerBuddies; // недопустимо, но предположим, что разрешено
employeeBuddies.setFirst(lowlyEmployee);
```

Даже если последний оператор и допустим, переменные `employeeBuddies` и `managerBuddies` все равно ссылаются на *один и тот же объект*. В итоге получается, что высшее руководство организации приравнивается к рядовым сотрудникам, что для класса `Pair<Manager>` невозможно.



НА ЗАМЕТКУ! Выше было проведено очень важное отличие между обобщенными типами и массивами в Java. Так, массив `Manager[]` можно присвоить переменной типа `Employee[]` следующим образом:

```
Manager[] managerBuddies = { ceo, cfo };
Employee[] employeeBuddies = managerBuddies; // Допустимо!
```

Но массивы снабжены дополнительной защитой. Если попытаться сохранить объект рядового сотрудника в элементе массива `employeeBuddies[0]`, то виртуальная машина сгенерирует исключение типа `ArrayStoreException`.

Параметризованный тип можно всегда преобразовать в базовый тип. Например, `Pair<Employee>` — это подтип базового типа `Pair`. Такое преобразование необходимо для взаимодействия с унаследованным кодом. А можно ли преобразовать базовый тип и тем самым вызвать ошибку соблюдения типов? К сожалению, да. Рассмотрим следующий пример кода:

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair rawBuddies = managerBuddies; // Допустимо!
rawBuddies.setFirst(new File(". . .")); // допускается, но только
// с предупреждением во время компиляции
```

Такой код, откровенно говоря, настораживает. Однако в этом отношении ситуация оказывается не хуже, чем в прежних версиях Java. Защита виртуальной машины не безгранична. Когда внешний объект извлекается методом `getFirst()` и присваивается переменной типа `Manager`, генерируется исключение типа `ClassCastException`, как и в старые добрые времена. Но в этом случае код лишается дополнительной защиты, которую обычно предоставляет обобщенное программирование.

И, наконец, одни обобщенные классы могут расширять или реализовывать другие обобщенные классы. В этом отношении они ничем не отличаются от обычных классов. Например, обобщенный класс `ArrayList<T>` реализует обобщенный интерфейс `List<T>`. Это означает, что объект типа `ArrayList<Manager>` может быть преобразован в объект типа `List<Manager>`. Но как было показано выше, объект типа `ArrayList<Manager>` — это *не* объект типа `ArrayList<Employee>` или `List<Employee>`. На рис. 12.2 схематически показаны все отношения наследования между этими обобщенными классами и интерфейсами.

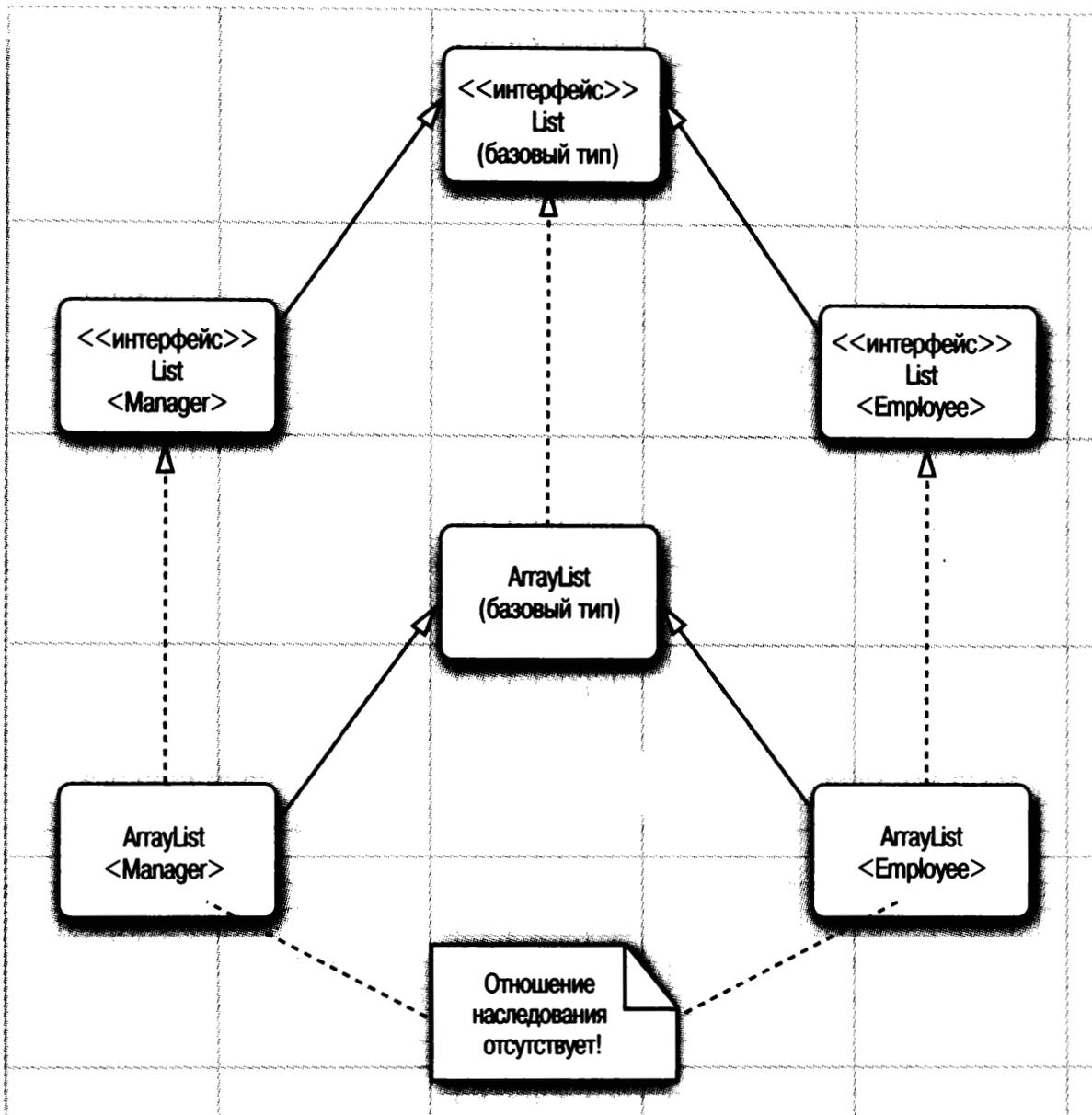


Рис. 12.2. Отношения наследования подтипов между обобщенными типами списочных массивов

Подстановочные типы

Исследователям систем типов уже давно известно, что жесткие системы обобщенных типов использовать довольно неприятно. Поэтому создатели Java придумали изящный (и тем не менее безопасный) выход из положения: *подстановочные типы*. Например, следующий подстановочный тип обозначает любой обобщенный тип `Pair`, параметр типа которого обозначает подкласс, производный от класса `Employee`, в частности, класс `Pair<Manager>`, но не класс `Pair<String>`.

```
Pair<? extends Employee>
```

Допустим, требуется написать следующий метод, который выводит пары сотрудников:

```
public static void printBuddies(Pair<Employee> p)
{
    Employee first = p.getFirst();
    Employee second = p.getSecond();
}
```

```

System.out.println(first.getName() + " and " +
    second.getName() + " are buddies.");
}

```

Как было показано в предыдущем разделе, передать объект типа `Pair<Manager>` этому методу нельзя, что не совсем удобно. Но из этого положения имеется простой выход — использовать подстановочный тип следующим образом:

```
public static void printBuddies(Pair<? extends Employee> p)
```

Тип `Pair<Manager>` является подтипом `Pair<? extends Employee>` (рис. 12.3).

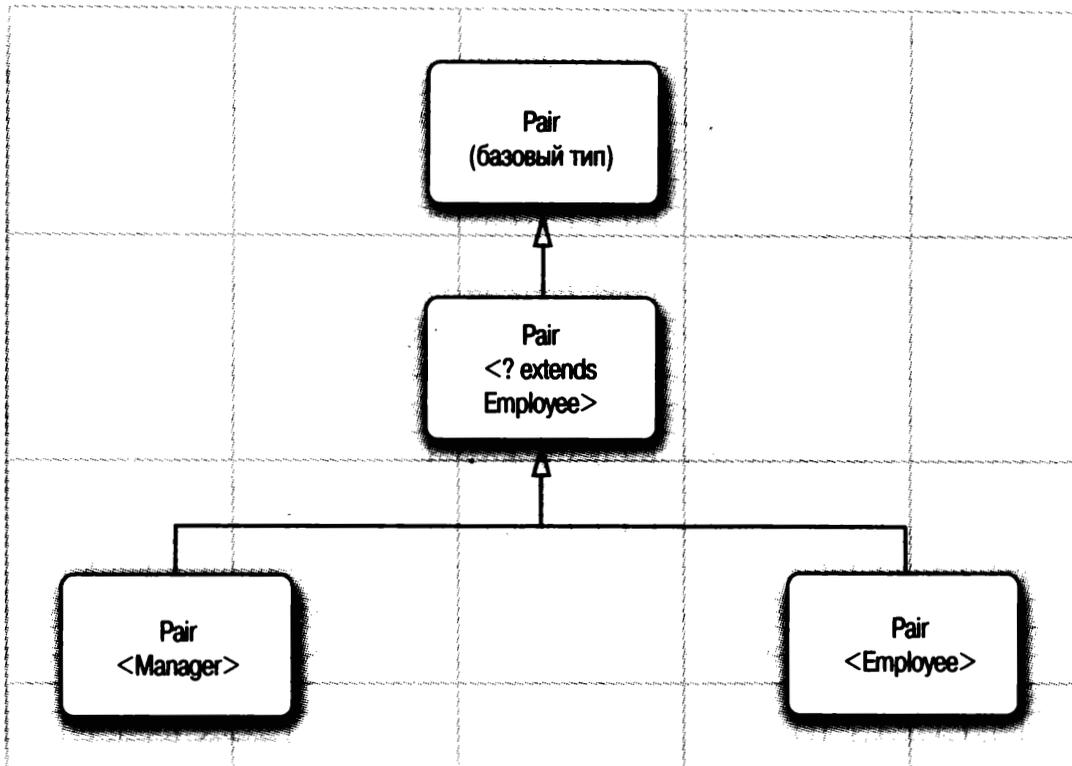


Рис. 12.3. Отношения наследования подтипов с подстановками

А могут ли подстановки нарушить тип `Pair<Manager>` по ссылке `Pair<? extends Employee>`? Не могут, как показано в приведенном ниже фрагменте кода.

```

Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies; // Допустимо!
wildcardBuddies.setFirst(lowlyEmployee); // Ошибка при компиляции

```

Но в то же время вызов метода `setFirst()` приведет к ошибке соблюдения типов. Чтобы стала понятнее причина этой ошибки, рассмотрим тщательнее обобщенный класс типа `Pair<? extends Employee>`. У него имеются следующие методы:

```

? extends Employee getFirst()
void setFirst(? extends Employee)

```

Это делает невозможным вызов метода `setFirst()`. Компилятору требуется какой-нибудь подтип `Employee`, но неизвестно, какой именно. Он отказывается передать любой конкретный тип, поскольку знак подстановки `?` может и не совпасть с этим типом. При вызове метода `getFirst()` такого затруднения не возникает. Значение, возвращаемое методом `getFirst()`, вполне допустимо присвоить переменной ссылки на объект типа `Employee`. В этом, собственно, и состоит главный смысл ограниченных подстановок. С их помощью можно теперь различать безопасные методы доступа от небезопасных модифицирующих методов.

Ограничения супертипа на подстановки

Ограничения на подстановки подобны ограничениям на переменные типа. Но у них имеется дополнительная возможность — определить *ограничение супертипа* следующим образом:

```
? super Manager
```

Эта подстановка ограничивается всеми супертипами `Manager`. (Просто удивительно, насколько удачно имеющееся в Java ключевое слово `super` так точно описывает подобное отношение наследования.) А зачем это может понадобиться? Подстановка с ограничением супертипа дает поведение, противоположное тому, что сказано выше о подстановочных типах. В частности, методам можно передавать параметры, но нельзя использовать возвращаемые ими значения. Например, в классе типа `Pair<? super Manager>` имеются следующие методы:

```
void setFirst(? super Manager)
? super Manager getFirst()
```

Компилятору точно неизвестен тип метода `setFirst()`, тем не менее он может вызвать этот метод с любым объектом типа `Manager`, `Employee` или `Object`, но не такого подтипа, как `Executive`. Более того, если вызывается метод `getFirst()`, то нет никакой гарантии относительно типа возвращаемого объекта. Его можно присвоить только переменной ссылки на объект типа `Object`.

Рассмотрим типичный пример. Допустим, имеется массив руководящих работников организации и сведения о минимальном и максимальном размере премии одного из них требуется ввести в объект класса `Pair`. К какому же типу относится класс `Pair`? Будет справедливо, если это окажется тип `Pair<Employee>` или же `Pair<Object>` (рис. 12.4). Следующий метод примет любой подходящий объект типа `Pair` в качестве своего параметра:

```
public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
    if (a == null || a.length == 0) return;
    Manager min = a[0];
    Manager max = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (min.getBonus() > a[i].getBonus()) min = a[i];
        if (max.getBonus() < a[i].getBonus()) max = a[i];
    }
    result.setFirst(min);
    result.setSecond(max);
}
```

Если рассуждать интуитивно, то подстановки с ограничениями супертипа позволяют записывать данные в обобщенный объект, а подстановки с ограничениями подтипа — читать данные из обобщенного объекта. Рассмотрим другой пример наложения ограничений супертипа. Интерфейс `Comparable` сам является обобщенным. Он объявляется следующим образом:

```
public interface Comparable<T>
{
    public int compareTo(T other);
}
```

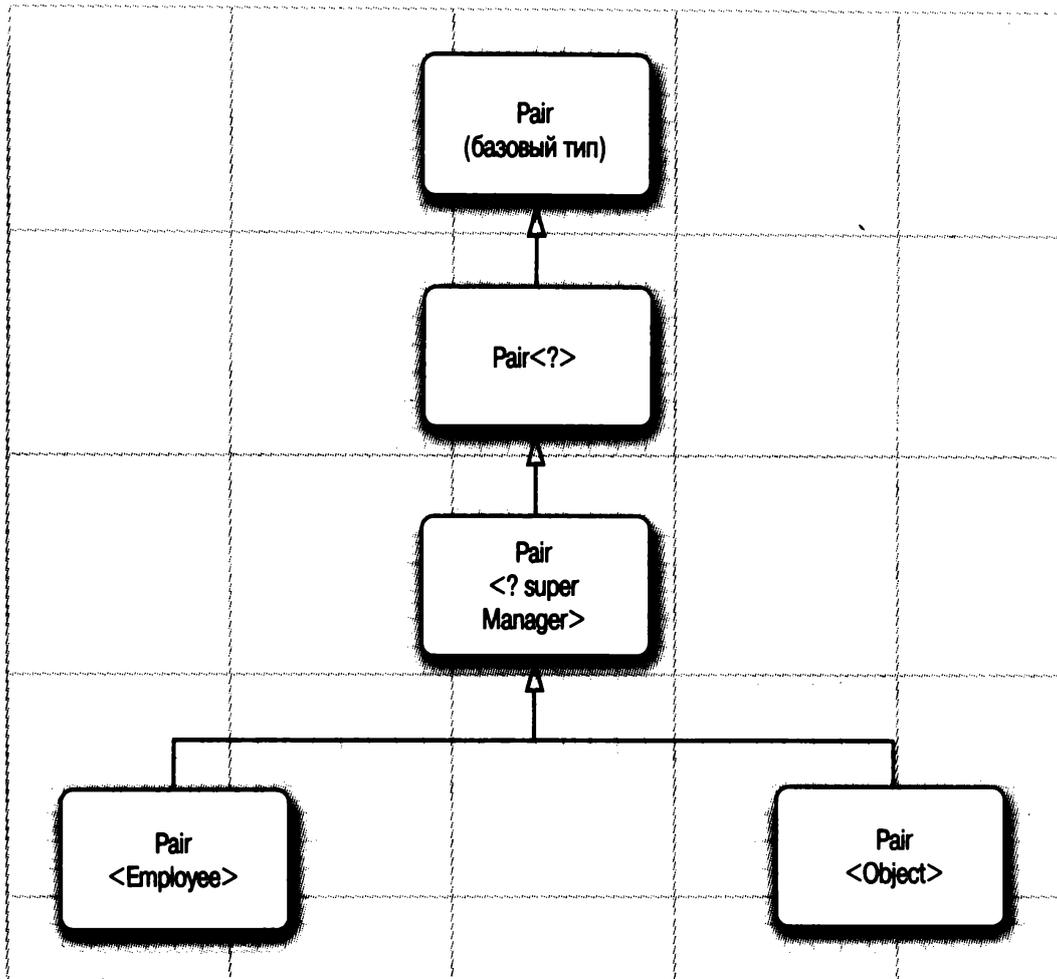


Рис. 12.4. Подстановка с ограничением супертипа

Здесь переменная типа обозначает тип параметра `other`. Например, класс `String` реализует интерфейс `Comparable<String>`, а его метод `compareTo()` объявляется следующим образом:

```
public int compareTo(String other)
```

И это хорошо. Ведь явно задаваемый параметр имеет правильный тип. До появления обобщений параметр `other` относился к типу `Object`, и поэтому в реализации данного метода требовалось приведение типов. А теперь интерфейс `Comparable` относится к обобщенному типу, что позволяет внести следующее усовершенствование в объявление метода `min()` в классе `ArrayList`:

```
public static <T extends Comparable<T>> T min(T[] a)
```

Такое объявление метода оказывается более основательным, чем простое расширение типа `T extends Comparable`, и должно вполне подходить для многих классов. Так, если определяется минимальное значение в массиве типа `String`, то обобщенный тип `T` становится типом `String`, а тот — подтипом `Comparable<String>`. Но при обработке массива объектов типа `GregorianCalendar` возникает затруднение. Оказывается, что `GregorianCalendar` — это подкласс, производный от класса `Calendar`, который реализует интерфейс `Comparable<Calendar>`. Поэтому и класс `GregorianCalendar` реализует интерфейс `Comparable<Calendar>`, а не интерфейс `Comparable<GregorianCalendar>`. В подобных случаях на помощь приходят супер-типы, как показано ниже.

```
public static <T extends Comparable<? super T>> T min(T[] a) . . .
```

Теперь метод `compareTo()` имеет следующую форму:

```
int compareTo(? super T)
```

Возможно, он объявляется для принятия в качестве параметра объекта обобщенного типа `T` или даже супертипа `T`, если, например, `T` обозначает тип `GregorianCalendar`. Как бы то ни было, методу `compareTo()` можно безопасно передать объект обобщенного типа `T`.

Непосвященных объявление вроде `<T extends Comparable<? super T>>` может повергнуть в невольный трепет. Это прискорбно, потому что такое объявление призвано помочь прикладным программистам, исключая ненужные ограничения на параметры вызова. Прикладные программисты, не интересующиеся обобщениями, вероятно, предпочтут поскорее пропустить такие объявления и принять на веру, что разработчики библиотеки знали, что делали. А тем, кто занимается разработкой библиотек, придется обратиться к подстановкам, иначе пользователи их библиотек помянут их недобрым словом и вынуждены будут прибегать в своем коде к разного рода приведению типов до тех пор, пока он не скомпилируется без ошибок.

Неограниченные подстановки

Подстановками можно пользоваться вообще без ограничений, например `Pair<?>`. На первый взгляд, этот тип похож на базовый тип `Pair`. На самом же деле типы отличаются. В классе типа `Pair<?>` имеются методы, аналогичные приведенным ниже.

```
? getFirst()
void setFirst(?)
```

Значение, возвращаемое методом `getFirst()`, может быть присвоено только переменной ссылки на объект типа `Object`. А метод `setFirst()` вообще нельзя вызвать — даже с параметром типа `Object`. Существенное отличие типов `Pair<?>` и `Pair` в том и состоит, что метод `setObject()` из класса базового типа `Pair` нельзя вызвать с *любым* объектом типа `Object`.



НА ЗАМЕТКУ! Тем не менее можно сделать вызов `setFirst(null)`.

Но зачем вообще может понадобиться такой непонятный тип? Оказывается, он удобен для выполнения очень простых операций. Например, в приведенном ниже методе проверяется, содержит ли пара пустую ссылку на объект. Такому методу вообще не требуется конкретный тип.

```
public static boolean hasNulls(Pair<?> p)
{
    return p.getFirst() == null || p.getSecond() == null;
}
```

Применения подстановочного типа можно было бы избежать, превратив `contains()` в обобщенный метод, как показано ниже. Но его вариант с подстановочным типом выглядит более удобочитаемым.

```
public static <T> boolean hasNulls(Pair<T> p)
```

Захват подстановок

Рассмотрим следующий метод, меняющий местами составные элементы пары:

```
public static void swap(Pair<?> p)
```

Подстановка не является переменной типа, поэтому знак ? нельзя указывать вместо типа. Иными словами, следующий фрагмент кода оказывается написанным неверно:

```
? t = p.getFirst(); // ОШИБКА!  
p.setFirst(p.getSecond());  
p.setSecond(t);
```

В связи с этим возникает затруднение, поскольку при перестановке нужно временно запоминать первый составной элемент пары. Правда, это затруднение можно разрешить довольно интересным способом, написав вспомогательный метод `swapHelper()` так, как показано ниже.

```
public static <T> void swapHelper(Pair<T> p)  
{  
    T t = p.getFirst();  
    p.setFirst(p.getSecond());  
    p.setSecond(t);  
}
```

Следует, однако, иметь в виду, что метод `swapHelper()` — обобщенный, тогда как метод `swap()` — необобщенный. У него имеется фиксированный параметр типа `Pair<?>`. Теперь метод `swapHelper()` можно вызвать из метода `swap()` следующим образом:

```
public static void swap(Pair<?> p) { swapHelper(p); }
```

В данном случае параметр обобщенного типа `T` *захватывает* подстановку во вспомогательном методе `swapHelper()`. Неизвестно, какой именно тип обозначает подстановка, но это совершенно определенный тип. Поэтому определение `<T>swapHelper()` имеет конкретный смысл, когда обобщение `T` обозначает этот тип.

Безусловно, пользоваться в данном случае подстановкой совсем не обязательно. Вместо этого можно было бы напрямую реализовать `<T> void swap(Pair<T> p)` как обобщенный метод без подстановок. Но рассмотрим следующий пример, в котором подстановочный тип появляется естественным образом посреди вычислений:

```
public static void maxminBonus(Manager[] a, Pair<? super Manager> result)  
{  
    minmaxBonus(a, result);  
    PairAlg.swap(result); // Допустимо, так как в методе swapHelper()  
                        // захватывается подстановочный тип  
}
```

В данном случае обойтись без механизма захвата подстановки нельзя. Захват подстановки допускается в очень ограниченных случаях. Компилятор должен иметь возможность гарантировать, что подстановка представляет один, определенный тип. Например, обобщение `T` в объявлении класса `ArrayList<Pair<T>>` вообще не сможет захватить подстановку в объявлении класса `ArrayList<Pair<?>>`. Списочный массив может содержать два элемента типа `Pair<?>`, в каждом из которых вместо знака ? будет подставлен свой тип.

Пример тестовой программы из листинга 12.3 подытоживает разнообразные способы обращения с обобщениями, рассмотренные в предыдущих разделах, чтобы продемонстрировать их непосредственно в коде.

Листинг 12.3. Исходный код из файла pair3/PairTest3.java

```
1 package pair3;
2
3 /**
4  * @version 1.01 2012-01-26
5  * @author Cay Horstmann
6  */
7 public class PairTest3
8 {
9     public static void main(String[] args)
10    {
11        Manager ceo = new Manager("Gus Greedy", 800000, 2003, 12, 15);
12        Manager cfo = new Manager("Sid Sneaky", 600000, 2003, 12, 15);
13        Pair<Manager> buddies = new Pair<>(ceo, cfo);
14        printBuddies(buddies);
15
16        ceo.setBonus(1000000);
17        cfo.setBonus(500000);
18        Manager[] managers = { ceo, cfo };
19
20        Pair<Employee> result = new Pair<>();
21        minmaxBonus(managers, result);
22        System.out.println("first: " + result.getFirst().getName()
23            + ", second: " + result.getSecond().getName());
24        maxminBonus(managers, result);
25        System.out.println("first: " + result.getFirst().getName()
26            + ", second: " + result.getSecond().getName());
27    }
28
29    public static void printBuddies(Pair<? extends Employee> p)
30    {
31        Employee first = p.getFirst();
32        Employee second = p.getSecond();
33        System.out.println(first.getName() + " and " + second.getName() +
34            " are buddies.");
35    }
36
37    public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
38    {
39        if (a == null || a.length == 0) return;
40        Manager min = a[0];
41        Manager max = a[0];
42        for (int i = 1; i < a.length; i++)
43        {
44            if (min.getBonus() > a[i].getBonus()) min = a[i];
45            if (max.getBonus() < a[i].getBonus()) max = a[i];
46        }
47        result.setFirst(min);
48        result.setSecond(max);
49    }
50    public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
51    {
```

```
52     minmaxBonus(a, result);
53     PairAlg.swapHelper(result); // в методе swapHelper()
54                                 // захватывается подстановочный тип
55 }
56 }
57
58 class PairAlg
59 {
60     public static boolean hasNulls(Pair<?> p)
61     {
62         return p.getFirst() == null || p.getSecond() == null;
63     }
64
65     public static void swap(Pair<?> p) { swapHelper(p); }
66
67     public static <T> void swapHelper(Pair<T> p)
68     {
69         T t = p.getFirst();
70         p.setFirst(p.getSecond());
71         p.setSecond(t);
72     }
73 }
```

Рефлексия и обобщения

Класс `Class` теперь является обобщенным. Например, `String.class` — на самом деле объект (по существу, единственный) класса `Class<String>`. Параметр типа удобен, потому что он позволяет методам обобщенного класса `Class<T>` быть более точными в отношении возвращаемых типов. В следующих методах из класса `Class<T>` используются преимущества параметра типа:

```
newInstance()
T cast(Object obj)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor<T> getDeclaredConstructor(Class... parameterTypes)
```

Метод `newInstance()` возвращает экземпляр класса, полученный от конструктора по умолчанию. Его возвращаемый тип теперь может быть объявлен как `T`, т.е. тот же тип, что и у класса, описанного в обобщении `Class<T>`. Такой прием позволяет экономить на приведении типов.

Метод `cast()` возвращает данный объект, объявленный теперь как имеющий обобщенный тип `T`, если его тип действительно является подтипом `T`. В противном случае он генерирует исключение типа `BadCastException`.

Метод `getEnumConstants()` возвращает пустое значение `null`, если данный класс не относится к типу перечислений (`enum`) или массивов перечислимых значений, о которых известно, что они принадлежат к обобщенному типу `T`.

И, наконец, методы `getConstructor()` и `getDeclaredConstructor()` возвращают объект обобщенного класса `Constructor<T>`. Класс `Constructor` также стал обобщенным, поэтому у метода `newInstance()` теперь имеется корректный возвращаемый тип.

java.lang.Class<T> 1.0

- `T newInstance()` 5.0
Возвращает новый экземпляр, созданный конструктором по умолчанию.
- `T cast(Object obj)` 5.0
Возвращает объект `obj`, если он пустой (`null`) или может быть преобразован в обобщенный тип `T`, а иначе генерирует исключение типа `BadCastException`.
- `T[] getEnumConstants()` 5.0
Возвращает массив всех значений, если `T` — перечислимый тип, в противном случае — пустое значение `null`.
- `Class<? super T> getSuperclass()` 5.0
Возвращает суперкласс данного класса или пустое значение `null`, если обобщенный тип `T` не обозначает класс или же обозначает класс `Object`.
- `Constructor<T> getConstructor(Class... parameterTypes)` 5.0
- `Constructor<T> getDeclaredConstructor(Class... parameterTypes)` 5.0
Получают открытый конструктор или же конструктор с заданными типами параметров.

java.lang.reflect.Constructor<T> 1.1

- `T newInstance(Object... parameters)` 5.0
Возвращает новый экземпляр, созданный конструктором с заданными параметрами.

Сопоставление типов с помощью параметров Class<T>

Иногда оказывается полезно сопоставить переменную типа параметра `Class<T>` в обобщенном методе. Ниже приведен характерный тому пример.

```
public static <T> Pair<T> makePair(Class<T> c)
    throws InstantiationException, IllegalAccessException
{
    return new Pair<>(c.newInstance(), c.newInstance());
}
```

Если сделать приведенный ниже вызов, то параметр `Employee.class` будет означать объект типа `Class<Employee>`. Параметр обобщенного типа `T` в методе `makePair()` сопоставляется с классом `Employee`, откуда компилятор может заключить, что данный метод возвращает объект типа `Pair<Employee>`.

```
makePair(Employee.class)
```

Сведения об обобщенных типах в виртуальной машине

Одной из замечательных особенностей обобщений в Java является стирание обобщенных типов в виртуальной машине. Как ни странно, но классы, подвергшиеся стиранию типов, все еще сохраняют некоторую память о своем обобщенном

происхождении. Например, базовому классу `Pair` известно, что он происходит от обобщенного класса `Pair<T>`, несмотря на то, что объект типа `Pair` не может отличить, был он сконструирован как объект типа `Pair<String>` или же как объект типа `Pair<Employee>`.

Аналогично метод

```
public static Comparable min(Comparable[] a)
```

является результатом стирания типов в приведенном ниже обобщенном методе.

```
public static <T extends Comparable<? super T>> T min(T[] a)
```

Прикладной интерфейс API для рефлексии можно использовать, чтобы определить следующее.

- Имеет ли обобщенный метод параметр типа `T`.
- Имеет ли параметр типа ограниченный подтип, который сам является обобщенным.
- Имеет ли ограничивающий тип подставляемый параметр.
- Имеет ли подставляемый параметр ограниченный суперттип.
- Имеет ли обобщенный метод обобщенный массив в качестве своего параметра.

Иными словами, с помощью рефлексии можно реконструировать все, что было объявлено в реализации обобщенных классов и методов. Но вряд ли можно узнать, каким образом разрешались параметры типа для конкретных объектов и вызовов методов.



НА ЗАМЕТКУ! Сведения о типе, содержащиеся в файлах классов для осуществления рефлексии обобщений, несовместимы с устаревшими виртуальными машинами.

Для того чтобы выразить объявления обобщенных типов, следует воспользоваться интерфейсом `Type`. У этого интерфейса имеются следующие подтипы.

- Класс `Class`, описывающий конкретные типы.
- Интерфейс `TypeVariable`, описывающий переменные типа (как, например, `T extends Comparable<? super T>`).
- Интерфейс `WildcardType`, описывающий подстановки (как, например, `? super T`).
- Интерфейс `ParameterizedType`, описывающий обобщенный класс или типы интерфейсов (как, например, `Comparable<? super T>`).
- Интерфейс `GenericArrayType`, описывающий обобщенные массивы (как, например, `T[]`).

На рис. 12.5 схематически показана иерархия наследования интерфейса `Type`. Обратите внимание на то, что последние четыре подтипа являются интерфейсами. Виртуальная машина создает экземпляры подходящих классов, реализующих эти интерфейсы.

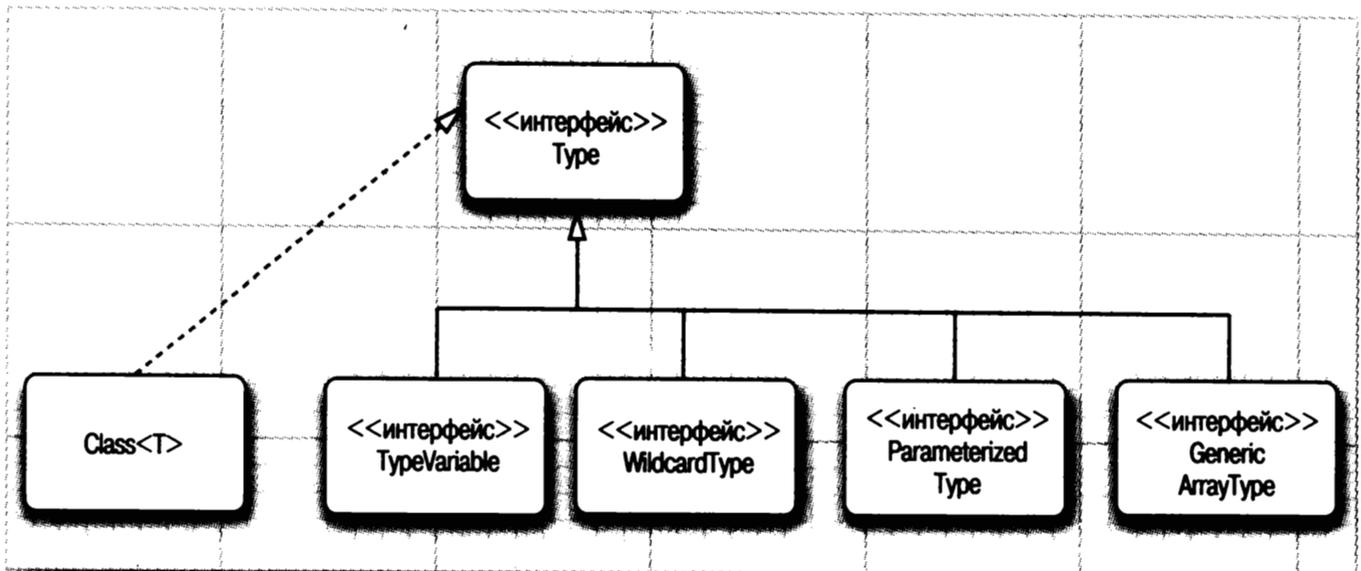


Рис. 12.5. Иерархия наследования интерфейса Type

В исходном коде примера программы из листинга 12.4 прикладной интерфейс API рефлексии обобщений применяется для вывода сведений о данном классе. Если выполнить эту программу вместе с классом `Pair`, то в результате будет получен следующий отчет:

```

class Pair<T> extends java.lang.Object
public T getFirst()
public T getSecond()
public void setFirst(T)
public void setSecond(T)
  
```

А если выполнить эту программу вместе с классом `ArrayAlg` в каталоге `PairTest2`, то в отчете появится следующий метод:

```

public static <T extends java.lang.Comparable> Pair<T> minmax(T[])
  
```

В приведенном далее кратком описании прикладного интерфейса API для рефлексии обобщений представлены методы, используемые в примере программы из листинга 12.4.

Листинг 12.4. Исходный код из файла `genericReflection/GenericReflectionTest.java`

```

1  package genericReflection;
2
3  import java.lang.reflect.*;
4  import java.util.*;
5  /**
6   * @version 1.10 2007-05-15
7   * @author Cay Horstmann
8   */
9  public class GenericReflectionTest
10 {
11     public static void main(String[] args)
12     {
13         // прочитать имя класса из аргументов командной строки
14         // или введенных пользователем данных
15         String name;
16         if (args.length > 0) name = args[0];
  
```

```
17     else
18     {
19         Scanner in = new Scanner(System.in);
20         System.out.println(
21             "Enter class name (e.g. java.util.Collections): ");
22         name = in.next();
23     }
24
25     try
26     {
27         // вывести обобщенные сведения о классе и его открытых методах
28         Class<?> cl = Class.forName(name);
29         printClass(cl);
30         for (Method m : cl.getDeclaredMethods())
31             printMethod(m);
32     }
33     catch (ClassNotFoundException e)
34     {
35         e.printStackTrace();
36     }
37 }
38
39 public static void printClass(Class<?> cl)
40 {
41     System.out.print(cl);
42     printTypes(cl.getTypeParameters(), "<", " ", ">", true);
43     Type sc = cl.getGenericSuperclass();
44     if (sc != null)
45     {
46         System.out.print(" extends ");
47         printType(sc, false);
48     }
49     printTypes(cl.getGenericInterfaces(),
50         " implements ", " ", " ", " ", false);
51     System.out.println();
52 }
53 public static void printMethod(Method m)
54 {
55     String name = m.getName();
56     System.out.print(Modifier.toString(m.getModifiers()));
57     System.out.print(" ");
58     printTypes(m.getTypeParameters(), "<", " ", "> ", true);
59
60     printType(m.getGenericReturnType(), false);
61     System.out.print(" ");
62     System.out.print(name);
63     System.out.print("(");
64     printTypes(m.getGenericParameterTypes(), " ", " ", " ", false);
65     System.out.println(")");
66 }
67
68 public static void printTypes(Type[] types, String pre,
69     String sep, String suf, boolean isDefinition)
70 {
71     if (pre.equals(" extends ") && Arrays.equals(types, new Type[]
72         { Object.class })) return;
```

```
73     if (types.length > 0) System.out.print(pre);
74     for (int i = 0; i < types.length; i++)
75     {
76         if (i > 0) System.out.print(sep);
77         printType(types[i], isDefinition);
78     }
79     if (types.length > 0) System.out.print(suf);
80 }
81
82 public static void printType(Type type, boolean isDefinition)
83 {
84     if (type instanceof Class)
85     {
86         Class<?> t = (Class<?>) type;
87         System.out.print(t.getName());
88     }
89     else if (type instanceof TypeVariable)
90     {
91         TypeVariable<?> t = (TypeVariable<?>) type;
92         System.out.print(t.getName());
93         if (isDefinition)
94             printTypes(t.getBounds(), " extends ", " & ", "", false);
94     }
95     else if (type instanceof WildcardType)
96     {
97         WildcardType t = (WildcardType) type;
98         System.out.print("?");
99         printTypes(t.getUpperBounds(), " extends ", " & ", "", false);
100        printTypes(t.getLowerBounds(), " super ", " & ", "", false);
101    }
102    else if (type instanceof ParameterizedType)
103    {
104        ParameterizedType t = (ParameterizedType) type;
105        Type owner = t.getOwnerType();
106        if (owner != null)
107        {
108            printType(owner, false);
109            System.out.print(".");
110        }
111        printType(t.getRawType(), false);
112        printTypes(t.getActualTypeArguments(), "<", ", ", ">", false);
113    }
114    else if (type instanceof GenericArrayType)
115    {
116        GenericArrayType t = (GenericArrayType) type;
117        System.out.print("");
118        printType(t.getGenericComponentType(), isDefinition);
119        System.out.print("[]");
120    }
121 }
122 }
```

java.lang.Class<T> 1.0

- `TypeVariable[] getTypeParameters()` 5.0
Получает переменные обобщенного типа, если этот тип был объявлен как обобщенный, а иначе — массив нулевой длины.
- `Type getGenericSuperclass()` 5.0
Получает обобщенный тип суперкласса, который был объявлен для этого типа, или же пустое значение `null`, если это тип `Object` или вообще не тип класса.
- `Type[] getGenericInterfaces()` 5.0
Получает обобщенные типы интерфейсов, которые были объявлены для этого типа, в порядке объявления, или массив нулевой длины, если данный тип не реализует интерфейсы.

java.lang.reflect.Method 1.1

- `TypeVariable[] getTypeParameters()` 5.0
Получает переменные обобщенного типа, если этот тип был объявлен как обобщенный, а иначе — массив нулевой длины.
- `Type getGenericReturnType()` 5.0
Получает обобщенный возвращаемый тип, с которым был объявлен данный метод.
- `Type[] getGenericParameterTypes()` 5.0
Получает обобщенные типы параметров, с которыми был объявлен данный метод. Если у метода отсутствуют параметры, возвращается массив нулевой длины.

java.lang.reflect.TypeVariable 5.0

- `String getName()`
Получает имя переменной типа.
- `Type[] getBounds()`
Получает ограничения на подкласс для данной переменной типа или массив нулевой длины, если ограничения на переменную отсутствуют.

java.lang.reflect.WildcardType 5.0

- `Type[] getUpperBounds()`
Получает для данной переменной типа ограничения на подкласс, определяемые оператором `extends`, или массив нулевой длины, если ограничения на подкласс отсутствуют.
- `Type[] getLowerBounds()`
Получает для данной переменной типа ограничения на суперкласс, определяемые оператором `super`, или массив нулевой длины, если ограничения на суперкласс отсутствуют.

java.lang.reflect.ParameterizedType 5.0

- `Type getRawType()`
Получает базовый тип для данного параметризованного типа.
- `Type[] getActualTypeArguments()`
Получает параметр типа, с которым был объявлен данный параметризованный тип.
- `Type getOwnerType()`
Получает тип внешнего класса, если данный тип — внутренний, или же пустое значение `null`, если это тип верхнего уровня.

java.lang.reflect.GenericArrayType 5.0

- `Type getGenericComponentType()`
Получает обобщенный тип компонента, с которым связан тип данного массива.

Теперь вы знаете, как пользоваться обобщенными классами и как программировать собственные обобщенные классы и методы, если возникнет такая потребность. Не менее важно и то, что вы знаете, как трактовать объявления обобщенных типов, которые вы можете встретить в документации на прикладной интерфейс API и в сообщениях об ошибках. А для того чтобы найти исчерпывающие ответы на вопросы, которые могут возникнуть у вас в связи с применением на практике механизма обобщений в Java, обратитесь к замечательному списку часто (и не очень часто) задаваемых вопросов по адресу <http://angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>. В следующей главе будет показано, каким образом механизм обобщений применяется в архитектуре коллекций в Java.

Коллекции

В этой главе...

- ▶ Интерфейсы коллекций
- ▶ Конкретные коллекции
- ▶ Архитектура коллекций
- ▶ Алгоритмы
- ▶ Унаследованные коллекции

Выбираемые структуры данных могут заметно отличаться как в отношении реализации методов в “естественном стиле”, так и в отношении производительности. При этом возникают следующие вопросы. Требуется ли быстрый поиск среди тысяч (или даже миллионов) отсортированных элементов? Нужен ли быстрый ввод и удаление элементов в середине упорядоченной последовательности? Требуется ли устанавливать связи между ключами и значениями?

В этой главе будет показано, как реализовать средствами библиотеки Java традиционные структуры данных, без которых немислимо серьезное программирование. На специальностях, связанных с вычислительной техникой, как правило, предусмотрен курс по структурам данных, поэтому имеется немало литературы по этому важному предмету. Наше изложение отличается от такого курса. В частности, мы опустим теорию и перейдем непосредственно к практике, поясняя на конкретных примерах, каким образом классы коллекций из стандартной библиотеки употребляются в коде.

Интерфейсы коллекций

В первоначальной версии Java предлагался лишь небольшой набор классов для наиболее употребительных структур данных: `Vector`, `Stack`, `Hashtable`, `BitSet`, а также интерфейс `Enumeration`, предоставляющий абстрактный механизм для

обращения к элементам, находящимся в произвольном контейнере. Безусловно, это было мудрое решение, ведь для реализации всеобъемлющей библиотеки классов коллекций требуется время и опыт.

После выхода версии Java SE 1.2 разработчики осознали, что настало время создать полноценный набор структур данных. При этом они столкнулись с множеством противоречивых требований. Они стремились к тому, чтобы библиотека была компактной и простой в освоении. Для этого нужно было избежать сложности стандартной библиотеки шаблонов (STL) в C++, но в то же время позаимствовать “обобщенные алгоритмы”, впервые появившиеся в STL. Кроме того, нужно было обеспечить совместимость унаследованных классов коллекций с новой архитектурой. И как это случается со всеми разработчиками библиотек коллекций, им приходилось не раз делать нелегкий выбор, находя попутно немало оригинальных решений. В этом разделе мы рассмотрим основную конструкцию архитектуры коллекций в Java, покажем, как применять их на практике, а также разъясним наиболее противоречивые их особенности.

Разделение интерфейсов и реализаций коллекций

Как это принято в современных библиотеках структур данных, в рассматриваемой здесь библиотеке коллекций Java *интерфейсы* и *реализации* разделены. Рассмотрим, как это разделение происходит на примере хорошо известной структуры данных — *очереди*.

Интерфейс очереди определяет, что элементы можно добавлять в хвосте очереди, удалять их в ее голове, а также выяснять, сколько элементов находится в очереди в данный момент. Очереди применяются в тех случаях, когда требуется накапливать объекты и извлекать их по принципу “первым пришел — первым обслужен” (рис. 13.1).

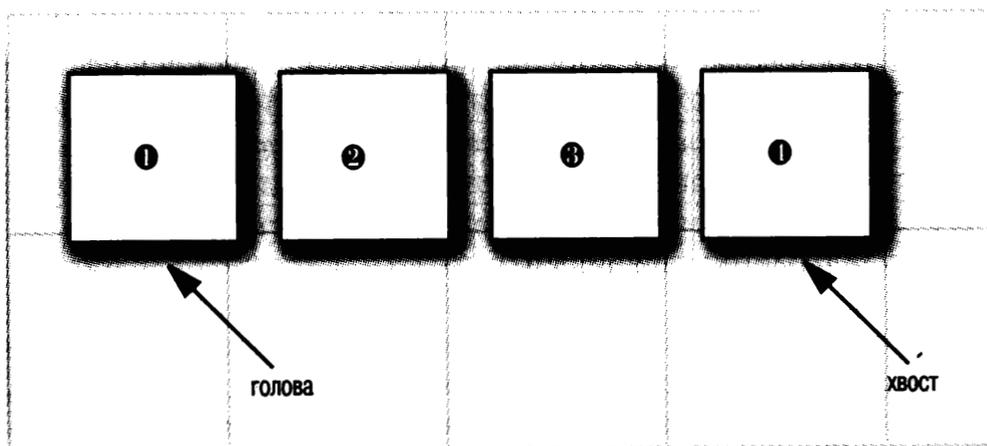


Рис. 13.1. Очередь

Самая элементарная форма интерфейса очереди может выглядеть так, как показано же.

```
interface Queue<E> // простейшая форма интерфейса очереди
                  // из стандартной библиотеки
{
    void add(E element);
    E remove();
    int size();
}
```

Из интерфейса нельзя ничего узнать, каким образом реализована очередь. В одной из широко распространенных реализаций очереди применяется циклический массив, а в другой — связный список (рис. 13.2).

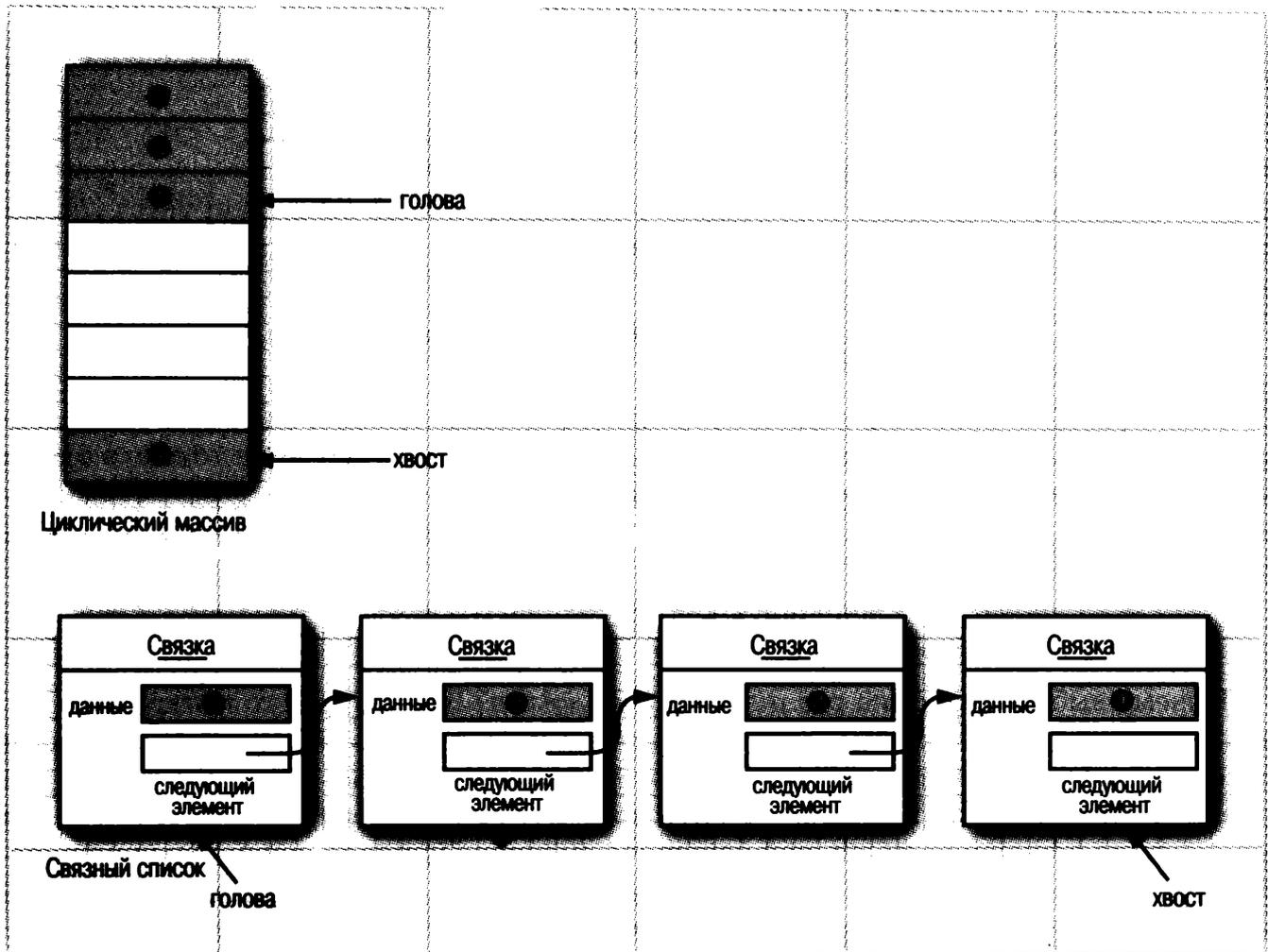


Рис. 13.2. Реализация очереди

Каждая реализация может быть выражена классом, реализующим интерфейс `Queue`, как показано ниже.

```
class CircularArrayQueue<E> implements Queue<E> // это не класс из библиотеки
{
    CircularArrayQueue(int capacity) { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    private E[] elements;
    private int head;
    private int tail;
}
```

```
class LinkedListQueue<E> implements Queue<E> // это не класс из библиотеки
{
    LinkedListQueue() { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    private Link head;
    private Link tail;
}
```



НА ЗАМЕТКУ! Библиотека Java на самом деле не содержит классы `CircularArrayQueue` и `LinkedListQueue`. Они служат здесь лишь в качестве примера, чтобы пояснить принципиальное отличие интерфейса и реализации. Если же требуется организовать очередь на основе циклического массива, воспользуйтесь классом `ArrayDeque`, а для организации очереди в виде связанного списка — классом `LinkedList`, реализующим интерфейс `Queue`.

Применяя в своей программе очередь, совсем не обязательно знать, какая именно реализация используется для построения коллекции. Таким образом, конкретный класс имеет смысл использовать *только* в том случае, если конструируется объект коллекции. А *тип интерфейса* служит лишь для ссылки на коллекцию, как показано ниже.

```
Queue<Customer> expressLane = new CircularArrayQueue<>(100);
expressLane.add(new Customer("Harry"));
```

Если придется изменить решение, то при таком подходе нетрудно будет воспользоваться другой реализацией. Для этого достаточно внести изменения только в одном месте программы: при вызове конструктора. Так, если остановить свой выбор на классе `LinkedListQueue`, код реализации очереди в виде связанного списка будет выглядеть следующим образом:

```
Queue<Customer> expressLane = new LinkedListQueue<>();
expressLane.add(new Customer("Harry"));
```

Почему выбирается одна реализация, а не другая? Ведь из самого интерфейса ничего нельзя узнать об эффективности реализации. Циклический массив в некотором отношении более эффективен, чем связанный список, и поэтому ему обычно отдается предпочтение. Но, как всегда, за все нужно платить. Циклический массив является *ограниченной коллекцией*, имеющей конечную емкость. Поэтому если неизвестен верхний предел количества объектов, которые должна накапливать прикладная программа, то имеет смысл выбрать реализацию очереди на основе связанного списка.

Изучая документацию на прикладной интерфейс API, вы непременно обнаружите другой набор классов, имена которых начинаются на `Abstract`, как, например, `AbstractQueue`. Эти классы предназначены для разработчиков библиотек. В том редком случае, когда вам потребуется реализовать свой собственный класс очереди, вы найдете, что сделать это проще, расширив класс `AbstractQueue`, чем реализовывать все методы интерфейса `Queue`.

Интерфейсы `Collection` и `Iterator` из библиотеки Java

Основополагающим для классов коллекций в библиотеке Java является интерфейс `Collection`. В его состав входят два основных метода:

```
public interface Collection<E>
{
    boolean add(E element);
    Iterator<E> iterator();
    . . .
}
```

В дополнение к ним имеется еще несколько методов, обсуждаемых далее в этой главе. Метод `add()` добавляет элемент в коллекцию. Он возвращает логическое значение `true`, если добавление элемента в действительности изменило коллекцию, а

если коллекция осталась без изменения — логическое значение `false`. Так, если попытаться добавить объект в совокупность, где такой объект уже имеется, вызов метода `add()` не даст желаемого результата, поскольку совокупность не допускает дублирование объектов. А метод `iterator()` возвращает объект, реализующий интерфейс `Iterator`. Объект итератора можно выбрать для обращения ко всем элементам коллекции по очереди.

Итераторы

В состав интерфейса `Iterator` входят три метода:

```
public interface Iterator<E>
{
    E next();
    boolean hasNext();
    void remove();
}
```

Множественно вызывая метод `next()`, можно обратиться к каждому элементу коллекции по очереди. Но если будет достигнут конец коллекции, то метод `next()` сгенерирует исключение типа `NoSuchElementException`. Поэтому перед вызовом метода `next()` следует вызывать метод `hasNext()`. Этот метод возвращает логическое значение `true`, если у объекта итератора все еще имеются объекты, к которым можно обратиться. Если же требуется перебрать все элементы коллекции, то следует запросить итератор, продолжая вызывать метод `next()` до тех пор, пока метод `hasNext()` возвращает логическое значение `true`. В приведенном ниже примере показано, как все это реализуется непосредственно в коде.

```
Collection<String> c = . . . ;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
    String element = iter.next();
    сделать что-нибудь с элементом element
}
```

В версии Java SE 5.0 появилось изящное сокращение этого цикла. Тот же самый код можно написать более компактно, организовав цикл в стиле `for each` следующим образом:

```
for (String element : c)
{
    сделать что-нибудь с элементом element
}
```

Компилятор просто преобразует цикл в стиле `for each` в цикл с итератором. Цикл в стиле `for each` подходит для любых объектов, класс которых реализует интерфейс `Iterable` со следующим единственным методом:

```
public interface Iterable<E>
{
    Iterator<E> iterator();
}
```

Интерфейс `Collection` расширяет интерфейс `Iterable`. Поэтому цикл в стиле `for each` подходит для обращения к элементам любой коллекции из стандартной библиотеки.

Порядок, в котором перебираются элементы, зависит от типа коллекции. Так, если осуществляется перебор элементов коллекции типа `ArrayList`, итератор начинает его с нулевого индекса, увеличивая последнее значение на каждом шаге итерации. Но если осуществляется перебор элементов коллекции типа `HashSet`, то они получаются в совершенно случайном порядке. Можно быть уверенным лишь в том, что за время итерации будут перебраны все элементы коллекции, хотя нельзя сделать никаких предположений относительно порядка их следования. Обычно это не особенно важно, потому что порядок не имеет значения при таких вычислениях, как, например, подсчет суммы или количества совпадений.



НА ЗАМЕТКУ! Программирующие на Java со стажем заметят, что методы `next()` и `hasNext()` из интерфейса `Iterator` служат той же цели, что и методы `nextElement()` и `hasMoreElements()` из интерфейса `Enumeration`. Разработчики библиотеки коллекций в Java могли бы отдать предпочтение интерфейсу `Enumeration`. Но им не понравились громоздкие имена методов, и поэтому они разработали новый интерфейс с более короткими именами методов.

Существует принципиальное отличие между итераторами из библиотеки коллекций Java и других библиотек. В традиционных библиотеках коллекций, как, например, `Standard Template Library` в C++, итераторы моделируются по индексам массива. Имея такой итератор, можно найти элемент, находящийся на данной позиции в массиве, подобно тому, как находится элемент массива `a[i]`, если имеется индекс `i`. Независимо от способа поиска элементов коллекции, итератор можно передвинуть на следующую позицию. Эта операция аналогична приращению индекса `i++` без поиска. Но итераторы в Java действуют иначе. Поиск элементов в коллекции и изменение их позиции тесно взаимосвязаны. Единственный способ найти элемент — вызвать метод `next()`, а в ходе поиска происходит переход на следующую позицию.

Напротив, итераторы в Java следует представлять себе так, как будто они находятся между элементами коллекции. Когда вызывается метод `next()`, итератор *перескакивает* следующий элемент и возвращает ссылку на тот элемент, который он только что прошел (рис. 13.3).



НА ЗАМЕТКУ! Можно прибегнуть к еще одной удобной аналогии, рассматривая объект `Iterator`. `next` в качестве эквивалента объекта `InputStream.read`. При вводе байта из потока этот байт автоматически потребляется. А при последующем вызове метода `read()` потребляется и возвращается следующий байт из потока ввода. Аналогично повторяющиеся вызовы метода `next()` позволяют ввести все элементы из коллекции.

Удаление элементов из коллекции

Метод `remove()` из интерфейса `Iterator` удаляет элемент, который был возвращен при последнем вызове метода `next()`. Во многих случаях это имеет смысл, поскольку нужно проанализировать элемент, прежде чем решаться на его удаление. Но если требуется удалить элемент, находящийся на определенной позиции, то сначала придется пройти его. В приведенном ниже примере показано, как удалить первый элемент из коллекции символьных строк.

```
Iterator<String> it = c.iterator();  
it.next(); // пройти первый элемент коллекции  
it.remove(); // а теперь удалить его
```

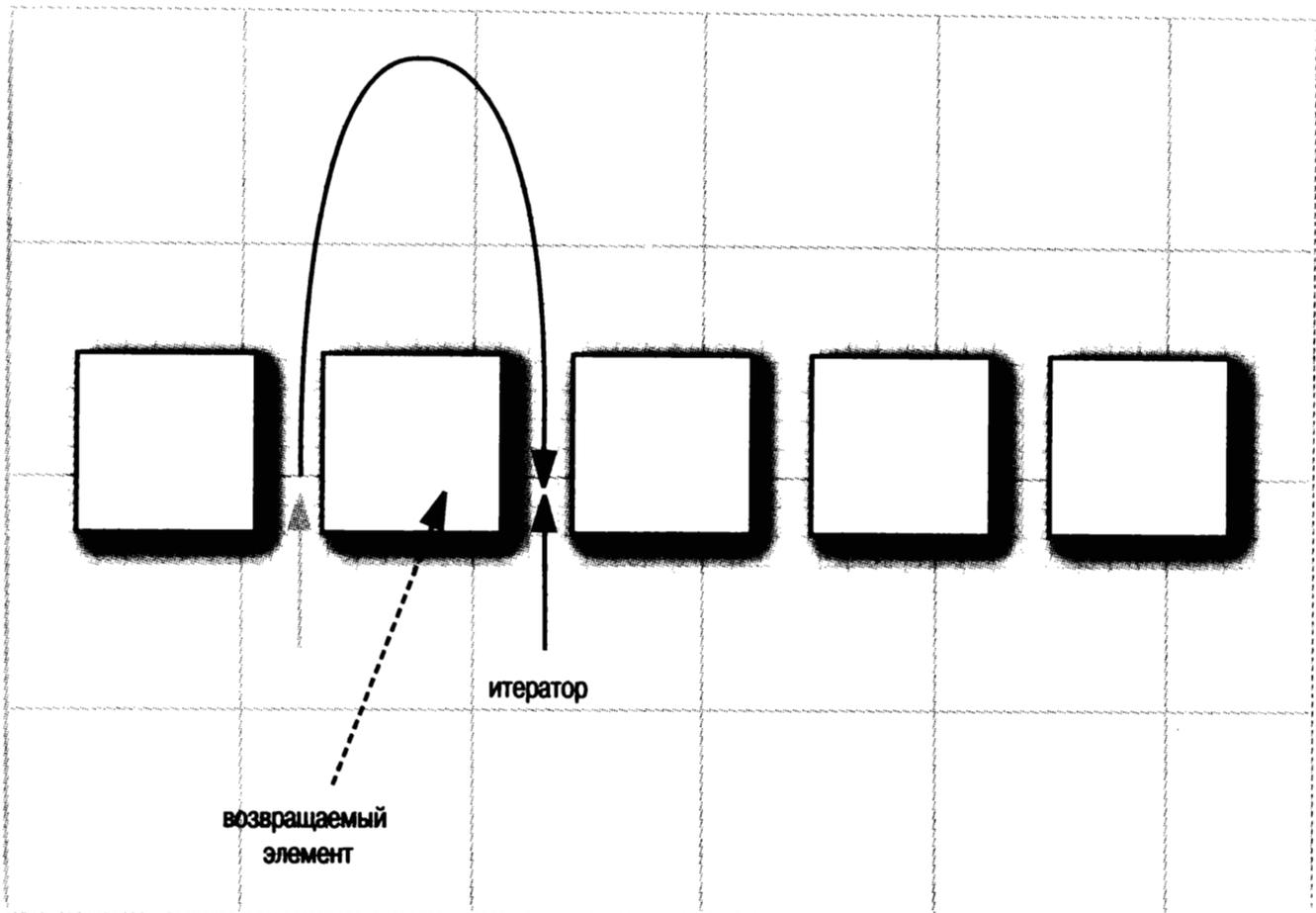


Рис. 13.3. Продвижение итератора по элементам коллекции

Но важнее то, что между вызовами методов `next()` и `remove()` существует определенная связь. В частности, вызывать метод `remove()` не разрешается, если перед ним не был вызван метод `next()`. Если же попытаться сделать это, будет сгенерировано исключение типа `IllegalStateException`. А если требуется удалить из коллекции два соседних элемента, то нельзя просто вызвать метод `remove()` два раза подряд, как показано ниже.

```
it.remove();  
it.remove(); // ОШИБКА!
```

Вместо этого нужно сначала вызвать метод `next()`, чтобы пройти удаляемый элемент, а затем удалить его, как следует из приведенного ниже примера кода.

```
it.remove();  
it.next();  
it.remove(); // Допустимо!
```

Обобщенные служебные методы

Интерфейсы `Collection` и `Iterator` являются обобщенными, а следовательно, можно написать служебные методы для обращения с разнотипными коллекциями. В качестве примера ниже приведен обобщенный служебный метод, проверяющий, содержит ли произвольная коллекция заданный элемент.

```
public static <E> boolean contains(Collection<E> c, Object obj)
{
    for (E element : c)
        if (element.equals(obj))
            return true;
    return false;
}
```

Разработчики библиотеки Java решили, что некоторые из этих служебных методов настолько полезны, что их следует сделать доступными из самой библиотеки. Таким образом, пользователи библиотеки должны быть избавлены от необходимости заново изобретать колесо. По существу, в интерфейсе `Collection` объявляется немало полезных методов, которые должны использоваться во всех реализующих его классах. К числу служебных методов относятся следующие:

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object other)
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

Многие из этих методов самоочевидны, а подробнее о них можно узнать из краткого описания прикладного интерфейса API в конце этого раздела. Безусловно, было бы неудобно, если бы в каждом классе, реализующем интерфейс `Collection`, пришлось реализовывать такое количество служебных методов. Чтобы облегчить жизнь разработчикам, в библиотеке коллекций Java предоставляется класс `AbstractCollection`, где основополагающие методы `size()` и `iterator()` оставлены абстрактными, но реализованы служебные методы:

```
public abstract class AbstractCollection<E>
implements Collection<E>
{
    . . .
    public abstract Iterator<E> iterator();

    public boolean contains(Object obj)
    {
        for (E element : this) // вызвать метод iterator()
            if (element.equals(obj))
                return = true;
        return false;
    }
    . . .
}
```

Конкретный класс коллекции теперь может расширить класс `AbstractCollection`. И тогда именно этот конкретный класс отвечает за реализацию метода `iterator()`, тогда как о служебном методе `contains()` уже позаботились в суперклассе `AbstractCollection`. Но если в его подклассе можно предложить более эффективный способ реализации служебного метода `contains()`, то ничто не мешает сделать это именно в нем.

Это удобная конструкция для архитектуры коллекций. В распоряжении пользователей классов коллекций имеется богатый набор доступных методов из обобщенного интерфейса, но разработчики конкретных структур данных могут не утруждать себя реализацией служебных методов.

`java.util.Collection<E>` 1.2

- `Iterator<E> iterator()`
Возвращает итератор, который можно использовать для перебора элементов коллекции.
- `int size()`
Возвращает количество элементов, хранящихся в коллекции на данный момент.
- `boolean isEmpty()`
Возвращает логическое значение `true`, если коллекция не содержит ни одного из элементов.
- `boolean contains(Object obj)`
Возвращает логическое значение `true`, если коллекция содержит объект, равнозначный `obj`.
- `boolean containsAll(Collection<?> other)`
Возвращает логическое значение `true`, если коллекция содержит все элементы из другой коллекции.
- `boolean add(Object element)`
Добавляет элемент в коллекцию. Возвращает логическое значение `true`, если в результате вызова этого метода коллекция изменилась.
- `boolean addAll(Collection<? extends E> other)`
Добавляет все элементы из другой коллекции в данную. Возвращает логическое значение `true`, если в результате вызова этого метода коллекция изменилась.
- `boolean remove(Object obj)`
Удаляет из коллекции объект, равнозначный `obj`. Возвращает логическое значение `true`, если в результате вызова этого метода коллекция изменилась.
- `boolean removeAll(Collection<?> other)`
Удаляет из данной коллекции все элементы другой коллекции. Возвращает логическое значение `true`, если в результате вызова этого метода коллекция изменилась.
- `void clear()`
Удаляет все элементы из данной коллекции.
- `boolean retainAll(Collection<?> other)`
Удаляет из данной коллекции все элементы, не равнозначные ни одному из элементов другой коллекции. Возвращает логическое значение `true`, если в результате вызова этого метода коллекция изменилась.
- `Object[] toArray()`
Возвращает из коллекции массив объектов.
- `<T> T[] toArray(T[] arrayToFill)`
Возвращает из коллекции массив объектов. Если заполняемый массив `arrayToFill` имеет достаточную длину, он заполняется элементами данной коллекции. Если же остается место, добавляются пустые элементы типа `null`. В противном случае выделяется и заполняется новый массив с тем же типом компонентов и длиной, что и у массива `arrayToFill`.

java.util.Iterator <E> 1.2

- `boolean hasNext()`
Возвращает логическое значение `true`, если в коллекции еще имеются элементы, к которым можно обратиться.
- `E next()`
Возвращает следующий перебираемый объект. Генерирует исключение типа `NoSuchElementException`, если достигнут конец коллекции.
- `void remove()`
Удаляет последний перебираемый объект. Этот метод должен вызываться сразу же после обращения к удаляемому элементу. Если коллекция была видоизменена после обращения к последнему ее элементу, этот метод генерирует исключение типа `IllegalStateException`.

Конкретные коллекции

Прежде чем перейти к подробному обсуждению всех интерфейсов коллекций, было бы полезно рассмотреть конкретные структуры данных, предоставляемые в библиотеке Java. Обсудив подробно наиболее употребительные классы, мы вернемся к вопросам, имеющим более абстрактный характер, чтобы выяснить, каким образом все эти классы организованы в архитектуре коллекций. В табл. 13.1 приведены коллекции из библиотеки Java вместе с кратким описанием назначения каждого из их классов. (Ради простоты в ней не указаны классы потокобезопасных коллекций, о которых речь пойдет к главе 14.) Все классы из табл. 13.1 реализуют интерфейс `Collection`, за исключением классов, имена которых оканчиваются на `Map`, поскольку эти классы реализуют интерфейс `Map`, подробнее рассматриваемый ниже, в разделе “Отображения”.

Таблица 13.1. Конкретные коллекции из библиотеки Java

Тип коллекции	Описание
<code>ArrayList</code>	Индексированная динамически расширяющаяся и сокращающаяся последовательность
<code>LinkedList</code>	Упорядоченная последовательность, допускающая эффективную вставку и удаление на любой позиции
<code>ArrayDeque</code>	Упорядоченная последовательность, реализуемая в виде циклического массива
<code>HashSet</code>	Неупорядоченная коллекция, исключающая дубликаты
<code>TreeSet</code>	Отсортированное множество
<code>EnumSet</code>	Множество значений перечислимого типа
<code>LinkedHashSet</code>	Множество, запоминающее порядок ввода элементов
<code>PriorityQueue</code>	Коллекция, позволяющая эффективно удалять наименьший элемент
<code>HashMap</code>	Структура данных для хранения связанных вместе пар “ключ–значение”
<code>TreeMap</code>	Отображение с отсортированными ключами

Окончание табл. 13.1

Тип коллекции	Описание
EnumMap	Отображение с ключами, относящимися к перечислимому типу
LinkedHashMap	Отображение с запоминанием порядка, в котором добавлялись элементы
WeakHashMap	Отображение со значениями, которые могут удаляться системой "сборки мусора", если они нигде больше не используются
IdentityHashMap	Отображение с ключами, сравниваемыми с помощью операции <code>==</code> , а не вызова метода <code>equals()</code>

Связные списки

Во многих примерах программ, приведенных ранее в этой книге, уже не раз использовались массивы и родственный им класс `ArrayList` динамического списочного массива. Но обычные и списочные массивы страдают существенным недостатком. Удаление элемента из середины массива обходится дорого с точки зрения потребляемых вычислительных ресурсов, потому что все элементы, следующие за удаляемым, приходится перемещать к началу массива (рис. 13.4). Это же справедливо и для ввода элементов в середине массива.

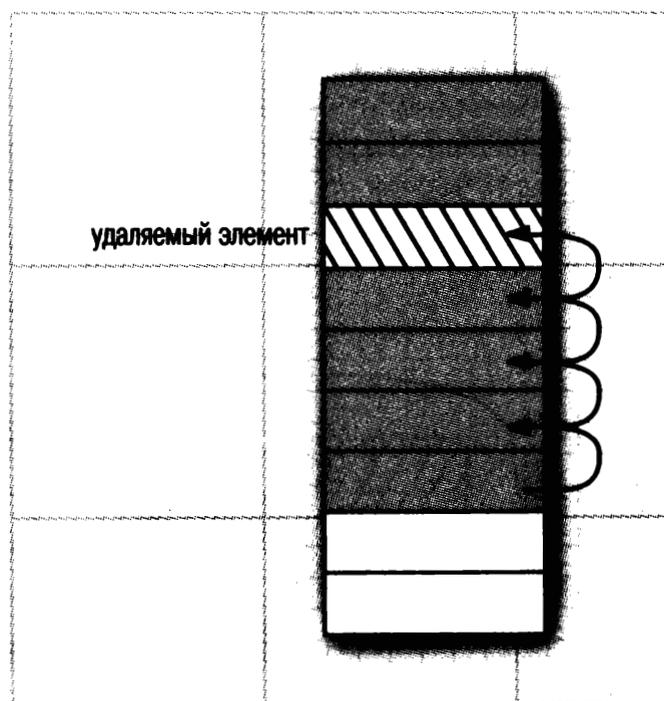


Рис. 13.4. Удаление элемента из массива

Этот недостаток позволяет устранить другая широко известная структура данных — *связный список*. Если ссылки на объекты из массива хранятся в последовательных областях памяти, то каждый объект из связного списка — в отдельной *связке*. В языке программирования Java все связные списки на самом деле являются двунаправленными; т.е. в каждой связке хранится ссылка на ее предшественника (рис. 13.5). Удаление элемента из середины связного списка — недорогая операция с точки зрения потребляемых вычислительных ресурсов, поскольку в этом случае достаточно обновить лишь связки, соседние с удаляемым элементом (рис. 13.6).

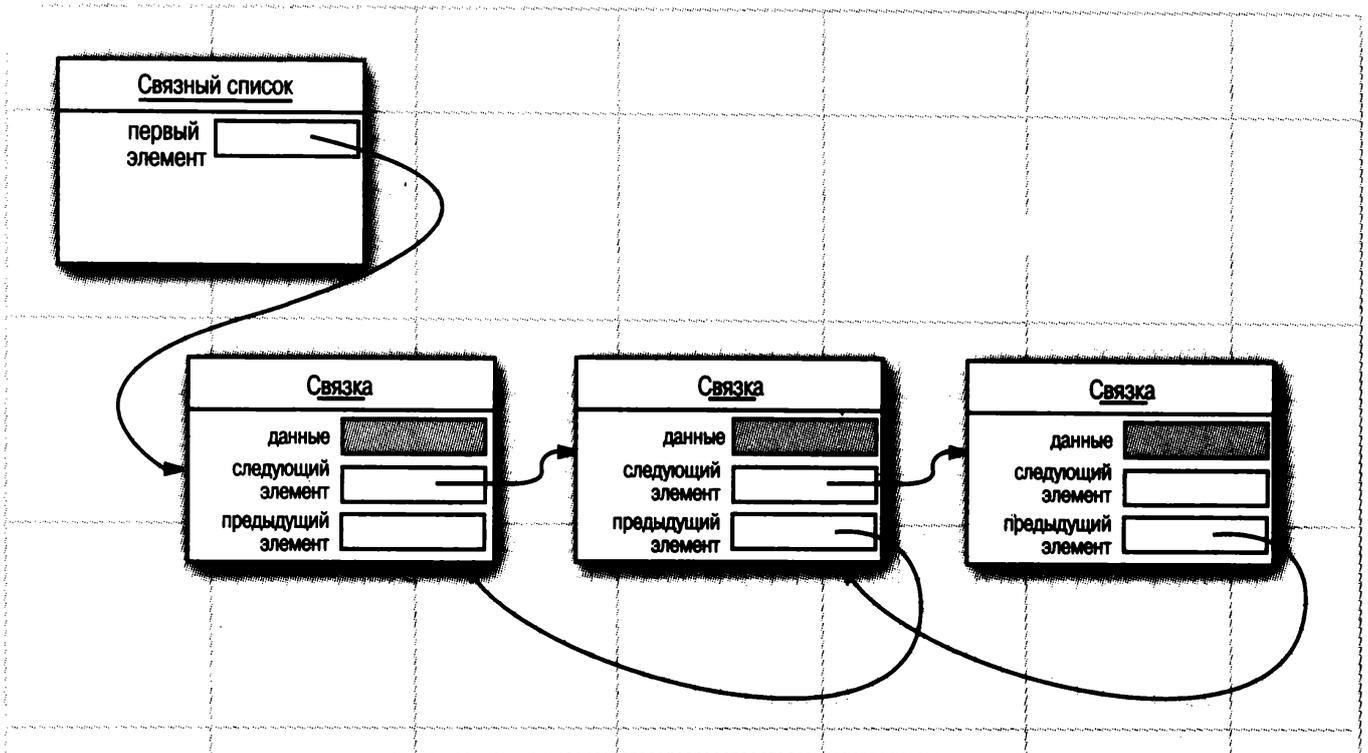


Рис. 13.5. Двухнаправленный связный список

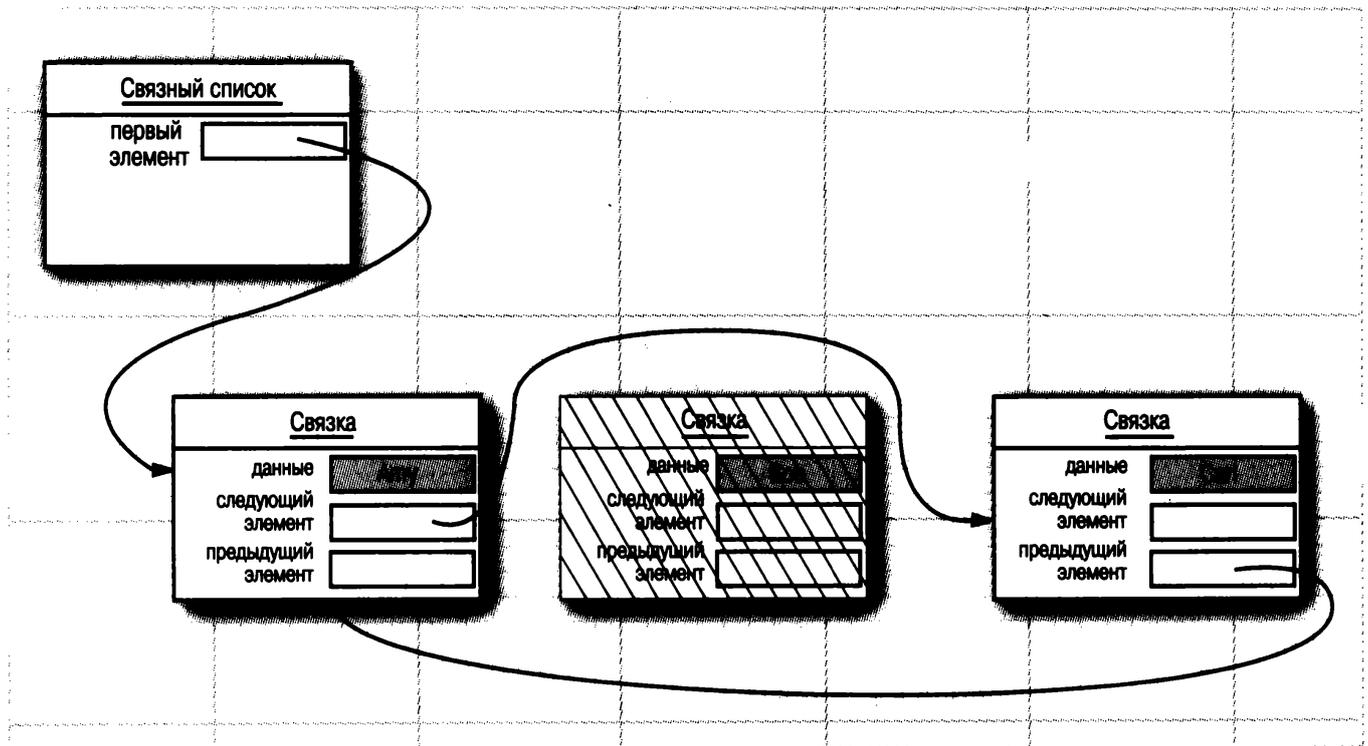


Рис. 13.6. Удаление элемента из связного списка

Если вы когда-нибудь изучали структуры данных и реализацию связных списков, то, возможно, еще помните, насколько хлопотно соединять связки при вводе или удалении элементов связного списка. В таком случае вы будете приятно удивлены, узнав, что в библиотеке коллекций Java для этой цели предусмотрен готовый к вашим услугам класс `LinkedList`. В приведенном ниже примере кода в связный список сначала вводятся три элемента, а затем удаляется второй из них.

```
List<String> staff = new LinkedList<>(); // Объект типа LinkedList,  
                                        // реализующий связный список  
staff.add("Amy");  
staff.add("Bob");  
staff.add("Carl");  
Iterator iter = staff.iterator();  
String first = iter.next(); // обратиться к первому элементу  
String second = iter.next(); // обратиться ко второму элементу  
iter.remove(); // удалить последний перебираемый элемент
```

Но связные списки существенно отличаются от обобщенных коллекций. Связный список — это *упорядоченная коллекция*, в которой имеет значение расположение объектов. Метод `LinkedList.add()` вводит объект в конце списка. Но объекты зачастую требуется вводить где-то в середине списка.

За этот метод `add()`, зависящий от расположения элементов в связном списке, отвечает итератор, поскольку в итераторе описывается расположение элементов в коллекции. Применение итераторов для ввода элементов имеет смысл только для коллекций, имеющих естественный порядок расположения. Например, коллекция типа *множества*, о которой пойдет речь в следующем разделе, не предполагает никакого порядка расположения элементов. Поэтому в интерфейсе `Iterator` отсутствует метод `add()`. Вместо него в библиотеке коллекций предусмотрен следующий подчиненный интерфейс `ListIterator`, содержащий метод `add()`:

```
interface ListIterator<E> extends Iterator<E>  
{  
    void add(E element);  
    . . .  
}
```

В отличие от метода `Collection.add()`, этот метод не возвращает логическое значение типа `boolean`. Предполагается, что операция ввода элемента в список всегда видоизменяет его. Кроме того, в интерфейсе `ListIterator` имеются следующие два метода, которые можно использовать для прохода по списку в обратном направлении:

```
E previous()  
boolean hasPrevious()
```

Как и метод `next()`, метод `previous()` возвращает объект, который он прошел. А метод `listIterator()` из класса `LinkedList` возвращает объект итератора, реализующего интерфейс `ListIterator`, как показано ниже.

```
listIterator<String> iter = staff.listIterator()
```

Метод `add()` вводит новый элемент до текущей позиции итератора. Например, в приведенном ниже фрагменте кода пропускается первый элемент связного списка и вводится элемент "Juliet" перед вторым его элементом (рис. 13.7).

```
List<String> staff = new LinkedList<>();  
staff.add("Amy");  
staff.add("Bob");  
staff.add("Carl");  
ListIterator<String> iter = staff.listIterator();  
iter.next(); // пройти первый элемент списка  
iter.add("Juliet");
```

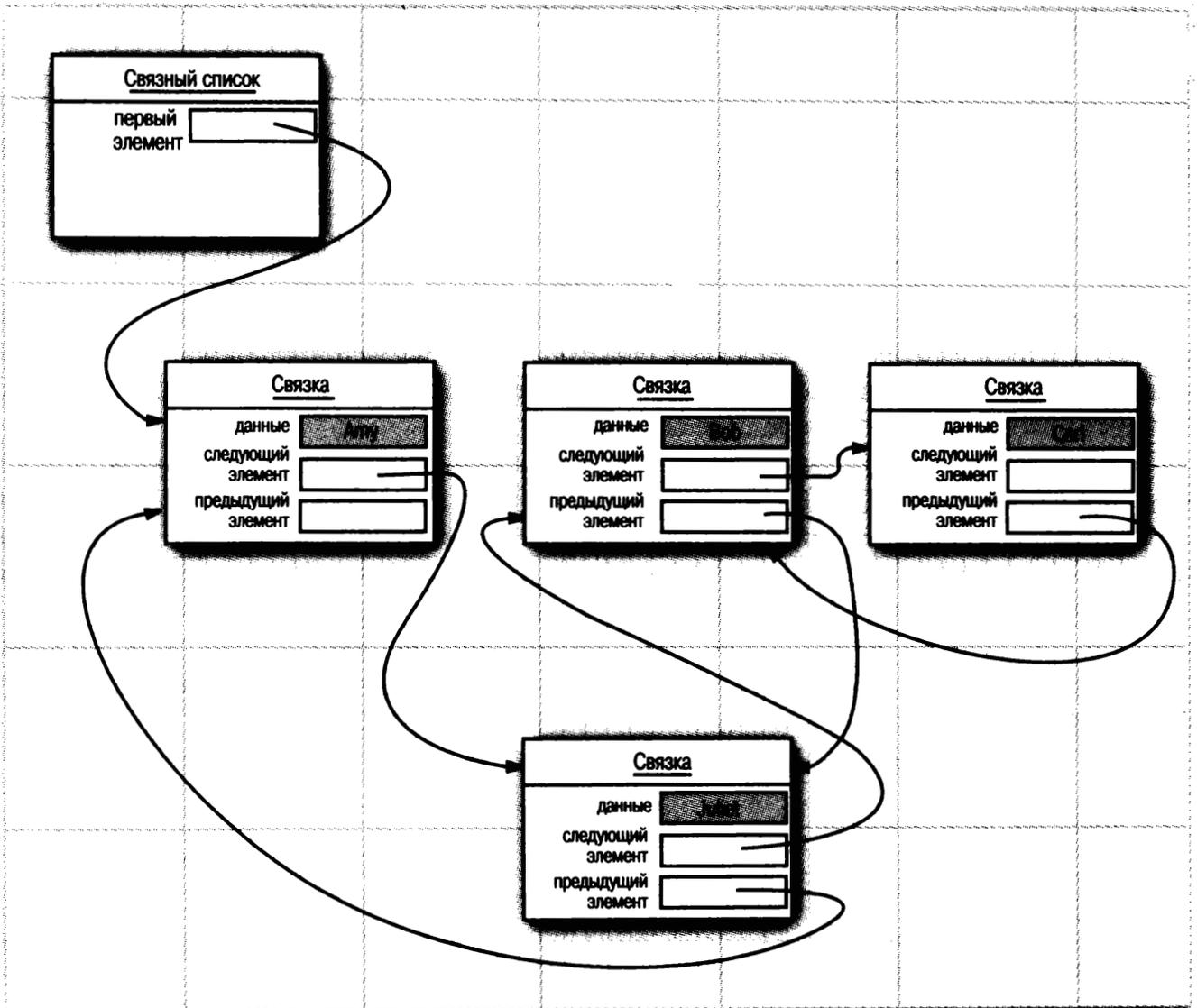


Рис. 13.7. Добавление элемента в связанный список

Если вызвать метод `add()` несколько раз, элементы будут просто вводиться в список в том порядке, в каком они предоставляются. Все они вводятся по очереди до текущей позиции итератора.

Когда выполняется операция ввода элемента с итератором, только что возвращенным методом `listIterator()` и указывающим на начало связанного списка, вводимый элемент располагается в начале списка. Когда же итератор проходит последний элемент списка (т.е. метод `hasNext()` возвращает логическое значение `false`), вводимый элемент располагается в конце списка. Если связанный список содержит n элементов, тогда для ввода нового элемента в нем имеется $n + 1$ доступных мест. Эти места соответствуют $n + 1$ возможным позициям итератора. Так, если связанный список содержит три элемента, A, B и C, для ввода нового элемента в нем имеются четыре возможные позиции, обозначенные ниже знаком `|`.

```
|ABC
A|BC
AB|C
ABC|
```



НА ЗАМЕТКУ! Впрочем, аналогия с курсором | не совсем точна. Операция удаления элемента из списка выполняется не совсем так, как при нажатии клавиши <Backspace>. Если метод `remove()` вызывается сразу же после метода `next()`, то он действительно удаляет элемент, расположенный слева от итератора, как это и происходит при нажатии клавиши <Backspace>. Но если он вызывается сразу же после метода `previous()`, то удаляется элемент, расположенный справа от итератора. К тому же метод `remove()` нельзя вызывать два раза подряд. В отличие от метода `add()`, действие которого зависит только от позиции итератора, действие метода `remove()` зависит и от состояния итератора.

И, наконец, метод `set()` заменяет новым элементом последний элемент, возвращаемый при вызове метода `next()` или `previous()`. Например, в следующем фрагменте кода первый элемент списка заменяется новым значением:

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next(); // возвращает первый элемент списка
iter.set(newValue); // устанавливает в первом элементе новое значение newValue
```

Нетрудно догадаться, что, если один итератор проходит коллекцию в то время, когда другой итератор видоизменяет ее, могут возникнуть конфликтные ситуации. Допустим, итератор установлен до элемента, который только что удален другим итератором. Теперь этот итератор недействителен и не должен больше использоваться. Итераторы связного списка способны обнаруживать такие видоизменения. Если итератор обнаруживает, что коллекция была видоизменена другим итератором или же методом самой коллекции, то он генерирует исключение типа `ConcurrentModificationException`. Рассмотрим в качестве примера следующий фрагмент кода:

```
List<String> list = . . . ;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next(); // генерирует исключение ConcurrentModificationException
```

При вызове метода `iter2.next()` генерируется исключение типа `ConcurrentModificationException`, поскольку итератор `iter2` обнаруживает, что список был внешне видоизменен. Чтобы избежать исключений в связи с попытками одновременного видоизменения, достаточно придерживаться следующего простого правила: к коллекции допускается присоединять сколько угодно итераторов, при условии, что все они служат только для чтения, но присоединить только один итератор, который служит как для чтения, так и для записи.

Обнаружение одновременного видоизменения достигается простым способом. В коллекции отслеживается количество изменяющих ее операций (например, ввода и удаления элементов). Каждый итератор хранит отдельный счетчик операций видоизменений, вызванных им самим. В начале каждого своего метода итератор просто сравнивает значение собственного счетчика видоизменений со значением счетчика видоизменений в коллекции. Если эти значения не равны, генерируется исключение типа `ConcurrentModificationException`.



НА ЗАМЕТКУ! Но из приведенного выше правила обнаружения одновременных видоизменений имеется одно любопытное исключение. Связный список отслеживает лишь структурные видоизменения вроде ввода и удаления связок. А действие метода `set()` не считается структурным

видоизменением. К связному списку можно присоединить несколько итераторов, способных вызывать метод `set()` для изменения содержимого списка. Наличие такой возможности требуется во многих алгоритмах, применяемых в классе `Collections`, о котором речь пойдет далее в этой главе.

Итак, мы рассмотрели основные методы из класса `LinkedList`. А интерфейс `ListIterator` служит как для прохода по элементам связного списка в любом направлении, так и для ввода и удаления элементов из списка.

Как было показано в предыдущем разделе, в интерфейсе `Collection` объявлено немало других полезных методов для операций со связными списками. Они реализованы главным образом в суперклассе `AbstractCollection` класса `LinkedList`. Например, метод `toString()` вызывает одноименный метод для всех элементов списка и формирует одну длинную строку в формате `[A, B, C]`, что удобно для отладки. А метод `contains()` удобен для проверки наличия конкретного элемента в связном списке. Например, в результате вызова `staff.contains("Harry")` возвращается логическое значение `true`, если связный список уже содержит символьную строку "Harry".

В библиотеке коллекций предусмотрен также ряд методов, которые с теоретической точки зрения кажутся избыточными. Ведь в связных списках не поддерживается быстрый произвольный доступ. Если требуется проанализировать n -й элемент связного списка, то начинать придется с самого начала и пройти первые $n - 1$ элементов списка, причем без всяких пропусков. Именно по этой причине программисты обычно не применяют связные списки в тех случаях, когда к элементам требуется обращаться по целочисленному индексу. Тем не менее в классе `LinkedList` предусмотрен метод `get()`, который позволяет обратиться к определенному элементу следующим образом:

```
LinkedList<String> list = . . . ;  
String obj = list.get(n);
```

Конечно, этот метод не очень эффективен. Если он все же применяется, то, скорее всего, к неподходящей для этого структуре данных. Столь обманчивый произвольный доступ *не* годится для обращения к элементам связного списка. Например, приведенный ниже фрагмент кода совершенно неэффективен.

```
for (int i = 0; i < list.size(); i++)  
    сделать что-нибудь с результатом вызова list.get(i);
```

Всякий раз, когда требуется обратиться к другому элементу с помощью метода `get()`, поиск начинается с самого начала списка. В объекте типа `LinkedList` не предпринимается никаких попыток буферизовать данные о расположении элементов в списке.



НА ЗАМЕТКУ! В методе `get()` предусмотрена единственная незначительная оптимизация: если указанный индекс больше величины `size()/2`, то поиск элемента начинается с конца списка.

В интерфейсе итератора списка имеется также метод, предоставляющий индекс текущей позиции в списке. Но поскольку итераторы в Java устроены таким образом, что обозначают позицию *между* элементами коллекции, то таких методов на самом деле два. Так, метод `nextIndex()` возвращает целочисленный индекс того элемента,

который должен быть возвращен при последующем вызове метода `next()`. А метод `previousIndex()` возвращает индекс того элемента, который был бы возвращен при последующем вызове метода `previous()`. И этот индекс будет, конечно, на единицу меньше, чем `nextIndex`. Оба метода действуют эффективно, поскольку в итераторе запоминается текущая позиция. И, наконец, если имеется целочисленный индекс `n`, в результате вызова метода `list.listIterator(n)` будет возвращен итератор, установленный до элемента с индексом `n`. Это означает, при вызове метода `next()` получается тот же самый элемент, что и при вызове метода `list.get(n)`. Следовательно, получение такого итератора малоэффективно.

Если связный список содержит немного элементов, то вряд ли стоит беспокоиться об издержках, связанных с применением методов `set()` и `get()`. Но в таком случае зачем вообще пользоваться связным списком? Единственная причина для его применения — минимизация издержек на ввод и удаление в середине списка. Если в коллекции предполагается лишь несколько элементов, то для ее составления лучше воспользоваться списочным массивом типа `ArrayList`.

Рекомендуется держаться подальше от всех методов, в которых целочисленный индекс служит для обозначения позиции в связном списке. Если требуется произвольный доступ к коллекции, лучше воспользоваться обычным или списочным массивом типа `ArrayList`, а не связным списком.

В примере программы из листинга 13.1 демонстрируется практическое применение связного списка. В этой программе сначала составляются два списка, затем они объединяются, и далее удаляется каждый второй элемент из второго списка, а в завершение проверяется метод `removeAll()`. Рекомендуется тщательно проанализировать порядок выполнения операций в этой программе, уделив особое внимание итераторам. Для этого, возможно, будет полезно составить схематическое представление позиций итератора аналогично следующему:

```
|ACE |BDFG
A|CE |BDFG
AB|CE B|DFG
. . .
```

Следует иметь в виду, что в результате приведенного ниже вызова выводятся все элементы связного списка. Для этой цели вызывается метод `toString()` из класса `AbstractCollection`.

```
System.out.println(a);
```

Листинг 13.1. Исходный код из файла `linkedList/LinkedListTest.java`

```
1 package linkedList;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируются операции со связными списками
7  * @version 1.11 2012-01-26
8  * @author Cay Horstmann
9  */
10 public class LinkedListTest
11 {
12     public static void main(String[] args)
13     {
14         List<String> a = new LinkedList<>();
```

```
15     a.add("Amy");
16     a.add("Carl");
17     a.add("Erica");
18
19     List<String> b = new LinkedList<>();
20     b.add("Bob");
21     b.add("Doug");
22     b.add("Frances");
23     b.add("Gloria");
24
25     // объединить слова из связанных списков b и a
26     ListIterator<String> aIter = a.listIterator();
27     Iterator<String> bIter = b.iterator();
28
29     while (bIter.hasNext())
30     {
31         if (aIter.hasNext()) aIter.next();
32         aIter.add(bIter.next());
33     }
34
35     System.out.println(a);
36
37     // удалить каждое второе слово из связанного списка b
38     bIter = b.iterator();
39     while (bIter.hasNext())
40     {
41         bIter.next(); // пропустить один элемент
42         if (bIter.hasNext())
43         {
44             bIter.next(); // пройти следующий элемент
45             bIter.remove(); // удалить этот элемент
46         }
47     }
48
49     System.out.println(b);
50
51     // групповая операция удаления из связанного списка a
52     // всех слов, составляющих связанный список b
53     a.removeAll(b);
54
55     System.out.println(a);
56 }
57 }
```

java.util.List<E> 1.2

- `ListIterator<E> listIterator()`
Возвращает итератор списка, который можно использовать для перебора элементов списка.
- `ListIterator<E> listIterator(int index)`
Возвращает итератор списка для обращения к элементам списка, если в результате вызова метода `next()` возвращается элемент этого списка с заданным индексом.

- `void add(int i, E element)`
Вводит элемент на указанной позиции в списке.
- `void addAll(int i, Collection<? extends E> elements)`
Вводит все элементы из коллекции на указанной позиции в списке.
- `E remove(int i)`
Удаляет и возвращает элемент на указанной позиции в списке.
- `E get(int i)`
Получает элемент на указанной позиции в списке.
- `E set(int i, E element)`
Заменяет элемент на указанной позиции в списке новым элементом и возвращает старый.
- `int indexOf(Object element)`
Возвращает позицию первого вхождения искомого элемента в списке или значение `-1`, если искомым элемент не найден.
- `int lastIndexOf(Object element)`
Возвращает позицию последнего вхождения искомого элемента в списке или значение `-1`, если искомым элемент не найден.

java.util.ListIterator<E> 1.2

- `void add(E newElement)`
Вводит новый элемент до текущей позиции в списке.
- `void set(E newElement)`
Заменяет новым элементом последний элемент, обращение к которому было сделано при вызове метода `next()` или `previous()`. Генерирует исключение типа `IllegalStateException`, если структура списка была видоизменена в результате последнего вызова метода `next()` или `previous()`.
- `boolean hasPrevious()`
Возвращает логическое значение `true`, если имеется еще один элемент для обращения при итерации по списку в обратном направлении.
- `E previous()`
Возвращает предыдущий объект. Генерирует исключение типа `NoSuchElementException`, если достигнуто начало списка.
- `int nextIndex()`
Возвращает индекс элемента, который должен быть возвращен при последующем вызове метода `next()`.
- `int previousIndex()`
Возвращает индекс элемента, который должен быть возвращен при последующем вызове метода `previous()`.

java.util.LinkedList<E> 1.2

- `LinkedList()`
Конструирует пустой связный список.
- `LinkedList(Collection<? extends E> elements)`
Конструирует связный список и вводит в него элементы из коллекции.
- `void addFirst(E element)`
- `void addLast(E element)`
Вводят элемент в начале или в конце списка.
- `E getFirst()`
- `E getLast()`
Возвращают элемент из начала или из конца списка.
- `E removeFirst()`
- `E removeLast()`
Удаляют и возвращают элемент из начала или из конца списка.

Списочные массивы

В предыдущем разделе обсуждался интерфейс `List` и реализующий его класс `LinkedList`. Интерфейс `List` описывает упорядоченную коллекцию, в которой имеет значение расположение элемента. Существуют два способа для обхода элементов: посредством итератора и произвольного доступа с помощью методов `get()` и `set()`. Второй способ не совсем подходит для связных списков, но применение методов `get()` и `set()` совершенно оправдано для массивов. В библиотеке коллекций предоставляет уже не раз упоминавшийся здесь класс `ArrayList`, также реализующий интерфейс `List`. Класс `ArrayList` инкапсулирует динамически выделяемый массив объектов.



НА ЗАМЕТКУ! Программирующим на Java со стажем, скорее всего, уже приходилось пользоваться классом `Vector` всякий раз, когда возникала потребность в динамическом массиве. Почему же вместо класса `Vector` для подобных целей следует применять класс `ArrayList`? По одной простой причине: все методы из класса `Vector` *синхронизированы*. К объекту типа `Vector` можно безопасно обращаться одновременно из двух потоков. Но если обращаться к такому объекту только из одного потока, что случается намного чаще, то в прикладном коде впустую тратится время на синхронизацию. В отличие от этого, методы `ArrayList` не синхронизированы. Поэтому рекомендуется пользоваться классом `ArrayList` вместо класса `Vector`, если только нет особой необходимости в синхронизации.

Хеш-множества

Связные списки и массивы позволяют указывать порядок, в котором должны следовать элементы. Но если нужно найти конкретный элемент, не помня его позиции в коллекции, то придется перебирать все элементы до тех пор, пока не будет

обнаружено совпадение по критерию поиска. На это может потребоваться некоторое время, если коллекция содержит достаточно много элементов. Если же порядок расположения элементов не имеет особого значения, то для подобных случаев предусмотрены структуры данных, которые позволяют намного быстрее находить элементы в коллекции. Но недостаток таких структур данных заключается в том, что они не обеспечивают никакого контроля над порядком расположения элементов в коллекции. Эти структуры данных организуют элементы в том порядке, который удобен для их собственных целей.

К числу широко известных структур данных для быстрого нахождения объектов относится так называемая *хеш-таблица*. Хеш-таблица вычисляет для каждого объекта целочисленное значение, называемое *хеш-кодом*. Хеш-код — это целочисленное значение, которое выводится определенным образом из данных в полях экземпляра объекта, причем предполагается, что объекты с разными данными порождают разные хеш-коды. В табл. 13.2 приведено несколько примеров хеш-кодов, получаемых в результате вызова метода `hashCode()` из класса `String`.

Таблица 13.2. Хеш-коды, возвращаемые методом `hashCode()`

Символьная строка	Хеш-код
"Lee"	76268
"lee"	107020
"eel"	100300

Если вы определяете свои собственные классы, то на вас ложится ответственность за самостоятельную реализацию метода `hashCode()` (подробнее об этом см. в главе 5). Ваша реализация данного метода должна быть совместима с методом `equals()`. Так, если в результате вызова `a.equals(b)` возвращается логическое значение `true`, то объекты `a` и `b` должны иметь одинаковые хеш-коды. Но самое главное, чтобы хеш-коды вычислялись быстро, а вычисление зависело только от состояния хешируемого объекта, а не от других объектов в хеш-таблице.

В Java хеш-таблицы реализованы в виде массивов или связанных списков. Каждый список называется *группой* (рис. 13.8). Чтобы найти место объекта в таблице, вычисляется его хеш-код и уменьшается его модуль до общего количества групп. Полученное в итоге числовое значение и будет индексом группы, содержащей искомый элемент. Так, если хеш-код объекта равен **76268**, а всего имеется **128** групп, объект должен быть размещен в группе под номером **108** (поскольку остаток от целочисленного деления **76268** на **128** равен **108**). Если повезет, то в этой группе не окажется других элементов, и тогда элемент просто вводится в группу. Безусловно, рано или поздно встретится непустая группа. Такое явление называется *хеш-конфликтом*. В таком случае новый объект сравнивается со всеми объектами в группе, чтобы проверить, находится ли он уже в группе. Если учесть сравнительно равномерное распределение хеш-кодов при достаточно большом количестве групп, то в конечном итоге потребуется лишь несколько сравнений.

Если требуется более полный контроль над производительностью хеш-таблицы, то можно указать первоначальное количество групп. Оно определяет количество групп, используемых для сбора объектов с одинаковыми хеш-значениями. Если же в хеш-таблицу вводится слишком много элементов, количество конфликтов возрастает, что отрицательно сказывается на производительности извлечения элементов из таблицы.

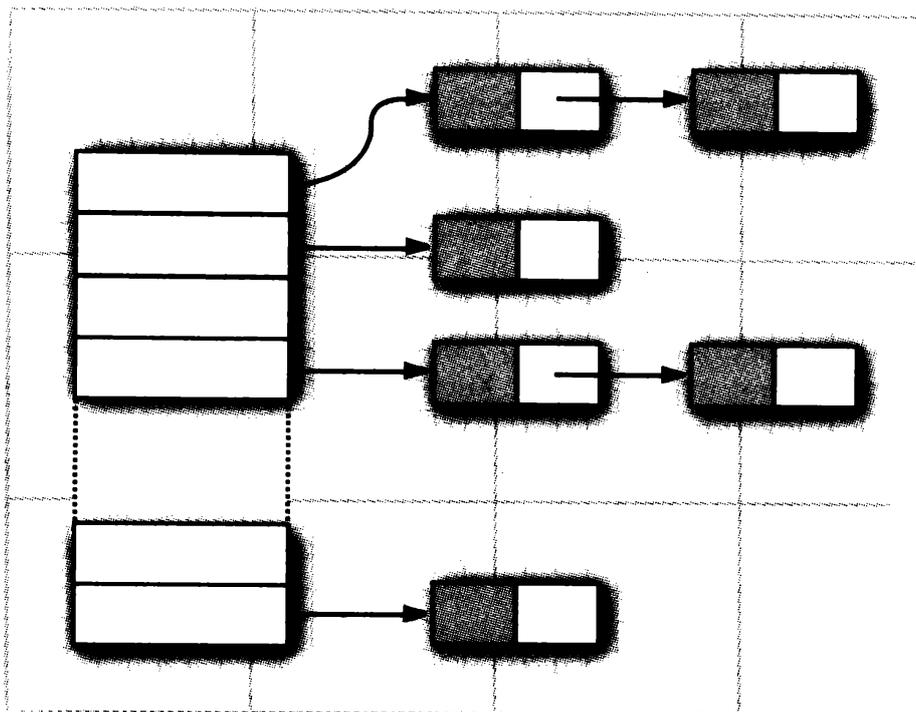


Рис. 13.8. Хеш-таблица

Если приблизительно известно, сколько элементов в конечном итоге окажется в хеш-таблице, можно установить количество групп. Обычно это количество устанавливается в пределах от 75 до 150% от ожидаемого числа элементов. Некоторые исследователи полагают, что количество групп должно быть выражено простым числом, чтобы предотвратить группирование ключей. Но на этот счет нет общего мнения. В стандартной библиотеке используются количества групп, выражаемые числом, являющимся степенью 2, по умолчанию — 16. (Любое указываемое значение размеров таблицы автоматически округляется до следующей степени 2.)

Безусловно, далеко не всегда известно, сколько элементов придется хранить в хеш-таблице, а кроме того, первоначально предполагаемое их количество может оказаться заниженным. Если хеш-таблица становится чрезмерно заполненной, ее следует *повторно* хешировать. Для повторного хеширования создается новая таблица с большим количеством групп, все элементы старой таблицы вводятся в новую, а старая таблица отвергается. *Коэффициент загрузки* определяет момент повторного хеширования таблицы. Так, если коэффициент загрузки равен 0,75 (по умолчанию), а таблица заполнена более чем на 75%, она автоматически хешируется повторно с увеличенным вдвое количеством групп. Для большинства приложений целесообразно оставить коэффициент загрузки равным 0,75.

Хеш-таблицы можно использовать для реализации ряда важных структур данных. Простейшая из них относится к типу множества. *Множество* — это совокупность элементов, не содержащая дубликатов. Метод `add()` из множества сначала пытается найти вводимый объект, и вводит только в том случае, если его еще нет во множестве.

В библиотеке коллекций Java предоставляется класс `HashSet`, реализующий множество на основе хеш-таблицы. Элементы вводятся в такое множество методом `add()`. А метод `contains()` переопределяется, чтобы осуществлять быстрый поиск элемента во множестве. Он проверяет элементы только одной группы, а не все элементы коллекции.

Итератор хеш-множества перебирает все группы по очереди. В результате хеширования элементы распределяются по таблице, и создается впечатление, будто

обращение к ним происходит в случайном порядке. Поэтому классом `HashSet` следует пользоваться только в том случае, если порядок расположения элементов в коллекции не имеет особого значения.

В примере программы из листинга 13.2 отдельные слова текста вводятся из потока `System.in` в хеш-множество, а затем выводятся из него. Например, данной программе можно направить англоязычный текст книги “Алиса в стране чудес” (доступный по адресу <http://www.gutenberg.net>), запустив ее из командной строки следующим образом:

```
java SetTest < alice30.txt
```

Программа введет все слова из стандартного потока ввода в хеш-множество, а затем переберет все неповторяющиеся слова в хеш-множестве и выведет их количество. (Так, текст книги “Алиса в стране чудес” содержит **5909** неповторяющихся слов, включая уведомление об авторском праве в самом начале.) Слова извлекаются из хеш-множества в случайном порядке.



ВНИМАНИЕ! Будьте внимательны и аккуратны, изменяя элементы хеш-множества. Если хеш-код элемента изменится, этот элемент уже не будет находиться на правильной позиции в структуре данных.

Листинг 13.2. Исходный код из файла `set/SetTest.java`

```
1 package set;
2
3 import java.util.*;
4
5 /**
6  * В этой программе выводятся все неповторяющиеся слова,
7  * введенные из стандартного потока System.in во множество
8  * @version 1.11 2012-01-26
9  * @author Cay Horstmann
10 */
11 public class SetTest
12 {
13     public static void main(String[] args)
14     {
15         Set<String> words = new HashSet<>(); // объект типа HashSet,
16                                             // реализующий хеш-множество
17         long totalTime = 0;
18
19         Scanner in = new Scanner(System.in);
20         while (in.hasNext())
21         {
22             String word = in.next();
23             long callTime = System.currentTimeMillis();
24             words.add(word);
25             callTime = System.currentTimeMillis() - callTime;
26             totalTime += callTime;
27         }
28
29         Iterator<String> iter = words.iterator();
30         for (int i = 1; i <= 20 && iter.hasNext(); i++)
31             System.out.println(iter.next());
```

```

32     System.out.println(" . . .");
33     System.out.println(words.size() + " distinct words. " +
34         totalTime + " milliseconds.");
35 }
36 }

```

java.util.HashSet<E> 1.2

- HashSet ()
Конструирует пустое хеш-множество.
- HashSet(Collection<? extends E> elements)
Конструирует хеш-множество и вводит в него все элементы из коллекции.
- HashSet(int initialCapacity)
Конструирует пустое хеш-множество заданной емкости (количество групп).
- HashSet(int initialCapacity, float loadFactor)
Конструирует пустое хеш-множество заданной емкости и с указанным коэффициентом загрузки (числовым значением в пределах от 0,0 до 1,0, определяющим предельный процент заполнения хеш-таблицы, по достижении которого происходит повторное хеширование).

java.lang.Object 1.0

- int hashCode ()
Возвращает хеш-код данного объекта. Хеш-код может быть любым целым значением (положительным или отрицательным). Определения методов equals () и hashCode () должны быть согласованы: если в результате вызова x.equals (y) возвращается логическое значение true, то в результате вызова x.hashCode () должно возвращать то же самое значение, что и в результате вызова y.hashCode ().

Древоподобные множества

Класс TreeSet реализует древоподобное множество, подобное хеш-множеству, но с одним дополнительным усовершенствованием: древоподобное множество представляет собой *отсортированную коллекцию*. В такую коллекцию можно вводить элементы в любом порядке. Когда же выполняется перебор ее элементов, извлекаемые из нее значения оказываются автоматически отсортированными. Допустим, в такую коллекцию сначала введены три символьные строки, а затем перебраны все введенные в нее элементы:

```

SortedSet<String> sorter = new TreeSet<>(); // объект типа TreeSet,
                                           // реализующий отсортированную коллекцию
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
for (String s : sorter) System.println(s);

```

Полученные в итоге значения выводятся в отсортированном порядке: Amy, Bob, Carl. Как следует из имени класса `TreeSet`, сортировка обеспечивается древовидной структурой данных. (В текущей реализации используется структура так называемого *красно-черного дерева*. Подробное описание древовидных структур данных приведено в книге *Алгоритмы: построение и анализ. 3-е издание*. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн, ИД "Вильямс", 2013 г.) Всякий раз при вводе элемента в древовидное множество он размещается на правильно отсортированной позиции. Таким образом, итератор всегда перебирает элементы такого множества в отсортированном порядке.

Ввод элемента в древовидное множество происходит медленнее, чем в хеш-таблицу, но все же намного быстрее, чем в требуемое место массива или связанного списка. Если древовидное множество состоит из n элементов, то в среднем требуется $\log_2 n$ сравнений, чтобы найти правильное расположение нового элемента. Так, если древовидное множество уже содержит 1000 элементов, для ввода нового элемента потребуется около 10 сравнений.

Таким образом, ввод элементов в коллекцию типа `TreeSet` происходит немного медленнее, чем в коллекцию типа `HashSet` (см. для сравнения табл. 13.3), но в то же время элементы коллекции типа `TreeSet` сортируются автоматически.

Таблица 13.3. Сравнение операций ввода элементов в древовидное и хеш-множество

Документ	Всего слов	Количество неповторяющихся слов	Коллекция типа <code>HashSet</code>	Коллекция типа <code>TreeSet</code>
"Алиса в стране чудес"	28195	5909	5 с	7 с
"Граф Монте-Кристо"	466300	37545	75 с	98 с

`java.util.TreeSet<E> 1.2`

- `TreeSet()`
Конструирует пустое древовидное множество.
- `TreeSet(Collection<? extends E> elements)`
Конструирует древовидное множество и вводит в него все элементы из коллекции.

Сравнение объектов

Откуда классу `TreeSet` известно, как сортировать элементы коллекции? По умолчанию в нем предполагается, что в коллекцию вводятся элементы в виде объектов класса, реализующего интерфейс `Comparable`. В этом интерфейсе определяется следующий единственный метод:

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

В результате вызова `a.compareTo(b)` должно быть возвращено нулевое значение, если объекты `a` и `b` равнозначны; отрицательное числовое значение, если объект `a` предшествует объекту `b` в порядке сортировки; или же положительное числовое

значение, если объект *a* следует после объекта *b*. Точное значение не играет роли; важен только знак (> 0 , 0 , или < 0). Несколько стандартных классов платформы Java реализуют интерфейс `Comparable`. Примером тому может служить класс `String`. Его метод `compareTo()` сравнивает символьные строки в алфавитном порядке (иногда называемом *лексикографическим порядком*).

Если вы вводите свои собственные объекты, то должны самостоятельно определить порядок их сортировки, реализуя интерфейс `Comparable`. В классе `Object` не предусмотрена реализация интерфейса `Comparable` по умолчанию. В качестве примера ниже показано, как отсортировать объекты типа `Item` по номеру изделия (`partNumber`):

```
class Item implements Comparable<Item>
{
    public int compareTo(Item other)
    {
        return partNumber - other.partNumber;
    }
    . . .
}
```

Если сравниваются два *положительных* целых значения, как, например, номера изделий в данном примере, то можно просто вернуть разность сравниваемых значений. Она будет отрицательной, если первый элемент должен предшествовать второму; равной нулю, если номера изделий совпадают; а иначе — положительной.



ВНИМАНИЕ! Такой прием оказывается пригодным лишь в том случае, если целые числовые значения находятся в относительно небольших пределах. Если же *x* окажется очень большим положительным числом, а *y* — таким же большим, но отрицательным, то их вычитание может привести к переполнению.

Но применение интерфейса `Comparable` для определения порядка сортировки имеет очевидные ограничения. Каждый класс может реализовать этот интерфейс лишь один раз. Но что делать, если требуется отсортировать группу элементов по номеру изделия в одной коллекции и по его описанию — в другой? И что делать, если нужно отсортировать объекты класса, разработчик которого не позаботился о реализации интерфейса `Comparable`?

В подобных случаях древовидному множеству следует дать команду использовать другой метод сравнения, передав ему объект типа `Comparator` в конструкторе класса `TreeSet`. В интерфейсе `Comparator` объявляется метод `compare()` с двумя явными параметрами, как показано ниже.

```
public interface Comparator<T>
{
    int compare(T a, T b);
}
```

Аналогично методу `compareTo()`, метод `compare()` возвращает отрицательное целое значение, если объект *a* предшествует объекту *b*; нулевое значение — если они одинаковы, а иначе — положительное целое значение. Для того чтобы отсортировать элементы по их описанию, достаточно определить класс, реализующий интерфейс `Comparator`, как показано ниже.

```
class ItemComparator implements Comparator<Item>
{
```

```
public int compare(Item a, Item b)
{
    String descrA = a.getDescription();
    String descrB = b.getDescription();
    return descrA.compareTo(descrB);
}
}
```

Затем объект этого класса передается конструктору древовидного множества следующим образом:

```
ItemComparator comp = new ItemComparator();
SortedSet<Item> sortByDescription = new TreeSet<>(comp);
```

Если конструируется дерево с компаратором, этот объект используется в нем всякий раз, когда требуется сравнить два элемента. Следует, однако, иметь в виду, что у такого компаратора элементов отсутствуют собственные данные. Он содержит метод сравнения. Такие объекты иногда называют *функциональными объектами*. Функциональные объекты часто создаются по ходу выполнения в виде экземпляров анонимных внутренних классов, как выделено полужирным в приведенном ниже примере кода.

```
SortedSet<Item> sortByDescription = new TreeSet<>(new
    Comparator<Item>()
{
    public int compare(Item a, Item b)
    {
        String descrA = a.getDescription();
        String descrB = b.getDescription();
        return descrA.compareTo(descrB);
    }
})
```



НА ЗАМЕТКУ! На самом деле обобщенный интерфейс `Comparator<T>` объявлен с двумя методами: `compare()` и `equals()`. Разумеется, в каждом классе имеется свой метод `equals()`, поэтому добавление этого метода в объявление интерфейса кажется не совсем оправданным. В документации на прикладной интерфейс API поясняется, что переопределять метод `equals()` совершенно не обязательно, но если все-таки сделать это, то в некоторых случаях можно повысить производительность. Например, метод `addAll()` из класса `TreeSet` действует наиболее эффективно, если элементы вводятся из другой коллекции, где используется такой же компаратор.

Если снова проанализировать данные, приведенные в табл. 13.3, то может возникнуть следующий вопрос: почему бы не пользоваться постоянно древовидным множеством вместо хеш-множества? Ведь ввод элементов в такое множество происходит не намного медленнее, а сами элементы автоматически сортируются. Ответ на этот вопрос зависит от данных, которые приходится обрабатывать. Если не нужны отсортированные данные, то и нет смысла расплачиваться за издержки сортировки. Но важнее другое: для некоторых данных намного сложнее определить порядок сортировки, чем хеш-функцию. Хеш-функция должна лишь достаточно равномерно распределять объекты, тогда как функция сравнения — различать объекты с абсолютной точностью.

Чтобы нагляднее продемонстрировать это различие, рассмотрим задачу составления коллекции прямоугольников. Если воспользоваться для этого классом `TreeSet`,

то придется предоставить объект типа `Comparator<Rectangle>`. Как следует сравнивать два прямоугольника? По их площади? Но такое сравнение не годится. Ведь можно создать два прямоугольника с разными координатами, но одинаковой площадью. Порядок сортировки для дерева должен быть *общим*. Любые два элемента должны быть сравнимы, а в результате сравнения должно быть возвращено нулевое значение, если элементы равны. Для прямоугольников имеется способ лексикографического упорядочения по координатам, но он кажется неестественным и сложным для вычисления. С другой стороны, хеш-функция всегда определена для класса `Rectangle`. Она просто хеширует координаты.



НА ЗАМЕТКУ! С версии Java SE 6 класс `TreeSet` реализует интерфейс `NavigableSet`. В этот интерфейс введено несколько служебных методов для обнаружения элементов и их прохода в обратном порядке. Подробнее о них — далее, в описании прикладного интерфейса API.

В примере программы из листинга 13.3 строятся два древовидных множества объектов типа `Item`. Первое множество сортируется по номеру изделия в каталоге, т.е. по порядку, выбираемому для объектов типа `Item` по умолчанию. А второе множество сортируется по описанию изделия с помощью специального компаратора. Само же изделие и его номер описываются в классе `Item` из листинга 13.4.

Листинг 13.3. Исходный код из файла `treeSet/TreeSetTest.java`

```
1 package treeSet;
2
3 /**
4  * @version 1.12 2012-01-26
5  * @author Cay Horstmann
6  */
7
8 import java.util.*;
9
10 /**
11  * В этой программе множество элементов сортируется
12  * путем сравнения их описания
13  */
14 public class TreeSetTest
15 {
16     public static void main(String[] args)
17     {
18         SortedSet<Item> parts = new TreeSet<>();
19         parts.add(new Item("Toaster", 1234));
20         parts.add(new Item("Widget", 4562));
21         parts.add(new Item("Modem", 9912));
22         System.out.println(parts);
23
24         SortedSet<Item> sortByDescription = new TreeSet<>(new
25             Comparator<Item>()
26             {
27                 public int compare(Item a, Item b)
28                 {
29                     String descrA = a.getDescription();
30                     String descrB = b.getDescription();
31                     return descrA.compareTo(descrB);
```

```
32     }
33     });
34
35     sortByDescription.addAll(parts);
36     System.out.println(sortByDescription);
37 }
38 }
```

Листинг 13.4. Исходный код из файла treeSet/Item.java

```
1 package treeSet;
2
3 import java.util.*;
4
5 /**
6  * Описание изделия и его номер по каталогу
7  */
8 public class Item implements Comparable<Item>
9 {
10     private String description;
11     private int partNumber;
12
13     /**
14      * Конструирует объект изделия
15      * @param aDescription Описание изделия
16      * @param aPartNumber Номер изделия по каталогу
17      */
18     public Item(String aDescription, int aPartNumber)
19     {
20         description = aDescription;
21         partNumber = aPartNumber;
22     }
23
24     /**
25      * Получает описание данного изделия
26      * @return Возвращает описание данного изделия
27      */
28     public String getDescription()
29     {
30         return description;
31     }
32
33     public String toString()
34     {
35         return "[description=" + description + ",
36             partNumber=" + partNumber + "]";
37     }
38     public boolean equals(Object otherObject)
39     {
40         if (this == otherObject) return true;
41         if (otherObject == null) return false;
42         if (getClass() != otherObject.getClass()) return false;
43         Item other = (Item) otherObject;
44         return Objects.equals(description, other.description) &&
45             partNumber == other.partNumber;
46     }
47
48     public int hashCode()
```

```
49 {
50     return Objects.hash(description, partNumber);
51 }
52
53 public int compareTo(Item other)
54 {
55     return Integer.compare(partNumber, other.partNumber);
56 }
57 }
```

java.lang.Comparable<T> 1.2

- `int compareTo(T other)`

Сравнивает данный объект (*this*) с другим объектом (*other*) и возвращает отрицательное числовое значение, если данный объект предшествует другому объекту; нулевое значение, если сравниваемые объекты считаются одинаковыми в выбранном порядке сортировки; или же положительное числовое значение, если данный объект следует после другого объекта.

java.util.Comparator<T> 1.2

- `int compare(T a, T b)`

Сравнивает два объекта и возвращает отрицательное числовое значение, если объект *a* предшествует объекту *b*; нулевое значение, если сравниваемые объекты считаются одинаковыми в выбранном порядке сортировки; или же положительное числовое значение, если объект *a* следует после объекта *b*.

java.util.SortedSet<E> 1.2

- `Comparator<? super E> comparator()`

Возвращает компаратор для сортировки элементов или пустое значение `null`, если элементы сравниваются методом `compareTo()` из интерфейса `Comparable`.

- `E first()`
- `E last()`

Возвращают наименьший и наибольший элементы из отсортированного множества.

java.util.NavigableSet<E> 6

- `E higher(E value)`
- `E lower(E value)`

Возвращают наименьший элемент, который больше *value*, или наибольший элемент, который меньше *value*, а если такого элемента не обнаружено — пустое значение `null`.

- `E ceiling(E value)`
- `E floor(E value)`

Возвращают наименьший элемент, который больше или равен `value`, или наибольший элемент, который меньше или равен `value`, а если такого элемента не обнаружено — пустое значение `null`.

- `E pollFirst()`
- `E pollLast()`

Удаляют и возвращают наименьший или наибольший элемент во множестве или же пустое значение `null`, если множество оказывается пустым.

- `Iterator<E> descendingIterator()`

Возвращает итератор, проходящий данное множество в обратном порядке.

`java.util.TreeSet<E> 1.2`

- `TreeSet()`

Конструирует древовидное множество для сортировки объектов типа `Comparable`.

- `TreeSet(Comparator<? super E> c)`

Конструирует древовидное множество и использует указанный компаратор для сортировки его элементов.

- `TreeSet(SortedSet<? extends E> elements)`

Конструирует древовидное множество и вводит в него элементы из отсортированного множества, используя тот же самый компаратор элементов, что и для заданного отсортированного множества.

Односторонние и двухсторонние очереди

Как упоминалось выше, обычная (односторонняя) очередь позволяет эффективно вводить элементы в свой хвост и удалять элементы из своей головы, а *двухсторонняя очередь* — вводить и удалять элементы на обоих своих концах, хотя ввод элементов в середине очереди не поддерживается. В версии Java SE 6 появился интерфейс `Deque`, реализуемый классами `ArrayDeque` и `LinkedList`, причем оба класса предоставляют двухстороннюю очередь, которая может расти по мере надобности. В главе 14 будут приведены примеры применения ограниченных одно- и двухсторонних очередей.

`java.util.Queue<E> 5.0`

- `boolean add(E element)`
- `boolean offer(E element)`

Вводят заданный элемент в конце очереди и возвращают логическое значение `true`, если очередь не заполнена. Если же очередь заполнена, первый метод генерирует исключение типа `IllegalStateException`, тогда как второй метод возвращает логическое значение `false`.

- `E remove()`

- `E poll()`

Удаляют и возвращают элемент из головы очереди, если очередь не пуста. Если же очередь пуста, то первый метод генерирует исключение типа `NoSuchElementException`, тогда как второй метод возвращает пустое значение `null`.

- `E element()`

- `E peek()`

Возвращают элемент из головы очереди, не удаляя его, если очередь не пуста. Если же очередь пуста, то первый метод генерирует исключение типа `NoSuchElementException`, тогда как второй метод возвращает пустое значение `null`.

`java.util.Deque<E>` 6

- `void addFirst(E element)`

- `void addLast(E element)`

- `boolean offerFirst(E element)`

- `boolean offerLast(E element)`

Вводят заданный элемент в голове или хвосте двухсторонней очереди. Если очередь заполнена, первые два метода генерируют исключение типа `IllegalStateException`, тогда как последние два метода возвращают логическое значение `false`.

- `E removeFirst()`

- `E removeLast()`

- `E pollFirst()`

- `E pollLast()`

Удаляют и возвращают элемент из головы очереди, если очередь не пуста. Если же она пуста, то первые два метода генерируют исключение типа `NoSuchElementException`, тогда как последние два метода возвращают пустое значение `null`.

- `E getFirst()`

- `E getLast()`

- `E peekFirst()`

- `E peekLast()`

Возвращают элемент из головы очереди, не удаляя ее, если очередь не пуста. Если же она пуста, первые два метода генерируют исключение типа `NoSuchElementException`, тогда как последние два метода возвращают пустое значение `null`.

java.util.ArrayDeque<E> 6

- `ArrayDeque()`
- `ArrayDeque(int initialCapacity)`

Конструируют неограниченные двунаправленные очереди с начальной емкостью 16 элементов или заданной начальной емкостью.

Очереди по приоритету

В очередях по приоритету элементы извлекаются в отсортированном порядке после того, как они были введены в произвольном порядке. Следовательно, в результате каждого вызова метода `remove()` получается наименьший из элементов, находящихся в очереди. Но в очереди по приоритету сортируются не все ее элементы. Если выполняется перебор элементов такой очереди, они не обязательно оказываются отсортированными. В очереди по приоритету применяется изящная и эффективная структура данных, называемая “кучей”. “Куча” — это самоорганизующееся двоичное дерево, в котором операции ввода и удаления вызывают перемещение наименьшего элемента в корень, не тратя времени на сортировку всех элементов очереди.

Подобно древовидному множеству, очередь по приоритету может содержать элементы класса, реализующего интерфейс `Comparable`, или же принимать объект типа `Comparator`, предоставляемый конструктору ее класса. Как правило, очередь по приоритету применяется для планирования заданий на выполнение. У каждого задания имеется свой приоритет. Задания вводятся в очередь в случайном порядке. Когда новое задание может быть запущено на выполнение, наиболее высокоприоритетное задание удаляется из очереди. (По традиции приоритет 1 считается наивысшим, поэтому в результате операции удаления из очереди извлекается наименьший элемент.)

В примере программы из листинга 13.5 демонстрируется применение очереди по приоритету непосредственно в коде. В отличие от перебора элементов древовидного множества, в данном примере элементы очереди не перебираются в отсортированном порядке. Но удаление из очереди по приоритету всегда касается наименьшего ее элемента.

Листинг 13.5. Исходный код из файла `priorityQueue/PriorityQueueTest.java`

```
1 package priorityQueue;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируется применение очереди по приоритету
7  * @version 1.01 2012-01-26
8  * @author Cay Horstmann
9  */
10 public class PriorityQueueTest
11 {
12     public static void main(String[] args)
13     {
14         PriorityQueue<GregorianCalendar> pq = new PriorityQueue<>();
```

```
15 pq.add(new GregorianCalendar(  
16     1906, Calendar.DECEMBER, 9)); // G. Hopper  
17 pq.add(  
18     new GregorianCalendar(1815, Calendar.DECEMBER, 10)); // A. Lovelace  
19 pq.add(  
20     new GregorianCalendar(1903, Calendar.DECEMBER, 3)); // J. von Neumann  
18 pq.add(new GregorianCalendar(1910, Calendar.JUNE, 22)); // K. Zuse  
19  
21 System.out.println("Iterating over elements...");  
22 for (GregorianCalendar date : pq)  
23     System.out.println(date.get(Calendar.YEAR));  
24 System.out.println("Removing elements...");  
25 while (!pq.isEmpty())  
26     System.out.println(pq.remove().get(Calendar.YEAR));  
27 }  
28 }
```

java.util.PriorityQueue 5.0

- PriorityQueue()
- PriorityQueue(int *initialCapacity*)
Конструируют очередь по приоритету для хранения объектов типа Comparable.
- PriorityQueue(int *initialCapacity*, Comparator<? super E> *c*)
Конструирует очередь по приоритету и использует указанный компаратор для сортировки ее элементов.

Отображения

Множество — это коллекция, которая позволяет быстро находить существующий элемент. Но для того, чтобы найти такой элемент, нужно обладать его точной копией. Это не слишком распространенная операция поиска, поскольку, как правило, имеется некоторая ключевая информация, по которой требуется найти соответствующий элемент. Для этой цели предназначена структура данных типа отображения. В *отображении* хранятся пары “ключ–значение”. Следовательно, значение можно найти, если предоставить связанный с ним ключ. Например, можно составить и сохранить таблицу записей о сотрудниках, где ключами служат идентификаторы сотрудников, а значениями — объекты типа Employee.

В библиотеке коллекций Java предоставляются две реализации отображений общего назначения: классы HashMap и TreeMap. Оба класса реализуют интерфейс Map. Хеш-отображение типа HashMap хеширует ключи, а древовидное отображение типа TreeMap использует общий порядок ключей для организации поискового дерева. Функции хеширования или сравнения применяются *только* к ключам. Значения, связанные с ключами, не хешируются и не сравниваются.

Когда же использовать хеш-отображение, а когда древовидное отображение? Как и во множествах, хеширование выполняется немного быстрее, и поэтому хеш-отображение оказывается более предпочтительным, если не требуется перебирать ключи в отсортированном порядке. В приведенном ниже примере кода показано, как организовать хеш-отображение для хранения записей о сотрудниках.

```
Map<String, Employee> staff = new HashMap<>(); // объект типа HashMap,  
// реализующий интерфейс Map  
Employee harry = new Employee("Harry Hacker");  
staff.put("987-98-9996", harry);  
...
```

Всякий раз, когда объект вводится в отображение, следует непременно указать и его ключ. В данном случае ключом является символьная строка, а соответствующим значением — объект типа `Employee`. Чтобы извлечь объект, нужно использовать (а значит запомнить) ключ следующим образом:

```
String s = "987-98-9996";  
e = staff.get(s); // получить объект harry
```

Если в отображении отсутствуют данные по указанному ключу, метод `get()` возвращает пустое значение `null`. Ключи должны быть однозначными. Нельзя сохранить два значения по одинаковым ключам. Если дважды вызвать метод `put()` с одним и тем же ключом, то второе значение заменит первое. По существу, метод `put()` возвращает предыдущее значение, сохраненное по ключу, указанному в качестве его параметра.

Метод `remove()` удаляет элемент из отображения по заданному ключу. А метод `size()` возвращает количество элементов в отображении. В архитектуре коллекций само отображение не рассматривается в качестве коллекции. (В других архитектурах для построения структур данных отображение рассматривается в качестве коллекции *пар* или коллекции значений, индексированных ключами.) Тем не менее можно получить *представления* отдельного отображения — объекты, реализующие интерфейс `Collection` или один из его подчиненных интерфейсов.

Имеются три таких представления: множество ключей, коллекция (не множество) значений и множество пар “ключ–значение”. Ключи и пары “ключ–значение” образуют множество, потому что в отображении может присутствовать только по одной копии каждого ключа. Приведенные ниже методы возвращают эти три представления. (Элементы последнего множества являются объектами статического внутреннего класса `Map.Entry`.) Следует также иметь в виду, что `keySet` — это объект не класса `HashSet` или `TreeSet`, а некоторого другого класса, реализующего интерфейс `Set`, который расширяет интерфейс `Collection`.

```
Set<K> keySet()  
Collection<K> values()  
Set<Map.Entry<K, V>> entrySet()
```

Например, все ключи в отображении можно перечислить следующим образом:

```
Set<String> keys = map.keySet();  
for (String key : keys)  
{  
    сделать что-нибудь с ключом  
}
```



СОВЕТ. Если требуется просмотреть ключи и значения, то можно избежать поиска значений, перечисляя элементы отображения. Для этой цели служит следующий скелетный код:

```
for (Map.Entry<String, Employee> entry : staff.entrySet())  
{  
    String key = entry.getKey();  
    Employee value = entry.getValue();  
    сделать что-нибудь с ключом и значением  
}
```

Если вызывается метод `remove()` итератора, то на самом деле из отображения удаляется ключ и связанное с ним значение. Но ввести элемент в представление множества ключей нельзя. Ведь нет никакого смысла вводить ключ без ввода связанного с ним значения. Если попытаться вызвать метод `add()`, он сгенерирует исключение типа `UnsupportedOperationException`. На представление множества элементов отображения накладывается такое же ограничение, даже если операция ввода новой пары "ключ-значение" имеет принципиальный смысл.

В примере программы из листинга 13.6 демонстрируется применение отображения непосредственно в коде. Сначала в отображение вводится пара "ключ-значение". Затем из него удаляется один ключ, а следовательно, и связанное с ним значение. Далее изменяется значение, связанное с ключом, и вызывается метод `get()` для нахождения значения. И, наконец, выполняется перебор множества элементов отображения.

Листинг 13.6. Исходный код из файла `map/MapTest.java`

```
1 package map;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируется применение отображения
7  * с ключами типа String и значениями типа Employee
8  * @version 1.11 2012-01-26
9  * @author Cay Horstmann
10 */
11 public class MapTest
12 {
13     public static void main(String[] args)
14     {
15         Map<String, Employee> staff = new HashMap<>();
16         staff.put("144-25-5464", new Employee("Amy Lee"));
17         staff.put("567-24-2546", new Employee("Harry Hacker"));
18         staff.put("157-62-7935", new Employee("Gary Cooper"));
19         staff.put("456-62-5527", new Employee("Francesca Cruz"));
20
21         // вывести все элементы отображения
22         System.out.println(staff);
23
24         // удалить элемент из отображения
25         staff.remove("567-24-2546");
26
27         // заменить элемент отображения
28         staff.put("456-62-5527", new Employee("Francesca Miller"));
29
30         // найти значение
31         System.out.println(staff.get("157-62-7935"));
32
33         // перебрать все элементы отображения
34         for (Map.Entry<String, Employee> entry : staff.entrySet())
35         {
36             String key = entry.getKey();
37             Employee value = entry.getValue();
38             System.out.println("key=" + key + ", value=" + value);
39         }
40     }
41 }
```

java.util.Map<K, V> 1.2

- `V get(Object key)`
Получает значение, связанное с ключом, а возвращает объект, связанный с ключом, или же пустое значение `null`, если ключ не найден в отображении. В реализующих классах могут быть запрещены пустые ключи типа `null`.
- `V put(K key, V value)`
Размещает в отображении связь ключа со значением. Если ключ уже присутствует, новый объект заменяет старый, ранее связанный с тем же самым ключом. Этот метод возвращает старое значение ключа или же пустое значение `null`, если ключ ранее отсутствовал. В реализующих классах могут быть запрещены пустые ключи или значения `null`.
- `void putAll(Map<? extends K, ? extends V> entries)`
Вводит все элементы из указанного отображения в данное.
- `boolean containsKey(Object key)`
Возвращает логическое значение `true`, если ключ присутствует в отображении.
- `boolean containsValue(Object value)`
Возвращает логическое значение `true`, если значение присутствует в отображении.
- `Set<Map.Entry<K, V>> entrySet()`
Возвращает представление множества объектов типа `Map.Entry`, т.е. пар "ключ-значение" в отображении. Из этого множества можно удалять имеющиеся в нем элементы, но в него нельзя вводить новые элементы.
- `Set<K> keySet()`
Возвращает представление множества всех ключей в отображении. Из этого множества можно удалять имеющиеся в нем элементы, и в этом случае будут удалены ключи и связанные с ними значения, но в него нельзя вводить новые элементы.
- `Collection<V> values()`
Возвращает представление множества всех значений в отображении. Из этого множества можно удалять имеющиеся в нем элементы, и в этом случае будут удалены значения и связанные с ними ключи, но в него нельзя вводить новые элементы.

java.util.Map.Entry<K, V> 1.2

- `K getKey()`
- `V getValue()`
Возвращают ключ или значение из данного элемента отображения.
- `V setValue(V newValue)`
Заменяет старое значение на новое в связанном с ним отображении и возвращает старое значение.

java.util.HashMap<K, V> 1.2

- `HashMap()`
- `HashMap(int initialCapacity)`
- `HashMap(int initialCapacity, float loadFactor)`
Конструируют пустое хеш-отображение указанной емкости и с указанным коэффициентом загрузки (числовым значением в пределах от 0,0 до 1,0, определяющим процент заполнения хеш-таблицы, по достижении которого происходит повторное хеширование). Коэффициент загрузки по умолчанию равен 0,75.

java.util.TreeMap<K, V> 1.2

- `TreeMap(Comparator<? super K> c)`
Конструирует древовидное отображение, используя указанный компаратор для сортировки ключей.
- `TreeMap(Map<? extends K, ? extends V> entries)`
Конструирует древовидное отображение и вводит все элементы из указанного отображения.
- `TreeMap(SortedMap<? extends K, ? extends V> entries)`
Конструирует древовидное отображение и вводит все элементы из отсортированного отображения, используя тот же самый компаратор элементов, что и для отсортированного отображения.

java.util.SortedMap<K, V> 1.2

- `Comparator<? super K> comparator()`
Возвращает компаратор, используемый для сортировки ключей, или пустое значение `null`, если ключи сравниваются методом `compareTo()` из интерфейса `Comparable`.
- `K firstKey()`
- `K lastKey()`
Возвращают наименьший и наибольший ключи в отображении.

Классы специализированных множеств и отображений

В состав библиотеки коллекций Java входит ряд классов отображений для специальных нужд. Они будут вкратце описаны в этом разделе.

Слабые хеш-отображения

Класс `WeakHashMap` был разработан для решения одной интересной задачи. Что произойдет со значением, ключ которого не используется нигде больше в программе? Допустим, последняя ссылка на ключ исчезла. Следовательно, не остается никакого

способа сослаться на объект-значение. Но поскольку ни одна часть программы больше не содержит обращения к данному ключу, то и соответствующая пара “ключ–значение” не может быть удалена из отображения. Почему бы системе “сборки мусора” не удалить эту пару? Разве это не ее задача — удалять неиспользуемые объекты?

К сожалению, все не так просто. Система “сборки мусора” отслеживает *действующие* объекты. До тех пор, пока объект хеш-отображения действует, *все* группы в нем активны и не могут быть освобождены из памяти. Поэтому прикладная программа должна позаботиться об удалении неиспользуемых значений из долгосрочных отображений. С другой стороны, можно воспользоваться структурой данных типа WeakHashMap. Эта структура данных взаимодействует с системой “сборки мусора” для удаления пар “ключ–значение”, когда единственной ссылкой на ключ остается ссылка из элемента хеш-таблицы.

Поясним принцип действия этого механизма. В хеш-отображении типа WeakHashMap используются *слабые ссылки* для хранения ключей. Объект типа WeakHashMap содержит ссылку на другой объект (в данном случае ключ из хеш-таблицы). Объекты этого типа интерпретируются системой “сборки мусора” особым образом. Если система “сборки мусора” обнаруживает отсутствие ссылок на конкретный объект, то она, как правило, освобождает занятую им память. А если объект доступен только из хеш-отображения типа WeakHashMap, то система “сборки мусора” освобождает и его, но размещает в очереди слабую ссылку на него. В операциях, выполняемых над хеш-отображением типа WeakHashMap, эта очередь периодически проверяется на предмет появления новых слабых ссылок. Появление такой ссылки в очереди свидетельствует о том, что ключ больше не используется нигде, но по-прежнему хранится в коллекции. В таком случае связанный с ним элемент удаляется из хеш-отображения типа WeakHashMap.

Связные хеш-множества и отображения

В версии Java SE 1.4 появились классы LinkedHashMap и LinkedHashMap, запоминающие порядок ввода в них элементов. Таким образом, можно избежать кажущегося случайным порядка расположения элементов в хеш-таблице. По мере ввода элементов в таблицу они присоединяются к двунаправленному связному списку (рис. 13.9).

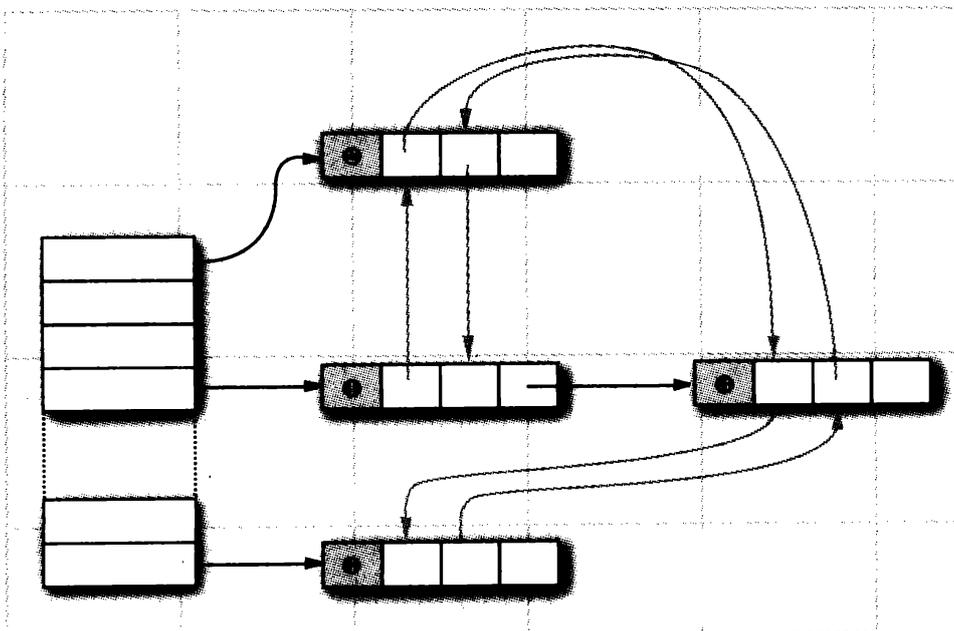


Рис. 13.9. Связная хеш-таблица

В качестве примера рассмотрим следующие элементы, введенные в отображение из листинга 13.6, но теперь это отображение типа `LinkedHashMap`:

```
Map staff = new LinkedHashMap();
staff.put("144-25-5464", new Employee("Amy Lee"));
staff.put("567-24-2546", new Employee("Harry Hacker"));
staff.put("157-62-7935", new Employee("Gary Cooper"));
staff.put("456-62-5527", new Employee("Francesca Cruz"));
```

Затем вызываемый итератор `staff.keySet().iterator()` перечисляет ключи в следующем порядке:

```
144-25-5464
567-24-2546
157-62-7935
456-62-5527
```

А вызываемый итератор `staff.values().iterator()` перечисляет значения в таком порядке:

```
Amy Lee
Harry Hacker
Gary Cooper
Francesca Cruz
```

Связное хеш-отображение позволяет сменить порядок ввода на *порядок доступа* для перебора его элементов. При каждом вызове метода `get()` или `put()` затрагиваемый элемент удаляется из его текущей позиции и вводится в *конец* связанного списка элементов. (Затрагивается только позиция в связанном списке элементов, а не в группах хеш-таблицы. Элемент всегда остается на том месте, которое соответствует хеш-коду его ключа.) Чтобы сконструировать такое хеш-отображение, достаточно сделать следующий вызов:

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

Порядок доступа удобен для реализации дисциплины кеширования с так называемым "наиболее давним использованием". Допустим, требуется сохранять часто используемые элементы в памяти и вводить менее часто используемые элементы из базы данных. Если нужный элемент не обнаруживается в таблице, а таблица уже почти заполнена, тогда можно получить итератор таблицы и удалить несколько первых элементов, которые он перечисляет. Ведь эти элементы использовались очень давно. Данный процесс можно даже автоматизировать. Для этого достаточно образовать подкласс, производный от класса `LinkedHashMap`, и переопределить в нем следующий метод:

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

И тогда при вводе нового элемента будет удаляться самый старый (`eldest`) элемент всякий раз, когда данный метод возвратит логическое значение `true`. Например, в следующем примере кода максимальный размер кеша поддерживается на уровне **100** элементов:

```
Map<K, V> cache = new
    LinkedHashMap<>(128, 0.75F, true)
    {
        protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
        {
            return size() > 100;
        }
    };
```

С другой стороны, можно проанализировать элемент `eldest`, чтобы решить, стоит ли его удалять. Например, можно проверить отметку времени, хранящуюся в этом элементе.

Перечислимые множества и отображения

В классе `EnumSet` эффективно реализуется множество элементов, относящихся к перечислимому типу. А поскольку у перечислимого типа ограниченное количество экземпляров, то класс `EnumSet` реализован внутренним образом в виде битовой последовательности. В каждом бите устанавливается 1, если соответствующее значение перечисления присутствует в множестве. У класса `EnumSet` отсутствуют открытые конструкторы. Для конструирования перечислимого множества используется статический фабричный метод, как показано ниже. А для видоизменения перечислимого множества типа `EnumSet` можно использовать обычные методы из интерфейса `Set`.

```
enum Weekday
{ MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday =
    EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
EnumSet<Weekday> mwf =
    EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);
```

Класс `EnumMap` реализует отображение с ключами, относящимися к перечислимому типу. Такое отображение реализуется просто и эффективно в виде массива значений. Для этого достаточно указать тип ключа в конструкторе следующим образом:

```
EnumMap<Weekday, Employee> personInCharge = new EnumMap<>(Weekday.class);
```



НА ЗАМЕТКУ! В документации на прикладной интерфейс API класса `EnumSet` можно обнаружить необычные параметры вроде `E extends Enum<E>`. Такое обозначение просто означает, что “E относится к перечислимому типу”. Перечислимые типы расширяют обобщенный класс `Enum`. Например, перечислимый тип `Weekday` расширяет класс `Enum<Weekday>`.

Хеш-отображения идентичности

В версии Java SE 1.4 появился еще один класс — `IdentityHashMap`, предназначенный для построения хеш-отображения идентичности, преследующего особые цели, когда хеш-значения ключей должны вычисляться не методом `hashCode()`, а методом `System.identityHashCode()`. В этом методе для вычисления хеш-кода, исходя из адреса объекта в памяти, используется метод `Object.hashCode()`. Кроме того, для сравнения объектов типа `IdentityHashMap` применяется операция `==`, а не вызов метода `equals()`.

Иными словами, разные объекты-ключи рассматриваются как отличающиеся, даже если они имеют одинаковое содержимое. Этот класс удобен для реализации алгоритмов обхода объектов (например, сериализации объектов), когда требуется отслеживать уже пройденные объекты.

java.util.WeakHashMap<K, V> 1.2

- `WeakHashMap()`
- `WeakHashMap(int initialCapacity)`
- `WeakHashMap(int initialCapacity, float loadFactor)`
Конструируют пустое хеш-отображение заданной емкости с указанным коэффициентом загрузки.

java.util.LinkedHashSet<E> 1.4

- `LinkedHashSet()`
- `LinkedHashSet(int initialCapacity)`
- `LinkedHashSet(int initialCapacity, float loadFactor)`
Конструируют пустое связанное хеш-множество заданной емкости с указанным коэффициентом загрузки.

java.util.LinkedHashMap<K, V> 1.4

- `LinkedHashMap()`
- `LinkedHashMap(int initialCapacity)`
- `LinkedHashMap(int initialCapacity, float loadFactor)`
- `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`
Конструируют пустое связанное хеш-отображение заданной емкости с указанным коэффициентом загрузки и упорядочением. Логическое значение `true` параметра `accessOrder` задает порядок доступа, а его логическое значение `false` — порядок ввода.
- `protected boolean removeEldestEntry(Map.Entry<K, V> eldest)`
Этот метод должен быть переопределен, чтобы возвращать логическое значение `true`, если требуется удалить самый старый элемент. Параметр `eldest` обозначает самый старый элемент, удаление которого рассматривается. Данный метод вызывается после того, как в отображение введен элемент. В его реализации по умолчанию возвращается логическое значение `false`, т.е. старые элементы по умолчанию не удаляются. Но этот метод можно переопределить для выборочного возврата логического значения `true`, если, например, самый старый элемент удовлетворяет определенным условиям или размеры отображения достигают определенной величины.

java.util.EnumSet<E extends Enum<E>> 5.0

- `static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)`
Возвращает множество, содержащее все значения заданного перечислимого типа.

- `static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)`
Возвращает пустое множество, способное хранить значения заданного перечислимого типа.
- `static <E extends Enum<E>> EnumSet<E> range(E from, E to)`
Возвращает множество, содержащее все значения от *from* до *to* включительно.
- `static <E extends Enum<E>> EnumSet<E> of(E value)`
- `static <E extends Enum<E>> EnumSet<E> of(E value, E... values)`
Возвращают множество, содержащее заданные значения.

java.util.EnumMap<K extends Enum<K>, V> 5.0

- `EnumMap(Class<K> keyType)`
Конструирует пустое отображение с ключами заданного типа.

java.util.IdentityHashMap<K, V> 1.4

- `IdentityHashMap()`
- `IdentityHashMap(int expectedMaxSize)`
Конструируют пустое хеш-отображение идентичности, емкость которого равна минимальной степени 2, превышающей величину $1.5 * \text{expectedMaxSize}$ (по умолчанию значение параметра `expectedMaxSize` равно 21).

java.lang.System 1.0

- `static int identityHashCode(Object obj) 1.1`
Возвращает хеш-код, вычисляемый методом `Object.hashCode()`, исходя из адреса памяти, по которому хранится объект, даже если в классе, к которому относится объект *obj*, переопределяется метод `hashCode()`.

Архитектура коллекций

Архитектура коллекций представляет собой набор классов, служащих основанием для расширения функциональных возможностей. Архитектура состоит из суперклассов, обладающих полезными функциональными возможностями, правилами и механизмами. Пользователь образует классы и подклассы, расширяющие полезные функциональные возможности архитектуры, чтобы не изобретать заново ее основные механизмы. Например, библиотека Swing — это архитектура для построения пользовательских интерфейсов.

Библиотека коллекций Java образует архитектуру классов коллекций. Она определяет множество интерфейсов и абстрактных классов, необходимых для реализации коллекций (рис. 13.10), а также предписывает определенные механизмы вроде

протокола итерации. Этими классами коллекций можно пользоваться, вообще ничего не зная об их архитектуре. Именно так и делалось в примерах кода из предыдущих разделов этой главы. Но если вы хотите реализовать обобщенные алгоритмы, пригодные для разнотипных коллекций, или же внедрить новый тип коллекции, то вам будет полезно уяснить внутреннее строение архитектуры коллекций в целом.

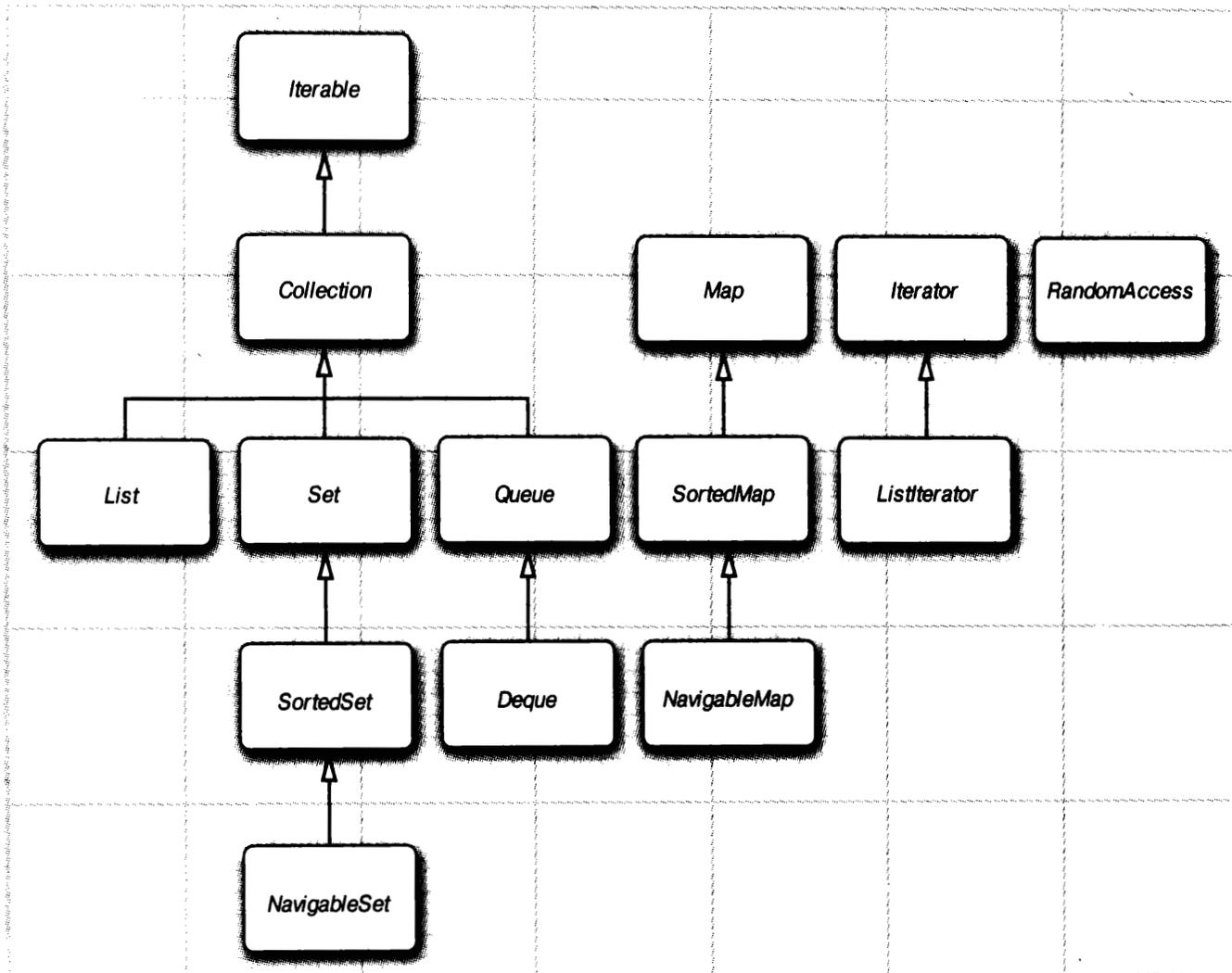


Рис. 13.10. Интерфейсы архитектуры коллекций

В рассматриваемой здесь архитектуре имеются два основополагающих интерфейса коллекций: `Collection` и `Map`. Для ввода элементов в коллекцию служит следующий метод:

```
boolean add(E element)
```

Но отображения содержат пары "ключ-значение", и поэтому для ввода этих пар вызывается метод `put()`, как показано ниже.

```
V put(K key, V value)
```

Для извлечения элементов из коллекции служит итератор, организующий обращение к ним. Но для извлечения значений из отображения вызывается метод `get()` следующим образом:

```
V get(K key)
```

Список представляет собой упорядоченную коллекцию. Элементы добавляются на определенной позиции в контейнере. Объект может быть расположен на нужной

позиции двумя способами: по целочисленному индексу и посредством итератора списка. В интерфейсе `List` определяются следующие методы для произвольного доступа:

```
void add(int index, E element)
E get(int index)
void remove(int index)
```

Как упоминалось ранее, интерфейс `List` предоставляет методы произвольного доступа независимо от того, насколько они эффективны для данной реализации. Чтобы избежать дорогостоящих (с точки зрения потребляемых вычислительных ресурсов) операций произвольного доступа, в версии Java SE 1.4 появился новый интерфейс `RandomAccess`. В этом интерфейсе отсутствуют методы, но с его помощью можно проверить, поддерживается ли в конкретной коллекции эффективный произвольный доступ, как показано ниже. Интерфейс `RandomAccess` реализуют классы `ArrayList` и `Vector`.

```
if (c instanceof RandomAccess)
{
    использовать алгоритм произвольного доступа
}
else
{
    использовать алгоритм последовательного доступа
}
```



НА ЗАМЕТКУ! С теоретической точки зрения имело бы смысл выделить интерфейс `Array`, расширяющий интерфейс `List` и объявляющий методы произвольного доступа. Если бы существовал отдельный интерфейс `Array`, то в тех алгоритмах, где требуется произвольный доступ, могли бы использоваться параметры интерфейса `Array`, и тогда их нельзя было бы случайно применить к коллекциям с медленным произвольным доступом. Но разработчики архитектуры коллекций решили не определять отдельный интерфейс, поскольку стремились сохранить небольшое количество интерфейсов в библиотеке. К тому же они не хотели ограничивать свободу действий программистов, которые вольны применять к связным спискам алгоритмы произвольного доступа, но в то же время они должны отдавать себе отчет, что за это придется расплачиваться снижением производительности.

В интерфейсе `ListIterator` определяется следующий метод для ввода элемента до позиции итератора:

```
void add(E element)
```

Для получения и удаления элементов на определенной позиции в коллекции служат методы `next()` и `remove()` из интерфейса `Iterator`. Интерфейс `Set` подобен интерфейсу `Collection`, но поведение его методов определено более строго. В частности, метод `add()` должен отвергать дубликаты во множестве. Метод `equals()` должен быть определен таким образом, чтобы два множества считались одинаковыми, если они содержат одни и те же элементы, но не обязательно в одинаковом порядке. А метод `hashCode()` должен быть определен таким образом, чтобы два множества с одинаковыми элементами порождали один и тот же хеш-код.

Зачем же объявлять отдельные интерфейсы, если сигнатуры их методов совпадают? В принципе не все коллекции являются множествами. Поэтому наличие интерфейса `Set` дает программистам возможность писать методы, принимающие только множества.

В интерфейсах `SortedSet` и `SortedMap` раскрывается объект-компаратор, применяемый для сортировки, а также определяются методы для получения представлений подмножеств коллекций. Об этих представлениях речь пойдет в следующем разделе.

И, наконец, в версии Java SE 6 внедрены интерфейсы `NavigableSet` и `NavigableMap`, содержащие дополнительные методы для поиска и обхода отсортированных множеств и отображений. (В идеальном случае эти методы должны быть просто включены в состав интерфейсов `SortedSet` и `SortedMap`.) Эти интерфейсы реализуются в классах `TreeSet` и `TreeMap`.

А теперь перейдем от интерфейсов к реализующим их классам. Как упоминалось ранее, в интерфейсах коллекций предусмотрено относительно немного методов, которые нетрудно реализовать на основе более фундаментальных методов. Многие реализации рутинных операций предоставляются в следующих абстрактных классах:

```
AbstractCollection
AbstractList
AbstractSequentialList
AbstractSet
AbstractQueue
AbstractMap
```

Если же вы реализуете свой собственный класс коллекции, то захотите расширить один из этих классов таким образом, чтобы можно было предложить свои реализации рутинных операций. В библиотеке коллекций Java предоставляются следующие конкретные классы:

```
LinkedList
ArrayList
ArrayDeque
HashSet
TreeSet
PriorityQueue
HashMap
TreeMap
```

На рис. 13.11 схематически показаны отношения между этими классами.

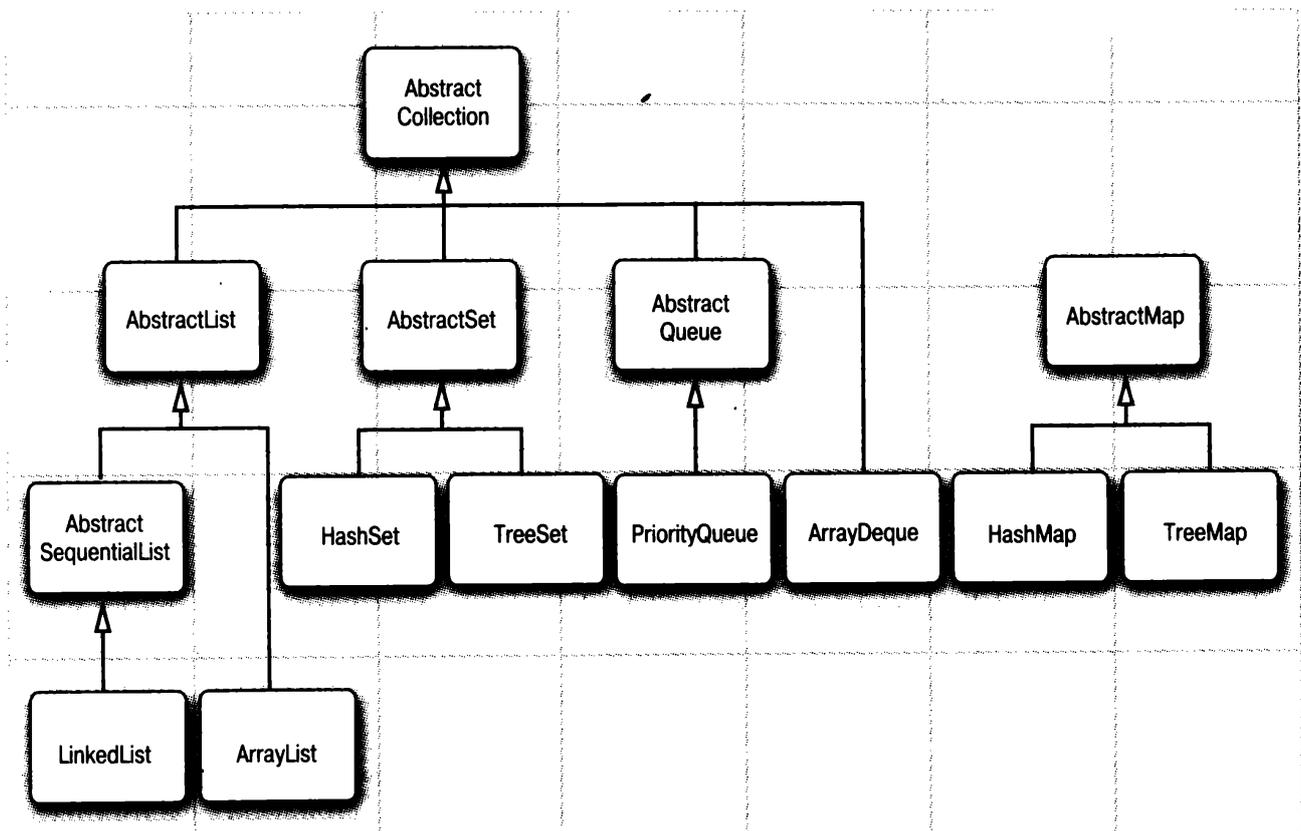


Рис. 13.11. Классы из архитектуры коллекций

И, наконец, по-прежнему поддерживаются несколько “унаследованных” контейнерных классов, впервые появившихся в первой версии Java, когда еще не было нынешней архитектуры коллекций. Эти классы перечислены ниже.

Vector
Stack
Hashtable
Properties

Все они были интегрированы в архитектуру коллекций, как показано на рис. 13.12. Они будут рассмотрены далее в этой главе.

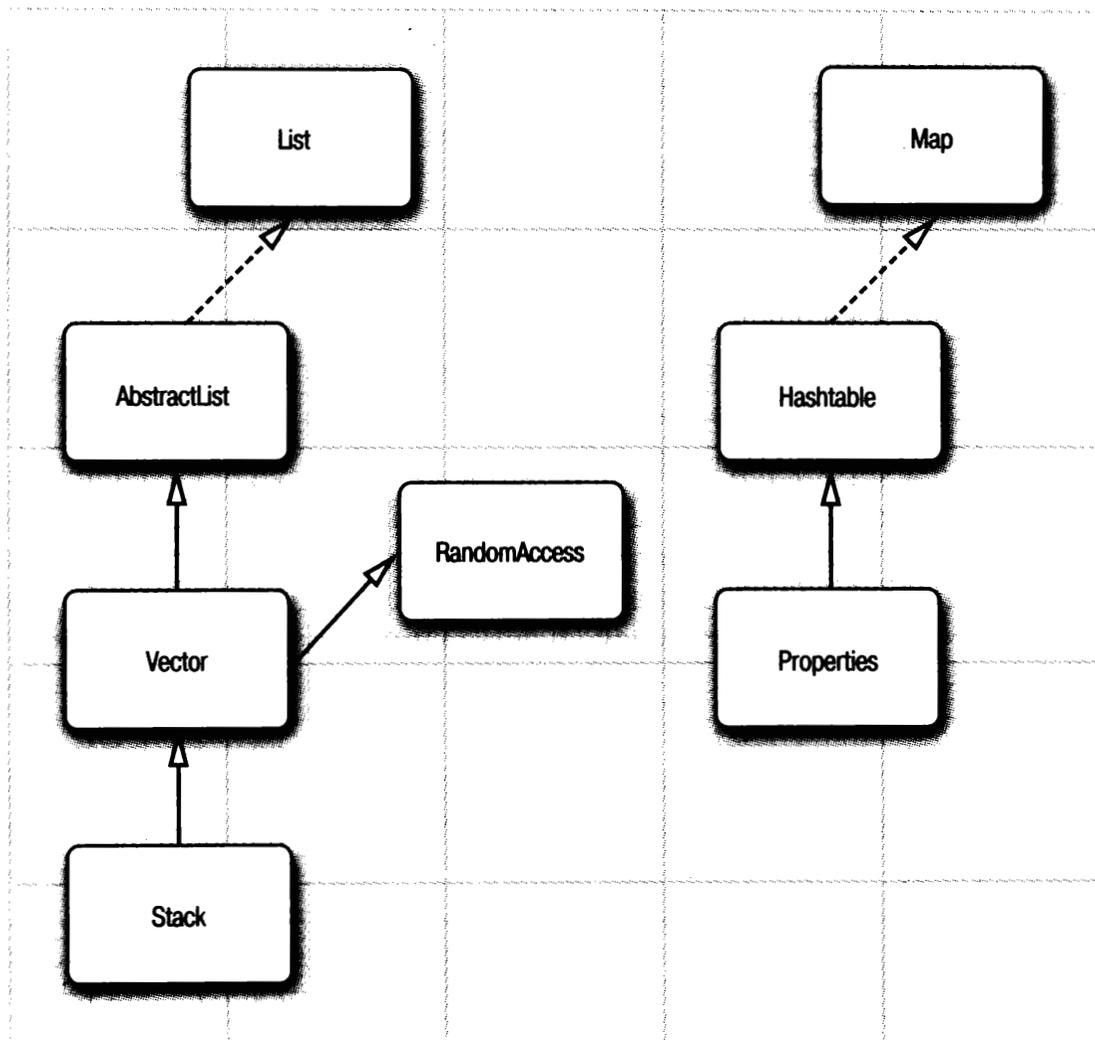


Рис. 13.12. Унаследованные классы в архитектуре коллекций

Представления и оболочки

Глядя на рис. 13.10 и 13.11, можно подумать, что излишне иметь так много интерфейсов и абстрактных классов для реализации столь скромного количества конкретных классов коллекций. Но эти рисунки не отражают всей картины. Используя *представления*, можно получить другие объекты, реализующие интерфейсы Collection или Map. Характерным тому примером служит упоминавшийся ранее метод `keySet()` из классов отображений. На первый взгляд данный метод создает новое множество, заполняет его всеми ключами из отображения и возвращает его. Но это не совсем так. Напротив, метод `keySet()` возвращает объект класса, который реализует интерфейс Set и методы которого манипулируют исходным отображением.

Такая коллекция называется *представлением*. У методики представлений имеется ряд полезных применений в архитектуре коллекций. Эти применения будут обсуждаться в последующих подразделах.

Легковесные оболочки коллекций

Статический метод `asList()` из класса `Arrays` возвращает объект типа `List` — оболочку, в которую заключен простой массив `Java`. Этот метод позволяет передать массив методу, принимающему список или коллекцию в качестве своего аргумента, как показано в приведенном ниже примере кода.

```
Card[] cardDeck = new Card[52];
. . .
List<Card> cardList = Arrays.asList(cardDeck);
```

Возвращаемый объект *не* относится к типу `ArrayList`. Это объект представления с методами `get()` и `set()`, имеющими доступ к положенному в его основание массиву. А все методы, которые обычно изменяют размер массива (например, методы `add()` и `remove()` соответствующего итератора), генерируют исключение типа `UnsupportedOperationException`.

В версии `Java SE 5.0` метод `asList()` был объявлен с переменным количеством аргументов. Вместо всего массива ему можно передавать отдельные элементы, как, например, в следующей строке кода:

```
List<String> names = Arrays.asList("Amy", "Bob", "Carl");
```

В результате вызова метода `Collections.nCopies(n, anObject)` возвращается неизменяемый объект класса, реализующего интерфейс `List`. При этом создается впечатление, будто это коллекция, насчитывающая `n` элементов, каждый из которых похож на объект `anObject`. Например, в результате следующего вызова создается коллекция типа `List`, содержащая **100** одинаковых символьных строк `"DEFAULT"`:

```
List<String> settings = Collections.nCopies(100, "DEFAULT");
```

Хранение такого списка обходится очень дешево с точки зрения потребляемых вычислительных ресурсов. И это характерный пример разумного применения методики представлений.



НА ЗАМЕТКУ! Класс `Collections` содержит немало служебных методов с параметрами или возвращаемыми значениями в виде коллекций. Не путайте его с интерфейсом `Collection`.

А в результате вызова метода `Collections.singleton(anObject)` возвращается объект представления, реализующий интерфейс `Set` (в отличие от метода `nCopies()`, который возвращает коллекцию типа `List`). Возвращаемый объект реализует неизменяемое одноэлементное множество без издержек, характерных для структуры данных. Методы `singletonList()` и `singletonMap()` ведут себя аналогичным образом.

Поддиапазоны

Для многих коллекций можно формировать представления поддиапазонов. Допустим, имеется список `staff`, из которого требуется извлечь элементы с **10-го по 19-й**. С этой целью следует вызвать метод `subList()`, чтобы получить представление поддиапазона списка, как показано ниже.

```
List group2 = staff.subList(10, 20);
```

Первый индекс включается в выборку элементов из списка, а второй исключается. Аналогичным образом действуют параметры метода `substring()` из класса `String`.

Над полученным в итоге поддиапазоном элементов списка можно выполнить любые операции, а их результаты автоматически отразятся на всем списке в целом. Например, весь поддиапазон элементов списка можно удалить, как показано ниже. В итоге выбранные элементы автоматически удаляются из списка `staff`, а поддиапазон `group2` становится пустым.

```
group2.clear(); // сократить штат
```

Для формирования поддиапазонов отсортированных множеств и отображений задается порядок сортировки, а не позиции элементов. С этой целью в интерфейсе `SortedSet` объявляются три метода:

```
SortedSet<E> subSet(E from, E to)
SortedSet<E> headSet(E to)
SortedSet<E> tailSet(E from)
```

Все они возвращают подмножества всех элементов, которые больше или равны значению параметра `from` и меньше значения параметра `to`. А для отсортированных отображений аналогичные методы возвращают представления отображений, состоящие из всех элементов, в которых ключи находятся в заданном поддиапазоне. Все эти методы перечислены ниже.

```
SortedMap<K, V> subMap(K from, K to)
SortedMap<K, V> headMap(K to)
SortedMap<K, V> tailMap(K from)
```

Интерфейс `NavigableSet`, внедренный в версии Java SE 6, предоставляет больше возможностей для контроля над подобными операциями с поддиапазонами элементов коллекций. Так, в приведенных ниже методах из этого интерфейса можно указать, следует ли включать в выборку элементов из коллекции заданные границы поддиапазона.

```
NavigableSet<E> subSet(
    E from, boolean fromInclusive, E to, boolean toInclusive)
NavigableSet<E> headSet(E to, boolean toInclusive)
NavigableSet<E> tailSet(E from, boolean fromInclusive)
```

Немодифицируемые представления

В состав класса `Collections` входят методы, производящие *немодифицируемые представления* коллекций. Такие представления вводят динамическую проверку в существующие коллекции. Если в ходе такой проверки обнаруживается попытка видоизменить коллекцию, генерируется исключение и коллекция остается нетронутой. Получить немодифицируемые представления можно следующими шестью методами:

```
Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableMap
Collections.unmodifiableSortedMap
```

Каждый из этих методов определен для работы с интерфейсом. Так, метод `Collections.unmodifiableList()` работает с классом `ArrayList`, `LinkedList` или любым другим классом, реализующим интерфейс `List`. Допустим, некоторой части

прикладной программы требуется предоставить возможность просматривать, но не изменять содержимое коллекции. В приведенном ниже фрагменте кода показано, как это можно сделать.

```
List<String> staff = new LinkedList<>();
. . .
lookAt(Collections.unmodifiableList(staff));
```

Метод `Collections.unmodifiableList()` возвращает объект класса, реализующего интерфейс `List`. Его методы доступа извлекают значения из коллекции `staff`. Конечно, метод `lookAt()` может вызывать все методы из интерфейса `List`, а не только методы доступа. Но все методы, изменяющие коллекцию, например `add()`, переопределены таким образом, чтобы немедленно генерировать исключение типа `UnsupportedOperationException` вместо передачи вызова базовой коллекции.

Немодифицируемое представление не делает саму коллекцию немодифицируемой. Коллекцию можно по-прежнему видоизменить по ее исходной ссылке (в данном случае — `stuff`). А модифицирующие методы можно вызывать для отдельных элементов коллекции.

Представления заключают в оболочку *интерфейс*, а не конкретный объект коллекции, и поэтому доступ требуется лишь к тем методам, которые определены в интерфейсе. Например, в состав класса `LinkedList` входят служебные методы `addFirst()` и `addLast()`, не являющиеся частью интерфейса `List`. Но эти методы недоступны через немодифицируемое представление.



ВНИМАНИЕ! Метод `unmodifiableCollection()` (как, впрочем, и методы `synchronizedCollection()` и `checkedCollection()`, о которых речь пойдет далее) возвращает коллекцию, в которой метод `equals()` не вызывает одноименный метод из базовой коллекции. Вместо этого он наследует метод `equals()` из класса `Object`, который просто проверяет объекты на равенство. Если просто преобразовать множество или список в коллекцию, то проверить ее содержимое на равенство не удастся.

Представление действует подобным образом, потому что проверка на равенство четко определена на данном уровне иерархии. Аналогичным образом в представлениях интерпретируется и метод `hashCode()`. Но в классах объектов, возвращаемых методами `unmodifiableSet()` и `unmodifiableList()`, используются методы `equals()` и `hashCode()` из базовой коллекции.

Синхронизированные представления

Если обращение к коллекции происходит из нескольких потоков, то нужно каким-то образом исключить ее непреднамеренное повреждение. Было бы, например, губительно, если бы один поток попытался ввести элемент в хеш-таблицу в тот момент, когда в другом выполнялось бы повторное хеширование ее элементов.

Вместо реализации потокобезопасных классов разработчики библиотеки коллекций воспользовались механизмом представлений, чтобы сделать потокобезопасными обычные коллекции. Например, статический метод `synchronizedMap()` из класса `Collections` может превратить любое отображение в объект типа `Map` с синхронизированными методами доступа следующим образом:

```
Map<String, Employee> map =
    Collections.synchronizedMap(new HashMap<String, Employee>());
```

После этого к объекту `map` можно обращаться из нескольких потоков. Такие методы, как `get()` и `put()`, сериализованы. Это означает, что каждый вызов метода должен

полностью завершаться до того, как другой поток сможет его вызвать. Вопросы синхронизированного доступа к структурам данных подробнее обсуждаются в главе 14.

Проверяемые представления

В версии Java SE 5.0 было внедрено множество *проверяемых* представлений, задуманных как средство, поддерживающее отладку ошибок, сопутствующих применению обобщенных типов в прикладном коде. Как пояснялось в главе 12, существует возможность незаконно внедрить в обобщенную коллекцию элементы неверного типа. Например, в приведенном ниже фрагменте кода ошибочный вызов метода `add()` не обнаруживается.

```
ArrayList<String> strings = new ArrayList<>();
ArrayList rawList = strings; // ради совместимости с унаследованным кодом
                             // при компиляции этой строки кода выдается
                             // только предупреждение, но не ошибка
rawList.add(new Date());    // теперь символьные строки содержат
                             // объект типа Date!
```

Вместо этого возникнет исключение, когда при последующем вызове метода `get()` будет сделана попытка привести результат к типу `String`. Проверяемое представление позволяет обнаружить этот недостаток в коде. Для этого сначала определяется безопасный список:

```
List<String> safeStrings = Collections.checkedList(strings, String.class);
```

Затем в методе представления `add()` проверяется принадлежность объекта, вводимого в коллекцию, заданному классу. И если обнаружится несоответствие, то немедленно генерируется исключение типа `ClassCastException`, как показано ниже. Преимущество такого подхода заключается в том, что ошибка произойдет в том месте кода, где ее можно обнаружить и обработать.

```
ArrayList rawList = safeStrings;
rawList.add(new Date()); // проверяемый список генерирует
                        // исключение типа ClassCastException
```



ВНИМАНИЕ! Проверяемые представления ограничиваются динамическими проверками, которые способна выполнить виртуальная машина. Так, если имеется списочный массив типа `ArrayList<Pair<String>>`, его нельзя защитить от ввода элемента типа `Pair<Date>`, поскольку виртуальной машине ничего неизвестно о единственном базовом классе `Pair`.

О необязательных операциях

Обычно на представление накладывается определенное ограничение: оно может быть доступно только для чтения, может не допускать изменений размера или поддерживать удаление, но запрещать ввод элементов, как это имеет место для представления ключей в отображении. Ограниченное таким образом представление генерирует исключение типа `UnsupportedOperationException`, если попытаться выполнить неразрешенную операцию.

В документации на API для интерфейсов коллекций и итераторов многие методы описаны как “необязательные операции”. На первый взгляд это противоречит самому понятию интерфейса. Разве не в том назначение интерфейсов, чтобы объявлять методы, которые класс *обязан* реализовать? На самом деле такая организация неудовлетворительна с теоретической точки зрения. Возможно, лучше было бы разработать отдельные

интерфейсы для представлений “только для чтения” и представлений, которые не могут изменять размер коллекции. Но это привело бы к значительному увеличению количества интерфейсов, что разработчики библиотеки сочли неприемлемым.

Следует ли распространять методику “необязательных” методов на собственные проекты? Вряд ли. Даже если коллекции используются часто, стиль программирования для их реализации не типичен для других предметных областей. Разработчикам библиотеки классов коллекций пришлось удовлетворить ряд жестких и противоречивых требований. Пользователи хотят, чтобы библиотеку было легко усвоить и удобно использовать, чтобы она была полностью обобщенной, защищенной от неумелого обращения и такой же эффективной, как и разработанные вручную алгоритмы. Удовлетворить всем этим требованиям одновременно или даже приблизиться к этой цели практически невозможно. Но в своей практике программирования на Java вы редко столкнетесь с таким суровым рядом ограничений. Тем не менее вы должны быть готовы находить решения, которые не опираются на такую крайнюю меру, как применение “необязательных” интерфейсных операций.

java.util.Collections 1.2

- `static <E> Collection unmodifiableCollection(Collection<E> c)`
- `static <E> List unmodifiableList(List<E> c)`
- `static <E> Set unmodifiableSet(Set<E> c)`
- `static <E> SortedSet unmodifiableSortedSet(SortedSet<E> c)`
- `static <K, V> Map unmodifiableMap(Map<K, V> c)`
- `static <K, V> SortedMap unmodifiableSortedMap(SortedMap<K, V> c)`
Конструируют представления коллекций, в которых модифицирующие методы генерируют исключение типа `UnsupportedOperationException`.
- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
- `static <E> List synchronizedList(List<E> c)`
- `static <E> Set synchronizedSet(Set<E> c)`
- `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`
Конструируют представления коллекций с синхронизированными методами.
- `static <E> Collection checkedCollection(Collection<E> c, Class<E> elementType)`
- `static <E> List checkedList(List<E> c, Class<E> elementType)`
- `static <E> Set checkedSet(Set<E> c, Class<E> elementType)`
- `static <E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)`
- `static <K, V> Map checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)`

- `static <K, V> SortedMap checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)`

Конструируют представления, в которых методы генерируют исключение типа `ClassCastException` при попытке ввести элемент неверного типа в коллекцию.

- `static <E> List<E> nCopies(int n, E value)`
- `static <E> Set<E> singleton(E value)`

Конструируют представление объекта в виде немодифицируемого списка из n одинаковых элементов или же в виде множества с единственным элементом.

java.util.Arrays 1.2

- `static <E> List<E> asList(E... array)`

Возвращает представление списка элементов массива, который является модифицируемым, но с неизменяемым размером.

java.util.List<E> 1.2

- `List<E> subList(int firstIncluded, int firstExcluded)`

Возвращает представление списка элементов в заданном диапазоне позиций.

java.util.SortedSet<E> 1.2

- `SortedSet<E> subSet(E firstIncluded, E firstExcluded)`
- `SortedSet<E> headSet(E firstExcluded)`
- `SortedSet<E> tailSet(E firstIncluded)`

Возвращают представление элементов отсортированного множества в заданном диапазоне.

java.util.NavigableSet<E> 6

- `NavigableSet<E> subSet(E from, boolean fromIncluded, E to, boolean toIncluded)`
- `NavigableSet<E> headSet(E to, boolean toIncluded)`
- `NavigableSet<E> tailSet(E from, boolean fromIncluded)`

Возвращают представление элементов множества в заданном диапазоне. Параметры типа `boolean` определяют, следует ли включать в представление заданные границы диапазона.

java.util.SortedMap<K, V> 1.2

- SortedMap<K, V> subMap(K *firstIncluded*, K *firstExcluded*)
- SortedMap<K, V> headMap(K *firstExcluded*)
- SortedMap<K, V> tailMap(K *firstIncluded*)

Возвращают представление элементов отсортированного отображения в заданном диапазоне.

java.util.NavigableMap<K, V> 6

- NavigableMap<K, V> subMap(K *from*, boolean *fromIncluded*, K *to*, boolean *toIncluded*)
- NavigableMap<K, V> headMap(K *from*, boolean *fromIncluded*)
- NavigableMap<K, V> tailMap(K *to*, boolean *toIncluded*)

Возвращают представление элементов отображения в заданном диапазоне. Параметры типа `boolean` определяют, следует ли включать в представление заданные границы диапазона.

Групповые операции

В большинстве приведенных до сих пор примеров кода для последовательного обхода элементов коллекции использовался итератор. Но зачастую можно избежать итераций, применяя групповые операции. Допустим, требуется найти *пересечение* двух множеств, т.е. элементы, общие для обоих множеств. Для хранения результата данной операции сначала создается новое множество:

```
Set<String> result = new HashSet<>(a);
```

В данном случае используется тот факт, что у каждой коллекции имеется свой конструктор, параметром которого является другая коллекция, содержащая неинициализированные значения. Затем вызывается метод `retainAll()` следующим образом:

```
result.retainAll(b);
```

Он оставляет в текущей коллекции только те элементы, которые имеются в коллекции `b`. В итоге пересечение двух множеств получается без всякого программирования цикла их поочередного обхода.

Этот принцип можно распространить и на групповые операции над *представлениями*. Допустим, имеется отображение, связывающее идентификаторы сотрудников с объектами, описывающими сотрудников, а также множество идентификаторов сотрудников, которые должны быть уволены, как показано ниже.

```
Map<String, Employee> staffMap = . . .;
Set<String> terminatedIDs = . . .;
```

Достаточно сформировать множество ключей и удалить идентификаторы всех увольняемых сотрудников, как показано в приведенной ниже строке кода. Это множество ключей является представлением отображения, и поэтому ключи и имена соответствующих сотрудников автоматически удаляются из отображения.

```
staffMap.keySet().removeAll(terminatedIDs);
```

Используя представление поддиапазона, можно ограничить групповые операции подписками и подмножествами. Допустим, требуется ввести первые 10 элементов списка в другой контейнер. Для этого сначала формируется подсписок из первых 10 элементов:

```
relocated.addAll(staff.subList(0, 10));
```

Поддиапазон также может стать целью модифицирующей групповой операции, как показано в следующей строке кода:

```
staff.subList(0, 10).clear();
```

Взаимное преобразование коллекций и массивов

Значительная часть прикладного интерфейса API платформы Java была разработана до появления архитектуры коллекций, поэтому рано или поздно традиционные массивы придется преобразовать в более современные коллекции. Так, если имеется массив, который требуется преобразовать в коллекцию, для этой цели служит обертка `Arrays.asList`, как показано в приведенном ниже примере кода.

```
String[] values = . . . ;  
HashSet<String> staff = new HashSet<>(Arrays.asList(values));
```

Получить массив из коллекции немного сложнее. Конечно, для этого можно воспользоваться методом `toArray()` следующим образом:

```
Object[] values = staff.toArray();
```

Но в итоге будет получен массив *объектов*. Даже если известно, что коллекция содержит объекты определенного типа, их все равно нельзя привести к нужному типу, как показано ниже.

```
String[] values = (String[]) staff.toArray(); // ОШИБКА!
```

Массив, возвращаемый методом `toArray()`, создан как массив типа `Object[]`, а этот тип изменить нельзя. Вместо этого следует воспользоваться приведенным ниже вариантом метода `toArray()`, передав ему массив нулевой длины и требуемого типа. Возвращаемый в итоге массив окажется *того же самого типа*.

```
String[] values = staff.toArray(new String[0]);
```

Если есть желание, можно сразу сконструировать массив нужного размера следующим образом:

```
staff.toArray(new String[staff.size()]);
```

В этом случае новый массив не создается.



НА ЗАМЕТКУ! У вас может возникнуть следующий вопрос: почему бы просто не передать объект типа `Class` (например, `String.class`) методу `toArray()`? Но ведь этот метод несет на себе "двойную нагрузку", заполняя существующий массив (если он достаточно длинный) и создавая новый.

Алгоритмы

Обобщенные интерфейсы коллекций дают огромное преимущество: конкретный алгоритм достаточно реализовать лишь один раз. В качестве примера рассмотрим

простой алгоритм вычисления наибольшего элемента в коллекции. Традиционно он реализуется в цикле. Ниже показано, как обнаружить самый большой элемент в массиве традиционным способом.

```
if (a.length == 0) throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
    if (largest.compareTo(a[i]) < 0)
        largest = a[i];
```

Разумеется, чтобы найти наибольший элемент в списочном массиве, придется написать код немного иначе:

```
if (v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
    if (largest.compareTo(v.get(i)) < 0)
        largest = v.get(i);
```

А как насчет связанного списка? Для связанного списка не существует эффективного произвольного доступа, но можно использовать итератор, как показано ниже.

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
    T next = iter.next();
    if (largest.compareTo(next) < 0)
        largest = next;
}
```

Все эти циклы писать довольно утомительно, а кроме того, они чреваты ошибками. Нужно постоянно проверять себя: не допущена ли ошибка выхода за допустимые пределы, правильно ли будут выполняться эти циклы с пустыми коллекциями, а как насчет контейнеров с одним элементом? Вряд ли захочется тестировать и отлаживать такой код каждый раз, но и желания реализовывать уйму методов вроде приведенных ниже также не возникнет.

```
static <T extends Comparable> T max(T[] a)
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

И здесь на помощь приходят интерфейсы коллекций. Подумайте о *минимальном* интерфейсе коллекции, который понадобится для эффективной реализации алгоритма. Произвольный доступ с помощью методов `get()` и `set()` стоит рангом выше, чем простая итерация. Как следует из приведенного выше примера вычисления наибольшего элемента в связанном списке, произвольный доступ совсем не обязателен для решения подобной задачи. Вычисление максимума может быть выполнено путем простой итерации по элементам коллекции. Следовательно, метод `max()` можно воплотить в коде таким образом, чтобы он принимал объект *любого* класса, реализующего интерфейс `Collection`, как показано ниже.

```
public static <T extends Comparable> T max(Collection<T> c)
{
    if (c.isEmpty()) throw new NoSuchElementException();
    Iterator<T> iter = c.iterator();
    T largest = iter.next();
    while (iter.hasNext())
```

```
{
    T next = iter.next();
    if (largest.compareTo(next) < 0)
        largest = next;
}
return largest;
}
```

Теперь единственным обобщенным методом можно вычислить максимум в связанном списке, в списочном или простом массиве. Это весьма эффективный алгоритм. На самом деле в стандартной библиотеке C++ имеются десятки полезных алгоритмов, каждый из которых оперирует обобщенной коллекцией. Библиотека Java не настолько богата, но в ней все же имеются самые основные алгоритмы: сортировки, двоичного поиска, а также некоторые служебные алгоритмы.

Сортировка и перетасовка

Ветераны программирования иногда вспоминают о том, как им приходилось пользоваться перфокартами и программировать алгоритмы сортировки вручную. Ныне алгоритмы сортировки уже вошли в состав стандартных библиотек большинства языков программирования, и в этом отношении Java не является исключением. Так, метод `sort()` из класса `Collections` сортирует коллекцию, реализующую интерфейс `List`, следующим образом:

```
List<String> staff = new LinkedList<>();
// заполнить коллекцию
Collections.sort(staff);
```

В этом методе предполагается, что элементы списка реализуют интерфейс `Comparable`. Если же требуется отсортировать список каким-то другим способом, вы можете передать объект типа `Comparator` в качестве второго параметра конструктора. (Компараторы рассматривались в разделе “Сравнение объектов” ранее в этой главе.) Ниже показано, как отсортировать список элементов.

```
Comparator<Item> itemComparator = new
Comparator<Item>()
{
    public int compare(Item a, Item b)
    {
        return a.partNumber - b.partNumber;
    }
};
Collections.sort(items, itemComparator);
```

Если требуется отсортировать список по *убывающей*, следует воспользоваться служебным статическим методом `Collections.reverseSort()`. Он возвращает компаратор, который в свою очередь возвращает результат вызова `b.compareTo(a)`. Например, в приведенной ниже строке кода элементы списка `staff` сортируются в обратном порядке, который задается методом `compareTo()` для типа элементов списка.

```
Collections.sort(staff, Collections.reverseOrder())
```

Аналогично в следующей строке кода изменяется на обратный порядок сортировки, задаваемый компаратором `itemComparator`:

```
Collections.sort(items, Collections.reverseOrder(itemComparator))
```

Вас может заинтересовать, каким образом метод `sort()` сортирует список. Если проанализировать алгоритм сортировки, рассматриваемый в специальной литературе по алгоритмам, то он обычно поясняется на примере обычных массивов с произвольным доступом к элементам. Но ведь произвольный доступ к элементам списка неэффективен. Списки лучше сортировать, используя алгоритм сортировки слиянием (см., например, книгу *Алгоритмы на С++* Роберта Седжвика (пер. с англ., ИД "Вильямс", 2011, ISBN 978-5-8459-1650-1). Но в реализации на Java этого не делается. Напротив, все элементы выводятся в массив, который затем сортируется с помощью другой разновидности сортировки слиянием, после чего отсортированная последовательность копируется обратно в список.

Алгоритм сортировки слиянием, применяемый в библиотеке коллекций, немного медленнее быстрой сортировки, традиционно выбираемой для алгоритмов сортировки общего назначения. Но у него имеется следующее важное преимущество: он *устойчив*, т.е. не меняет местами равнозначные элементы. А зачем вообще беспокоиться о порядке следования равнозначных элементов? Рассмотрим распространенный случай. Допустим, имеется список сотрудников, который уже отсортирован по их Ф.И.О., а теперь их нужно отсортировать по зарплате. Как же будут отсортированы сотрудники с одинаковой зарплатой? При устойчивой сортировке упорядочение по Ф.И.О. сохраняется. Иными словами, в конечном итоге получится список, отсортированный сначала по зарплате, а затем по Ф.И.О. сотрудников.

В коллекциях не нужно реализовывать все "необязательные" методы, поэтому все методы, принимающие коллекции в качестве параметров, должны указывать, когда безопасно передавать коллекцию алгоритму. Например, совершенно очевидно, что вряд ли стоит передавать немодифицируемый список алгоритму сортировки. Какие же списки можно передавать? Согласно документации, список должен быть модифицируемым, но его размер не должен быть изменяемым. Ниже поясняется, что все это означает.

- Список является *модифицируемым*, если он поддерживает метод `set()`.
- Список имеет *изменяемый размер*, если он поддерживает методы `add()` и `remove()`.

В классе `Collections` реализован алгоритм перетасовки и соответствующий метод `shuffle()`, который выполняет противоположную сортировке задачу, изменяя случайным образом порядок расположения элементов в списке, как показано в приведенном ниже примере кода.

```
ArrayList<Card> cards = . . . ;
Collections.shuffle(cards);
```

Если предоставить список, который не реализует интерфейс `RandomAccess`, то метод `shuffle()` скопирует все его элементы в массив, перетасует этот массив, после чего скопирует перетасованные элементы обратно в список.

В примере программы из листинга 13.7 списочный массив заполняется 49 объектами типа `Integer`, содержащими числа от 1 до 49. Затем они перетасовываются случайным образом в списке, откуда далее выбираются первые 6 значений. И, наконец, выбранные значения сортируются и выводятся.

Листинг 13.7. Исходный код из файла `shuffle/ShuffleTest.java`

```
1 package shuffle;
2
3 import java.util.*;
4 /**
5  * В этой программе демонстрируются алгоритмы
6  * произвольной перетасовки и сортировки
7  * @version 1.11 2012-01-26
8  * @author Cay Horstmann
9  */
10 public class ShuffleTest
11 {
12     public static void main(String[] args)
13     {
14         List<Integer> numbers = new ArrayList<>();
15         for (int i = 1; i <= 49; i++)
16             numbers.add(i);
17         Collections.shuffle(numbers);
18         List<Integer> winningCombination = numbers.subList(0, 6);
19         Collections.sort(winningCombination);
20         System.out.println(winningCombination);
21     }
22 }
```

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> void sort(List<T> elements)`
- `static <T> void sort(List<T> elements, Comparator<? super T> c)`
Сортируют элементы в списке, используя алгоритм устойчивой сортировки. Выполнение алгоритма гарантируется за время $O(n \log n)$, где n — длина списка.
- `static void shuffle(List<?> elements)`
- `static void shuffle(List<?> elements, Random r)`
Случайно перетасовывают элементы в списке. Этот алгоритм выполняется за время $O(n \cdot a(n))$, где n — длина списка, тогда как $a(n)$ — среднее время доступа к элементу списка.
- `static <T> Comparator<T> reverseOrder()`
Возвращает компаратор, сортирующий элементы в порядке, обратном задаваемому методом `compareTo()` из интерфейса `Comparable`.
- `static <T> Comparator<T> reverseOrder(Comparator<T> comp)`
Возвращает компаратор, сортирующий элементы в порядке, обратном задаваемому параметром `comp`.

Двоичный поиск

Чтобы найти объект в массиве, обычно перебираются все его элементы до тех пор, пока не будет найден тот, который соответствует критерию поиска. Но если массив отсортирован, то можно сразу перейти к среднему элементу и сравнить, больше ли

он искомого элемента. Если он больше, то поиск нужно продолжить в первой половине массива, а иначе — во второй его половине. Благодаря этому задача поиска сокращается наполовину. Дальше поиск продолжается в том же духе. Так, если массив содержит 1024 элемента, то совпадение с искомым элементом (или его отсутствие в массиве) будет обнаружено после 10 шагов, тогда как для линейного поиска потребуется в среднем 512 шагов, если элемент присутствует в массиве, и 1024 шага, чтобы убедиться в его отсутствии.

Такой алгоритм двоичного поиска реализуется в методе `binarySearch()` из класса `Collections`. Следует, однако, иметь в виду, что коллекция уже должна быть отсортирована, иначе алгоритм даст неверный результат. Чтобы найти нужный элемент в коллекции, следует передать ее, при условии, что она реализует интерфейс `List`, а также элемент, который требуется найти. Если коллекция не отсортирована методом `compareTo()` из интерфейса `Comparable`, то необходимо предоставить также объект компаратора, как показано ниже.

```
i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, comparator);
```

Неотрицательное числовое значение, возвращаемое методом `binarySearch()`, обозначает индекс найденного объекта. Следовательно, элемент `c.get(i)` равнозначен элементу `element` по порядку сравнения. Если же возвращается отрицательное числовое значение, это означает, что элемент не найден. Но возвращаемое значение можно использовать для определения места, куда *следует* ввести элемент `element` в коллекцию, чтобы сохранить порядок сортировки. Место ввода элемента определяется следующим образом:

```
insertionPoint = -i - 1;
```

Это не просто значение `-i`, потому что нулевое значение было бы неоднозначным. Иными словами, приведенная ниже операция вводит элемент на нужном месте в коллекции.

```
if (i < 0)
    c.add(-i - 1, element);
```

Алгоритм двоичного поиска нуждается в произвольном доступе, чтобы быть эффективным. Так, если итерацию приходится выполнять поэлементно на половине связного списка, чтобы найти средний элемент, теряются все преимущества двоичного поиска. Поэтому данный алгоритм в методе `binarySearch()` превращается в линейный поиск, если этому методу передается связный список.



НА ЗАМЕТКУ! В версии Java SE 1.3 отсутствовал отдельный интерфейс для упорядоченной коллекции с эффективным произвольным доступом, и поэтому в методе `binarySearch()` выполнялась довольно поверхностная проверка, расширяет ли параметр `list` класс `AbstractSequentialList`. Это положение было исправлено в версии Java SE 1.4. Теперь в методе `binarySearch()` проверяется, реализует ли параметр `list` интерфейс `RandomAccess`. Если реализует, то выполняется двоичный поиск, в противном случае — линейный поиск.

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> int binarySearch(List<T> elements, T key)`
- `static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)`

Осуществляет поиск по ключу *key* в отсортированном списке, используя линейный поиск, если объект *elements* расширяет класс `AbstractSequentialList`, в противном случае — двоичный поиск. Выполнение алгоритма поиска гарантируется за время $O(a(n) \log n)$, где *n* — длина списка, тогда как *a(n)* — среднее время доступа к элементу. Возвращают индекс ключа *key* в списке или отрицательное значение *i*, если ключ не найден в списке. В этом случае ключ *key* должен быть введен на позиции с индексом $-i-1$, чтобы список остался отсортированным.

Простые алгоритмы

В состав класса `Collections` входит ряд простых, но полезных алгоритмов. Один из них был приведен ранее в примере кода, где обнаруживается наибольший элемент в коллекции. К числу других алгоритмов относится копирование элементов из одного списка в другой, заполнение контейнера постоянным значением и обращение списка.

Зачем же включать такие простые алгоритмы в стандартную библиотеку? Ведь их нетрудно реализовать в простых циклах. Простые алгоритмы удобны тем, что они упрощают чтение исходного кода. Когда вы читаете цикл, реализованный кем-то другим, вам приходится расшифровывать намерения автора этого кода. Так, если вам встречается в коде вызов такого метода, как, например, `Collections.max()`, вы знаете точно, что этот код делает. В представленном ниже описании прикладного интерфейса API вкратце описываются простые алгоритмы из класса `Collections`.

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> T min(Collection<T> elements)`
- `static <T extends Comparable<? super T>> T max(Collection<T> elements)`
- `static <T> min(Collection<T> elements, Comparator<? super T> c)`
- `static <T> max(Collection<T> elements, Comparator<? super T> c)`
Возвращают наименьший или наибольший элемент из коллекции (границы параметров упрощены для ясности).
- `static <T> void copy(List<? super T> to, List<T> from)`
Копирует все элементы из исходного списка на те же позиции целевого списка. Целевой список должен быть не короче исходного.
- `static <T> void fill(List<? super T> l, T value)`
Устанавливает на всех позициях списка одно и то же значение.

- `static <T> boolean addAll(Collection<? super T> c, T... values) 5.0`

Вводит все значения в заданную коллекцию и возвращает логическое значение `true`, если в результате этого коллекция изменилась.

- `static <T> boolean replaceAll(List<T> l, T oldValue, T newValue) 1.4`

Заменяет на `newValue` все элементы, равные `oldValue`.

- `static int indexOfSubList(List<?> l, List<?> s) 1.4`

- `static int lastIndexOfSubList(List<?> l, List<?> s) 1.4`

Возвращают индекс первого и последнего подсписков `l`, равнозначных `s`, или значение `-1`, если ни один из подсписков `l` не равнозначен `s`. Так, если список `l` содержит `[s, t, a, r]`, а список `s` — `[t, a, r]`, то оба метода возвращают индекс, равный `1`.

- `static void swap(List<?> l, int i, int j) 1.4`

Меняет местами элементы на указанных позициях.

- `static void reverse(List<?> l)`

Меняет порядок следования элементов в списке. Например, в результате обращения списка `[t, a, r]` порождается список `[r, a, t]`. Этот метод выполняется за время $O(n)$, где `n` — длина списка.

- `static void rotate(List<?> l, int d) 1.4`

Циклически сдвигает элементы в списке, перемещая элемент по индексу `i` на позицию `(i+d)%l.size()`. Например, в результате циклического сдвига списка `[t, a, r]` на `2` позиции порождается список `[a, r, t]`. Этот метод выполняется за время $O(n)$, где `n` — длина списка.

- `static int frequency(Collection<?> c, Object o) 5.0`

Возвращает количество элементов в коллекции `c`, равнозначных объекту `o`.

- `boolean disjoint(Collection<?> c1, Collection<?> c2) 5.0`

Возвращает логическое значение `true`, если у коллекций отсутствуют общие элементы.

Написание собственных алгоритмов

Собираясь писать свой собственный алгоритм (по существу, любой метод, принимающий коллекцию в качестве параметра), старайтесь по возможности оперировать *интерфейсами*, а не конкретными их реализациями. Предположим, что компонент типа `JMenu` требуется заполнить рядом пунктов меню. Традиционно такой метод может быть реализован примерно так:

```
void fillMenu(JMenu menu, ArrayList<JMenuItem> items)
{
    for (JMenuItem item : items)
        menu.add(item);
}
```

Но в этом случае ограничиваются возможности той части кода, где вызывается приведенный выше метод, поскольку ему должны передаваться пункты в списочном массиве типа `ArrayList`. Если это окажется другой контейнер, его придется сначала переупаковать. Намного лучше принимать в качестве параметра обобщенную коллекцию.

Необходимо задать себе следующий вопрос: какой интерфейс коллекции является наиболее обобщенным, чтобы справиться с поставленной задачей? В данном случае нужно просто перебрать все элементы, что способен сделать базовый интерфейс `Collection`. Ниже показано, как можно переписать метод `fillMenu()`, чтобы он принимал коллекцию любого типа. Теперь всякий может вызвать этот метод, передав ему объект типа `ArrayList`, `LinkedList` или даже массив, заключенный в оболочку `Arrays.asList`.

```
void fillMenu(JMenu menu, Collection<JMenuItem> items)
{
    for (JMenuItem item : items)
        menu.add(item);
}
```



НА ЗАМЕТКУ! Если использовать интерфейс коллекции в качестве параметра метода настолько удобно, то почему же в библиотеке Java не так часто соблюдается этот замечательный принцип? Например, в классе `JComboBox` имеются следующие два конструктора:

```
JComboBox(Object[] items)
JComboBox(Vector<?> items)
```

Причина проста: библиотека `Swing` была создана до библиотеки коллекций.

Если вы собираетесь писать метод, возвращающий коллекцию, то лучше организуйте в нем возврат интерфейса вместо класса. Ведь в дальнейшем вы можете передумать и реализовать свой метод еще раз, но уже с другой коллекцией. В качестве примера рассмотрим приведенный ниже метод `getAllItems()`, возвращающий все пункты меню.

```
List<JMenuItem> getAllItems(JMenu menu)
{
    List<JMenuItem> items = new ArrayList<>()
    for (int i = 0; i < menu.getItemCount(); i++)
        items.add(menu.getItem(i));
    return items;
}
```

Если в дальнейшем будет решено, что вместо копирования элементов достаточно предоставить их представление, для этого окажется достаточно вернуть анонимный подкласс, производный от класса `AbstractList`, как показано ниже.

```
List<JMenuItem> getAllItems(final JMenu menu)
{
    return new
        AbstractList<>()
    {
        public JMenuItem get(int i)
        {
            return menu.getItem(i);
        }
        public int size()
        {
            return menu.getItemCount();
        }
    };
}
```

Конечно, эта методика относится к разряду передовых, а следовательно, непростых. И если уж вы применяете ее, не забудьте точно задокументировать поддерживаемые “необязательные” операции. В данном случае следует предупредить пользователя метода `getAllItems()`, что возвращаемая им коллекция представляет собой немодифицируемый список.

Унаследованные коллекции

В этом разделе речь пойдет о классах коллекций, которые существуют в Java с самого начала. Это класс `Hashtable` и его удобный подкласс `Properties`, подкласс `Stack`, производный от класса `Vector`, а также класс `BitSet`.

Класс `Hashtable`

Традиционный класс `Hashtable` служит той же цели, что и класс `HashMap`, и по существу имеет тот же самый интерфейс. Как и методы класса `Vector`, методы класса `Hashtable` синхронизированы. Если же не нужна синхронизация или совместимость с унаследованным кодом, то вместо него следует использовать класс `HashMap`.



НА ЗАМЕТКУ! В имени `Hashtable` данного класса вместо прописной присутствует строчная буква `t`. Если вместо этого написать в коде имя `HashTable`, то в Windows будет выдано странное сообщение об ошибке, потому что в файловой системе Windows регистр символов не учитывается, тогда как в компиляторе Java он учитывается.

Перечисления

В унаследованных коллекциях используется интерфейс `Enumeration` для обхода последовательностей элементов. У интерфейса `Enumeration` имеются два метода: `hasMoreElements()` и `nextElement()`. Они являются полными аналогами методов `hasNext()` и `next()` из интерфейса `Iterator`. Например, метод `elements()` из класса `Hashtable` порождает объект для перечисления значений из таблицы, как показано ниже.

```
Enumeration<Employee> e = staff.elements();
while (e.hasMoreElements())
{
    Employee e = e.nextElement();
    . . .
}
```

Иногда может встретиться унаследованный метод, который ожидает перечисление в качестве своего параметра. Статический метод `Collections.enumeration()` порождает объект перечисления всех элементов коллекции, как показано в следующем примере кода:

```
List<InputStream> streams = . . . ;
SequenceInputStream in =
    new SequenceInputStream(Collections.enumeration(streams));
// в конструкторе класса SequenceInputStream ожидается перечисление
```



НА ЗАМЕТКУ C++! В языке C++ итераторы очень часто используются в качестве параметров. Правда, при программировании на платформе Java лишь очень немногие пользуются этим приемом. Разумнее передавать коллекцию, чем итератор, поскольку объект коллекции удобнее. Код, принимающий параметры, может всегда получить итератор из коллекции, когда потребуется. К тому же он получает в свое распоряжение все методы коллекции. Но в унаследованном коде можно встретить перечисления, поскольку раньше они были единственным доступным механизмом обобщенных коллекций до появления архитектуры коллекций в версии Java SE 1.2.

`java.util.Enumeration<E> 1.0`

- `boolean hasMoreElements()`
Возвращает логическое значение `true`, если в коллекции еще остались элементы для просмотра.
- `E nextElement()`
Возвращает следующий элемент для просмотра. Не вызывайте этот метод, если метод `hasMoreElements()` возвратит логическое значение `false`.

`java.util.Hashtable<K, V> 1.0`

- `Enumeration<K> keys()`
Возвращает объект перечисления, проходящий по всем ключам хеш-таблицы.
- `Enumeration<V> elements()`
Возвращает объект перечисления, проходящий по всем значениям хеш-таблицы.

`java.util.Vector<E> 1.0`

- `Enumeration<E> elements()`
Возвращает объект перечисления, проходящий по всем элементам вектора.

Таблицы свойств

Таблица свойств — это структура отображения особенного типа. Она обладает следующими особенностями.

- Ключи и значения являются символьными строками.
- Таблица может быть сохранена в файле и загружена из файла.
- Используется вторичная таблица для установок по умолчанию.

Класс платформы Java, реализующий таблицу свойств, называется `Properties`. Таблицы свойств широко применяются для описания глобальных параметров настройки программ (см. главу 10).

java.util.Properties 1.0

- `Properties()`
Создает пустую таблицу свойств.
- `Properties(Properties defaults)`
Создает пустую таблицу свойств с рядом установок по умолчанию.
- `String getProperty(String key)`
Получает связь со свойством. Возвращает символьную строку, связанную с ключом, или аналогичную строку из таблицы установок по умолчанию, если ключ отсутствует в текущей таблице.
- `String getProperty(String key, String defaultValue)`
Получает свойство со значением по умолчанию, если ключ не найден. Возвращает символьную строку, связанную с ключом, или символьную строку по умолчанию, если ключ отсутствует в текущей таблице.
- `void load(InputStream in)`
Загружает таблицу свойств из потока ввода `InputStream`.
- `void store(OutputStream out, String commentString)`
Выводит карту таблицы свойств в поток вывода `OutputStream`.

Стеки

С версии 1.0 в стандартную библиотеку Java входит класс `Stack` с хорошо известными методами `push()` и `pop()`. Но класс `Stack` расширяет класс `Vector`, что неудовлетворительно с теоретической точки зрения, поскольку в нем допускаются операции, не характерные для стека. Например, вызывая методы `insert()` и `remove()`, можно вводить и удалять значения откуда угодно, а не только из вершины стека.

java.util.Stack<E> 1.0

- `E push(E item)`
Проталкивает элемент `item` в стек и возвращает его.
- `E pop()`
Вытаскивает и возвращает элемент из вершины стека. Не вызывайте этот метод, если стек пуст.
- `E peek()`
Возвращает элемент, находящийся на вершине стека, не вытаскивая его. Не вызывайте этот метод, если стек пуст.

Битовые множества

В классе `BitSet` платформы Java хранятся последовательности битов. (Это не *множество* в математическом смысле. Битовый *вектор*, или битовый *массив*, — возможно, более подходящий для этого термин.) Битовое множество применяется в тех случаях, когда требуется эффективно хранить последовательность битов (например, признаков или флагов). А поскольку битовое множество упаковывает биты в байты, то пользоваться им намного эффективнее, чем списочным массивом типа `ArrayList` с объектами типа `Boolean`.

Класс `BitSet` предоставляет удобный интерфейс для чтения, установки или переустановки отдельных битов. Применяя этот интерфейс, можно избежать маскирования или других операций с битами, которые потребовались бы, если бы биты хранились в переменных типа `long`.

Например, в результате вызова `bucketOfBits.get(i)` для объекта `bucketOfBits` типа `BitSet` возвращается логическое значение `true`, если *i*-й бит установлен, а иначе — логическое значение `false`. Аналогично в результате вызова `bucketOfBits.set(i)` устанавливается *i*-й бит. И наконец, в результате вызова `bucketOfBits.clear(i)` сбрасывается *i*-й бит.



НА ЗАМЕТКУ C++! Шаблон `bitset` в C++ обладает теми же функциональными возможностями, что и класс `BitSet` платформы Java.

`java.util.BitSet` 1.0

- `BitSet(int initialCapacity)`
Конструирует битовое множество.
- `int length()`
Возвращает “логическую длину” битового множества: 1 + индекс самого старшего установленного бита.
- `boolean get(int bit)`
Получает бит.
- `void set(int bit)`
Устанавливает бит.
- `void clear(int bit)`
Сбрасывает бит.
- `void and(BitSet set)`
Выполняет логическую операцию И над данным и другим битовым множеством.
- `void or(BitSet set)`
Выполняет логическую операцию ИЛИ над данным и другим битовым множеством.
- `void xor(BitSet set)`
Выполняет логическую операцию исключающее ИЛИ над данным и другим битовым множеством.
- `void andNot(BitSet set)`
Сбрасывает все биты данного битового множества, установленные в другом битовом множестве.

Реализация алгоритма “Решето Эратосфена”

Продемонстрируем применение битовых множеств на примере реализации алгоритма “Решето Эратосфена”, служащего для нахождения простых чисел. (Простое число — это число вроде 2, 3 или 5, которое делится только на самое себя и на 1, а “Решето Эратосфена” было одним из первых методов, изобретенных для перечисления этих основополагающих математических единиц.) Это не самый лучший алгоритм для нахождения простых чисел, но по ряду причин он получил широкое распространение в качестве теста производительности компилятора. (Впрочем, это далеко не лучший тест, поскольку он тестирует в основном битовые операции.) Отдавая дань традиции, реализуем этот алгоритм в программе, подсчитывающей все простые числа от 2 до 2000000. (В этом диапазоне находится 148 933 простых числа, так что вряд ли стоит выводить весь этот ряд чисел.)

Не слишком вдаваясь в детали, поясним: ключ к достижению цели лежит в проходе по битовому множеству из 2 млн. бит. Сначала все эти биты устанавливаются в 1. Затем сбрасываются те биты, которые кратны известным простым числам. Позиции битов, оставшихся после этого процесса, сами представляют простые числа. В листинге 13.8 приведен исходный код данной программы на Java, а в листинге 13.9 — на C++.

Листинг 13.8. Исходный код из файла sieve/Sieve.java

```
1 package sieve;
2
3 import java.util.*;
4
5 /**
6  * В этой программе выполняется тест по алгоритму "Решето Эратосфена"
7  * для нахождения всех простых чисел вплоть до 2000000
8  * @version 1.21 2004-08-03
9  * @author Cay Horstmann
10 */
11 public class Sieve
12 {
13     public static void main(String[] s)
14     {
15         int n = 2000000;
16         long start = System.currentTimeMillis();
17         BitSet b = new BitSet(n + 1);
18         int count = 0;
19         int i;
20         for (i = 2; i <= n; i++)
21             b.set(i);
22         i = 2;
23         while (i * i <= n)
24         {
25             if (b.get(i))
26             {
27                 count++;
28                 int k = 2 * i;
29                 while (k <= n)
30                 {
31                     b.clear(k);
32                     k += i;
33                 }

```

```
34     }
35     i++;
36 }
37 while (i <= n)
38 {
39     if (b.get(i)) count++;
40     i++;
41 }
42 long end = System.currentTimeMillis();
43 System.out.println(count + " primes");
44 System.out.println((end - start) + " milliseconds");
45 }
46 }
```

Листинг 13.9. Исходный код из файла sieve/sieve.cpp

```
1  /**
2   * @version 1.21 2004-08-03
3   * @author Cay Horstmann
4   */
5
6  #include <bitset>
7  #include <iostream>
8  #include <ctime>
9  using namespace std;
10
11 int main()
12 {
13     const int N = 2000000;
14     clock_t cstart = clock();
15
16     bitset<N + 1> b;
17     int count = 0;
18     int i;
19     for (i = 2; i <= N; i++)
20         b.set(i);
21     i = 2;
22     while (i * i <= N)
23     {
24         if (b.test(i))
25         {
26             count++;
27             int k = 2 * i;
28             while (k <= N)
29             {
30                 b.reset(k);
31                 k += i;
32             }
33         }
34         i++;
35     }
36     while (i <= N)
37     {
38         if (b.test(i))
39             count++;
40         i++;
```

```
41 }
42
43 clock_t cend = clock();
44 double millis = 1000.0 * (cend - cstart) / CLOCKS_PER_SEC;
45
46 cout << count << " primes\n" << millis << " milliseconds\n";
47
48 return 0;
49 }
```



НА ЗАМЕТКУ! Несмотря на то что “Решето Эратосфена” — не самый лучший тест, мы все же не преминули сравнить время работы обеих представленных выше реализаций данного алгоритма. Ниже приведены результаты прогона обеих реализаций теста на переносном компьютере ThinkPad с двухъядерным процессором на 2,4 ГГц и ОЗУ на 4 Гбайт, работающем под управлением ОС Ubuntu 10.04:

- C++ (g++ 4.6.3): 160 мс
- Java (Java SE 7): 84 мс

Мы запускали этот тест для восьми изданий данной книги, и в четырех последних язык Java легко одерживал верх над языком C++. Справедливости ради, следует сказать, что при повышении уровня оптимизации компилятора язык C++ обгонял Java на 20 мс. А язык Java мог сравниться с этим показателем только при достаточно долгой работе программы, чтобы вступил в действие динамический компилятор Hotspot.

На этом рассмотрение архитектуры коллекций Java завершается. Как вы сами могли убедиться, в библиотеке Java предоставляется широкое разнообразие классов коллекций для ваших потребностей в программировании. В завершающей главе первого тома данной книги будет обсуждаться очень важная тема параллельного программирования.

Многопоточная обработка

В этой главе...

- ▶ Назначение потоков
- ▶ Прерывание потоков
- ▶ Состояния потоков
- ▶ Свойства потоков
- ▶ Синхронизация
- ▶ Блокирующие очереди
- ▶ Потокобезопасные коллекции
- ▶ Интерфейсы `Callable` и `Future`
- ▶ Исполнители
- ▶ Синхронизаторы
- ▶ Потоки и библиотека `Swing`

Вам, вероятно, хорошо известна *многозадачность* используемой вами операционной системы — возможность одновременно выполнять несколько программ. Например, вы можете печатать документы во время их редактирования или приема электронной почты. Вполне возможно, что ваш современный компьютер оснащен более чем одним центральным процессором, но число одновременно выполняющихся процессов не ограничивается количеством процессоров. Операционная система выделяет время процессора квантами для каждого процесса, создавая впечатление параллельного выполнения программ.

Многопоточные программы расширяют принцип многозадачности, перенося его на один уровень ниже, чтобы отдельные приложения могли выполнять многие задачи одновременно. Каждая задача обычно называется *поток*ом, т.е. потоком управления. Программы, способные одновременно выполнять больше одного потока, называются *многопоточными*.

Так в чем же отличие между многими процессами и многими потоками? Оно состоит в следующем: если у каждого процесса имеется собственный набор переменных, то потоки могут разделять одни и те же общие данные. Это кажется несколько рискованным, и на самом деле так оно и есть, как станет ясно далее в этой главе. Но разделяемые переменные обеспечивают более высокую эффективность взаимодействия потоков и облегчают связь между потоками. Кроме того, в некоторых операционных системах потоки являются более “легковесными”, чем процессы, — они требуют меньших издержек на свое создание и уничтожение по сравнению с процессами.

Многопоточная обработка имеет исключительную практическую ценность. Например, браузер должен уметь одновременно загружать многие изображения. Веб-серверу нужно обслуживать параллельные запросы. Программы с ГПИ имеют отдельные потоки для сбора событий в пользовательском интерфейсе из среды операционной системы. В этой главе речь пойдет о том, как внедрять многопоточные возможности в прикладные программы на Java.

Следует, однако, иметь в виду, что многопоточная обработка может оказаться очень сложным делом. В этой главе будут рассмотрены все инструментальные средства, которые, скорее всего, понадобятся прикладным программистам. Но для более сложного программирования на системном уровне рекомендуется обратиться к таким основательным первоисточникам, как, например, книга Брайана Гоеца (Brian Goetz) *Java Concurrency in Practice* (издательство Addison-Wesley Professional, 2006 г.).

Назначение потоков

Начнем с рассмотрения примера программы, в которой не применяются средства многопоточной обработки, и, как следствие, пользователь испытывает затруднение, пытаясь решить сразу несколько задач с помощью этой программы. Проанализировав ее, мы покажем, насколько просто заставить эту программу запускать отдельные потоки на выполнение. Эта программа осуществляет анимацию скачущего мяча, постоянно перемещая его по экрану и меняя направление его движения, когда он отскакивает от стены (в данном случае от края экрана; рис. 14.1).

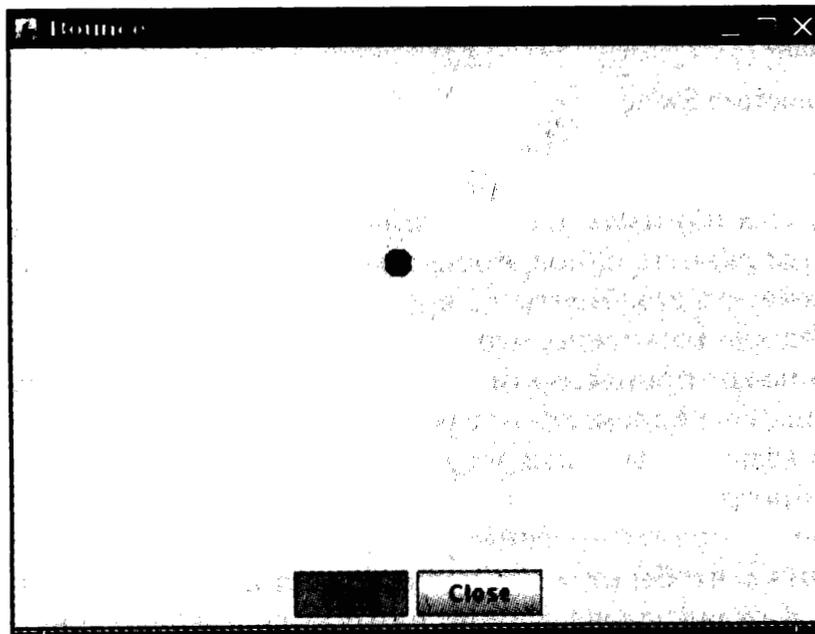


Рис. 14.1. Применение потока для анимации скачущего мяча

Как только вы щелкнете на кнопке Start (Пуск), программа начнет движение скачущего мяча из левого верхнего угла. Обработчик событий от кнопки Start вызывает метод `addBall()`, который содержит цикл из 1000 движений мяча. При каждом вызове метода `move()` мяч перемещается на небольшое расстояние, изменяя направление своего движения, если он отскакивает от стены, после чего панель перерисовывается с учетом нового положения мяча. Все эти операции с мячом реализуются в приведенном ниже фрагменте кода.

```
Ball ball = new Ball();
panel.add(ball);
for (int i = 1; i <= STEPS; i++)
{
    ball.move(panel.getBounds());
    panel.paint(panel.getGraphics());
    Thread.sleep(DELAY);
}
```

В результате вызова `Thread.sleep()` новый поток не создается. Статический метод `sleep()` из класса `Thread` временно приостанавливает действие текущего потока. Но метод `sleep()` может сгенерировать исключение типа `InterruptedException`. Мы еще вернемся к обсуждению этого исключения и его правильной обработки. А до тех пор достаточно прекратить отскоки мяча, если возникнет такое исключение.

Если запустить эту программу на выполнение, мяч будет скакать почти идеально, но программа будет полностью занята управлением его движения. Если вам надоест наблюдать за скачущим мячом до того, как он выполнит 1000 движений, и вы щелкнете на кнопке Close (Закреть), мяч все равно продолжит скакать. Вы не сможете взаимодействовать с программой до тех пор, пока мяч не перестанет скакать.



НА ЗАМЕТКУ! Если вы внимательно проанализируете исходный код из листингов в конце этого раздела, то обнаружите вызов `comp.paint(comp.getGraphics())` в теле метода `addBall()` из класса `BounceFrame`. Это довольно необычный прием. Ведь обычно вызывается метод `repaint()`, чтобы библиотека AWT позаботилась о графическом контексте и рисовании в нем. Но если вы попытаетесь сделать вызов `comp.repaint()` в рассматриваемой здесь программе, то панель вообще не будет перерисована, потому что метод `addBall()` полностью захватывает управление программой. Обратите также внимание на то, что компонент, отображающий мяч, расширяет класс `JPanel`, что облегчает очистку фона. В следующем примере программы, где предполагается использовать отдельный поток для вычисления текущего положения мяча, можно будет вернуться к привычному применению метода `repaint()` и класса `JComponent`.

Очевидно, что такое поведение данной программы никуда не годится. Оно совершенно не подходит для программ, рассчитанных на более или менее длительную работу. Ведь когда данные загружаются по сетевому соединению, нередко приходится иметь дело с задачами, которые на самом деле следовало бы прервать. Допустим, вы загружаете крупное изображение и, увидев его часть, решаете, что вам не нужно видеть остальное. Очевидно, что было бы неплохо иметь возможность щелчком на кнопке Stop (Останов) или Back (Назад) прервать процесс загрузки. В следующем разделе мы покажем, как предоставить пользователю контроль над таким процессом, запуская критические разделы кода в *отдельном* потоке. Исходный код рассматриваемой здесь программы приведен в листингах 14.1–14.3.

Листинг 14.1. Исходный код из файла bounce/Bounce.java

```
1 package bounce;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * В этой программе осуществляется анимация скачущего мяча
9  * @version 1.33 2007-05-17
10 * @author Cay Horstmann
11 */
12 public class Bounce
13 {
14     public static void main(String[] args)
15     {
16        .EventQueue.invokeLater(new Runnable()
17         {
18             public void run()
19             {
20                 JFrame frame = new BounceFrame();
21                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22                 frame.setVisible(true);
23             }
24         });
25     }
26 }
27
28 /**
29  * Фрейм с компонентом мяча и кнопками
30 */
31 class BounceFrame extends JFrame
32 {
33     private BallComponent comp;
34     public static final int STEPS = 1000;
35     public static final int DELAY = 3;
36
37     /**
38      * Конструирует фрейм с компонентом, отображающим
39      * скачущий мяч, а также кнопки Start и Close
40      */
41     public BounceFrame()
42     {
43         setTitle("Bounce");
44
45         comp = new BallComponent();
46         add(comp, BorderLayout.CENTER);
47         JPanel buttonPanel = new JPanel();
48         addButton(buttonPanel, "Start", new ActionListener()
49         {
50             public void actionPerformed(ActionEvent event)
51             {
52                 addBall();
53             }
54         });
55     }
```

```
56     addButton(buttonPanel, "Close", new ActionListener()
57     {
58         public void actionPerformed(ActionEvent event)
59         {
60             System.exit(0);
61         }
62     });
63     add(buttonPanel, BorderLayout.SOUTH);
64     pack();
65 }
66 /**
67  * Вводит кнопку в контейнер
68  * @param c Контейнер
69  * @param title Надпись на кнопке
70  * @param listener Приемник действий кнопки
71  */
72 public void addButton(Container c, String title, ActionListener listener)
73 {
74     JButton button = new JButton(title);
75     c.add(button);
76     button.addActionListener(listener);
77 }
78
79 /**
80  * Вводит скачущий мяч на панели и производит 1000 его отскоков
81  */
82 public void addBall()
83 {
84     try
85     {
86         Ball ball = new Ball();
87         comp.add(ball);
88
89         for (int i = 1; i <= STEPS; i++)
90         {
91             ball.move(comp.getBounds());
92             comp.paint(comp.getGraphics());
93             Thread.sleep(DELAY);
94         }
95     }
96     catch (InterruptedException e)
97     {
98     }
99 }
100 }
```

Листинг 14.2. Исходный код из файла bounce/Ball.java

```
1 package bounce;
2
3 import java.awt.geom.*;
4
5 /**
6  * Анимация движения и отскока мяча от краев прямоугольника
7  * @version 1.33 2007-05-17
8  * @author Cay Horstmann
```

```
9 */
10 public class Ball
11 {
12     private static final int XSIZE = 15;
13     private static final int YSIZE = 15;
14     private double x = 0;
15     private double y = 0;
16     private double dx = 1;
17     private double dy = 1;
18
19     /**
20      * Перемещает мяч в следующее положение, изменяя направление его
21      * движения, как только он достигает одного из краев прямоугольника
22      */
23     public void move(Rectangle2D bounds)
24     {
25         x += dx;
26         y += dy;
27         if (x < bounds.getMinX())
28         {
29             x = bounds.getMinX();
30             dx = -dx;
31         }
32         if (x + XSIZE >= bounds.getMaxX())
33         {
34             x = bounds.getMaxX() - XSIZE;
35             dx = -dx;
36         }
37         if (y < bounds.getMinY())
38         {
39             y = bounds.getMinY();
40             dy = -dy;
41         }
42         if (y + YSIZE >= bounds.getMaxY())
43         {
44             y = bounds.getMaxY() - YSIZE;
45             dy = -dy;
46         }
47     }
48
49     /**
50      * Получает форму мяча в его текущем положении
51      */
52     public Ellipse2D getShape()
53     {
54         return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
55     }
56 }
```

Листинг 14.3. Исходный код из файла bounce/BallComponent.java

```
1 package bounce;
2
3 import java.awt.*;
4 import java.util.*;
5 import javax.swing.*;
6
```

```
7 /**
8  * Компонент, рисующий скачущий мяч
9  * @version 1.34 2012-01-26
10 * @author Cay Horstmann
11 */
12 public class BallComponent extends JPanel
13 {
14     private static final int DEFAULT_WIDTH = 450;
15     private static final int DEFAULT_HEIGHT = 350;
16
17     private java.util.List<Ball> balls = new ArrayList<>();
18
19     /**
20      * Вводит мяч в компонент
21      * @param b Вводимый мяч
22      */
23     public void add(Ball b)
24     {
25         balls.add(b);
26     }
27
28     public void paintComponent(Graphics g)
29     {
30         super.paintComponent(g); // очистить фон
31         Graphics2D g2 = (Graphics2D) g;
32         for (Ball b : balls)
33         {
34             g2.fill(b.getShape());
35         }
36     }
37
38     public Dimension getPreferredSize()
39     { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
40 }
```

java.lang.Thread 1.0

- `static void sleep(long millis)`

Переходит в состояние ожидания на заданное число миллисекунд.

Параметры: *millis* Количество миллисекунд ожидания

Предоставление возможности для выполнения других задач с помощью потоков

Попробуем теперь сделать рассматриваемую здесь программу анимации скачущего мяча более отзывчивой, запустив код передвижения скачущего мяча в отдельном потоке. На самом деле в отдельных потоках можно запустить в движение много мячей. Кроме того, *поток диспетчеризации событий* в AWT продолжит работать параллельно, взяв на себя заботу о событиях в пользовательском интерфейсе. А поскольку каждый поток получает возможность для выполнения, то и у потока диспетчеризации событий будет возможность уведомить программу, когда пользователь щелкнет на кнопке Close во время передвижения скачущего мяча. Затем действие закрытия программы может быть обработано в отдельном потоке.

Пример программы анимации скачущего мяча служит для того, чтобы дать визуальное представление о необходимости параллелизма. Вообще говоря, следует с большой осторожностью относиться ко всякого рода долго выполняющимся вычислениям. Ведь такие вычисления всегда являются частью некоторой более крупной среды вроде ГПИ или веб-приложения. Всякий раз, когда в этой среде вызываются методы из прикладной программы, ожидается быстрый возврат. Если выполнение задачи требует длительного времени, ее следует выполнять в отдельном потоке.

Ниже приведена простая процедура для выполнения задачи в отдельном потоке.

1. Введите код выполняемой задачи в тело метода `run()` из класса, реализующего интерфейс `Runnable`. Этот интерфейс очень прост и содержит следующий единственный метод:

```
public interface Runnable
{
    void run();
}
```

2. Вам достаточно реализовать класс аналогично следующему:

```
class MyRunnable implements Runnable
{
    public void run()
    {
        код задачи
    }
}
```

3. Сконструируйте объект своего класса следующим образом:

```
Runnable r = new MyRunnable();
```

4. Сконструируйте объект типа `Thread` из объекта `r` типа `Runnable`, как показано ниже.

```
Thread t = new Thread(r);
```

5. Запустите поток на выполнение следующим образом:

```
t.start();
```

Чтобы ввести рассматриваемую здесь программу анимации скачущего мяча в отдельный поток, достаточно реализовать класс `BallRunnable` и ввести код анимации скачущего мяча в тело метода `run()`, как показано ниже.

```
class BallRunnable implements Runnable
{
    . . .
    public void run()
    {
        try
        {
            for (int i = 1; i <= STEPS; i++)
            {
                ball.move(component.getBounds());
                component.repaint();
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception)
    }
}
```

```
    {  
    }  
    }  
    . . .  
}
```

И в этом случае следует перехватить исключение типа `InterruptedException`, которое может сгенерировать метод `run()`. Об исключениях речь пойдет в следующем разделе. Обычно это исключение служит для того, чтобы запросить прерывание потока. А это, в свою очередь, приводит к выходу из метода `run()`. Всякий раз, когда выбирается кнопка `Start`, метод `addBall()` запускает новый поток, как показано в приведенном ниже коде и на рис. 14.2.

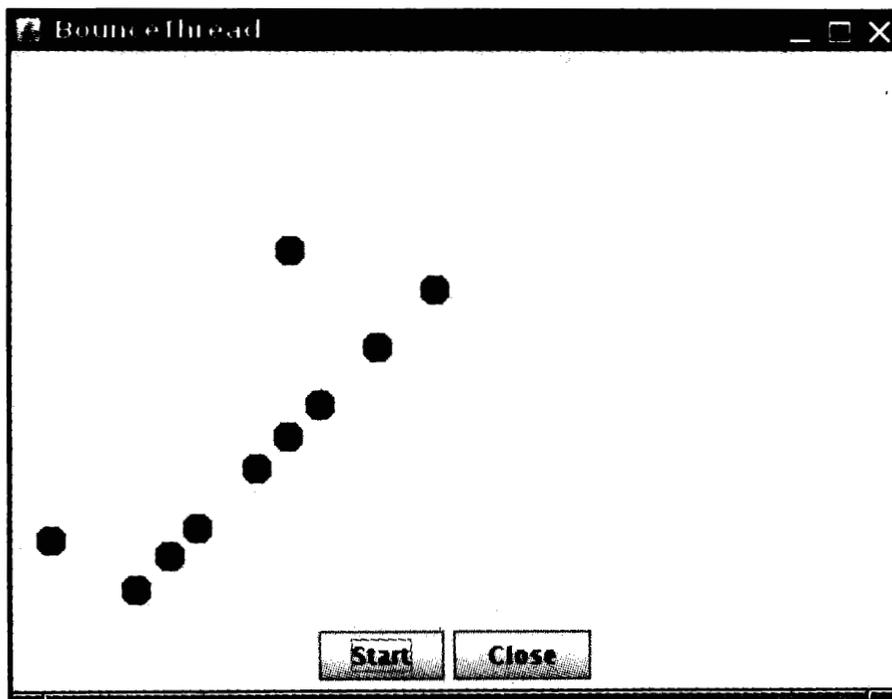


Рис. 14.2. Многие работающие потоки

```
Ball b = new Ball();  
panel.add(b);  
Runnable r = new BallRunnable(b, panel);  
Thread t = new Thread(r);  
t.start();
```

Вот, собственно, и все! Теперь вы знаете, как запускать задачи на параллельное выполнение. Остальная часть этой главы будет посвящена вопросам взаимодействия потоков. Весь исходный код рассмотренной здесь программы приведен в листинге 14.4.



НА ЗАМЕТКУ! Потоки можно также определить, создавая подклассы, производные от класса `Thread`, как показано ниже.

```
class MyThread extends Thread  
{  
    public void run()  
    {  
        код задачи  
    }  
}
```

Далее конструируется объект этого подкласса и вызывается его метод `start()`. Но такой подход не рекомендуется. Задачу, которая должна выполняться параллельно, следует отделять от механизма ее выполнения. Если имеется много задач, то было бы слишком неэффективно создавать отдельный поток для каждой из них. Вместо этого можно организовать пул потоков, как поясняется в разделе "Исполнители" далее в этой главе.



ВНИМАНИЕ! Не вызывайте метод `run()` из класса `Thread` или объекта типа `Runnable`. При прямом вызове этого метода задача будет просто выполнена в том же потоке, а новый поток не будет запущен. Вместо этого вызывайте метод `Thread.start()`. Он создаст новый поток, где будет выполнен метод `run()`.

Листинг 14.4. Исходный код из файла `bounceThread/BounceThread.java`

```
1 package bounceThread;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * В этой программе осуществляется анимация скачущих мячей
9  * @version 1.33 2007-05-17
10 * @author Cay Horstmann
11 */
12 public class BounceThread
13 {
14     public static void main(String[] args)
15     {
16         EventQueue.invokeLater(new Runnable()
17         {
18             public void run()
19             {
20                 JFrame frame = new BounceFrame();
21                 frame.setTitle("BounceThread");
22                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23                 frame.setVisible(true);
24             }
25         });
26     }
27 }
28
29 /**
30 * Класс исполняемого потока для анимации скачущего мяча
31 */
32 class BallRunnable implements Runnable
33 {
34     private Ball ball;
35     private Component component;
36     public static final int STEPS = 1000;
37     public static final int DELAY = 5;
38
39     /**
40     * Конструирует объект для исполняемого потока
```

```
41  * @param aBall Скачущий мяч
42  * @param aComponent Компонент, в котором скачет мяч
43  */
44  public BallRunnable(Ball aBall, Component aComponent)
45  {
46      ball = aBall;
47      component = aComponent;
48  }
49  public void run()
50  {
51      try
52      {
53          for (int i = 1; i <= STEPS; i++)
54          {
55              ball.move(component.getBounds());
56              component.repaint();
57              Thread.sleep(DELAY);
58          }
59      }
60      catch (InterruptedException e)
61      {
62      }
63  }
64
65  }
66
67  /**
68  * Фрейм с панелью и кнопками
69  */
70  class BounceFrame extends JFrame
71  {
72      private BallComponent comp;
73
74      /**
75       * Конструирует фрейм с компонентом, отображающим
76       * скачущий мяч, а также кнопки Start и Close
77       */
78      public BounceFrame()
79      {
80          comp = new BallComponent();
81          add(comp, BorderLayout.CENTER);
82          JPanel buttonPanel = new JPanel();
83          addButton(buttonPanel, "Start", new ActionListener()
84              {
85                  public void actionPerformed(ActionEvent event)
86                  {
87                      addBall();
88                  }
89              });
90
91          addButton(buttonPanel, "Close", new ActionListener()
92              {
93                  public void actionPerformed(ActionEvent event)
94                  {
95                      System.exit(0);
96                  }
97              });
```

```
98     add(buttonPanel, BorderLayout.SOUTH);
99     pack();
100 }
101
102 /**
103  * Вводит кнопку в контейнер
104  * @param c Контейнер
105  * @param title Надпись на кнопке
106  * @param listener Приемник действий кнопки
107  */
108 public void addButton(Container c, String title, ActionListener listener)
109 {
110     JButton button = new JButton(title);
111     c.add(button);
112     button.addActionListener(listener);
113 }
114
115 /**
116  * Вводит мяч на холст и запускает поток для анимации скачущего мяча
117  */
118 public void addBall()
119 {
120     Ball b = new Ball();
121     comp.add(b);
122     Runnable r = new BallRunnable(b, comp);
123     Thread t = new Thread(r);
124     t.start();
125 }
126 }
```

java.lang.Thread 1.0

- `Thread(Runnable target)`
Конструирует новый исполняемый поток, вызывающий метод `run()` для указанного целевого объекта.
- `void start()`
Запускает поток, иницируя вызов метода `run()`. Этот метод немедленно возвращает управление. Новый поток выполняется параллельно.
- `void run()`
Вызывает метод `run()` для связанного с ним объекта типа `Runnable`.

java.lang.Runnable 1.0

- `void run()`
Этот метод должен быть переопределен и содержать инструкции для выполнения требуемой задачи в потоке.

Прерывание потоков

Поток прерывается, когда его метод `run()` возвращает управление, выполнив оператор `return` вслед за последним оператором в своем теле, или в том случае, если возникает исключение, которое не перехватывается в данном методе. В первоначальной версии Java также присутствовал метод `stop()`, который мог быть вызван из другого потока, чтобы прервать выполнение данного потока. Но теперь этот метод не рекомендуется к применению, как поясняется далее в разделе “Причины, по которым методы `stop()` и `suspend()` не рекомендованы к применению”.

Поток можно прервать *принудительно*. Но метод `interrupt()` может вызываться только для *запроса* прерывания потока. Когда метод `interrupt()` вызывается для потока, устанавливается *состояние прерывания* данного потока. Это признак типа `boolean`, имеющийся у каждого потока. В каждом потоке следует периодически проверять значение этого признака, чтобы знать, когда он должен быть прерван. Для того чтобы выяснить, было ли установлено состояние прерывания потока, вызывается статический метод `Thread.currentThread().isInterrupted()`, получающий текущий поток и вызывающий далее метод `isInterrupted()`:

```
while (!Thread.currentThread().isInterrupted() && дополнительные действия)
{
    выполнить дополнительные действия
}
```

Но если поток заблокирован, то проверить состояние его прерывания нельзя. И здесь на помощь приходит исключение типа `InterruptedException`. Когда метод `interrupt()` вызывается для потока, который заблокирован, например, вызовом метода `sleep()` или `wait()`, блокирующий вызов прерывается исключением типа `InterruptedException`. (Существуют блокирующие вызовы ввода-вывода, которые не могут быть прерваны. В таких случаях следует рассмотреть альтернативные способы прерывания потока. Подробнее об этом речь пойдет в главах 1 и 3 второго тома данной книги.)

В языке Java не существует требования, чтобы прерванный поток прекратил выполнение. Прерывание лишь привлекает внимание потока. А прерванный поток может сам решить, как реагировать на прерывание. Некоторые потоки настолько важны, что должны обработать исключение и продолжать свое выполнение. Но зачастую поток должен просто интерпретировать прерывание как запрос на прекращение своего выполнения. Метод `run()` такого потока имеет следующий вид:

```
public void run()
{
    try
    {
        . . .
        while (!Thread.currentThread().isInterrupted() && дополнительные действия)
        {
            выполнить дополнительные действия
        }
    }
    catch(InterruptedException e)
    {
        // поток прерван во время ожидания или приостановки
    }
    finally
```

```

{
    выполнить очистку, если требуется
}
// выходом из метода run() завершается выполнение потока
}

```

Проверка состояния прерывания потока вызовом метода `isInterrupted()` не обязательна и не удобна, если можно вызвать метод `sleep()` (или другой прерываемый метод) после каждого рабочего шага цикла. Если же метод `sleep()` вызывается, когда установлено состояние прерывания, поток не переходит в состояние ожидания. Вместо этого он очищает состояние (!) и генерирует исключение типа `InterruptedException`. Таким образом, если метод `sleep()` вызывается в цикле, то проверять состояние прерывания не следует. Вместо этого лучше организовать перехват исключения типа `InterruptedException`, как показано ниже.

```

public void run()
{
    try
    {
        . . .
        while (дополнительные действия)
        {
            выполнить дополнительные действия
            Thread.sleep(delay);
        }
    }
    catch(InterruptedException e)
    {
        // поток прерван во время ожидания
    }
    finally
    {
        выполнить очистку, если требуется
    }
    // выходом из метода run() завершается выполнение потока
}

```



НА ЗАМЕТКУ! Существуют два очень похожих метода `interrupted()` и `isInterrupted()`. Статический метод `interrupted()` проверяет, был ли прерван текущий поток. Более того, вызов этого метода приводит к очистке состояния прерывания потока. С другой стороны, метод экземпляра `isInterrupted()` можно использовать для проверки, был ли прерван любой поток. Его вызов не приводит к изменению состояния прерывания.

Можно найти массу опубликованного кода, где на низком уровне прерывание типа `InterruptedException` игнорируется:

```

void mySubTask()
{
    . . .
    try { sleep(delay); }
    catch (InterruptedException e) {} // НЕ ИГНОРИРОВАТЬ!
    . . .
}

```

Не поступайте так! Если вы не можете придумать ничего полезного из того, что можно было бы сделать в блоке `catch`, вам остаются на выбор два обоснованных варианта.

- Сделать в блоке `catch` вызов `Thread.currentThread().interrupt()`, чтобы установить состояние прерывания потока, как выделено ниже полужирным. И тогда это состояние может быть проверено в вызывающей части программы.

```
void mySubTask()  
{  
    . . .  
    try { sleep(delay); }  
    catch (InterruptedException e) { Thread.currentThread().interrupt(); }  
    . . .  
}
```

- А еще лучше указать `throws InterruptedException` в сигнатуре метода, как выделено ниже полужирным, а также удалить блок `try` из его тела. И тогда это прерывание может быть перехвачено в вызывающей части программы, а в крайнем случае — в методе `run()`.

```
void mySubTask() throws InterruptedException  
{  
    . . .  
    sleep(delay);  
    . . .  
}
```

java.lang.Thread 1.0

- `void interrupt()`
Посылает потоку запрос на прерывание. Признак состояния прерывания потока устанавливается равным логическому значению `true`. Если поток в данный момент блокирован вызовом метода `sleep()`, генерируется исключение типа `InterruptedException`.
- `static boolean interrupted()`
Проверяет, был ли прерван текущий поток. Следует иметь в виду, что это статический метод. Его вызов имеет побочный эффект: признак состояния прерывания текущего потока устанавливается равным логическому значению `false`.
- `boolean isInterrupted()`
Проверяет, был ли прерван поток. В отличие от статического метода `interrupted()`, вызов этого метода не изменяет состояние прерывания потока.
- `static Thread currentThread()`
Возвращает объект типа `Thread`, представляющий текущий выполняющийся поток.

Состояния потоков

Потоки могут находиться в одном из шести состояний:

- **новый;**
- **исполняемый;**
- **блокированный;**
- **ожидающий;**

- временно ожидающий;
- завершённый.

Каждое из этих состояний поясняется в последующих разделах. Чтобы определить текущее состояние потока, достаточно вызвать метод `getState()`.

Новые потоки

Если поток создан в результате операции `new`, например `new Thread(r)`, то он еще не запущен на выполнение. Это означает, что он находится в новом состоянии и что программа еще не запустила на выполнение код в данном потоке. Прежде чем поток будет запущен, необходимо выполнить определенные подготовительные операции.

Исполняемые потоки

Как только вызывается метод `start()`, поток оказывается в исполняемом состоянии. Исполняемый поток может выполняться или не выполняться в данный момент, поскольку от операционной системы зависит, будет ли выделено потоку время на выполнение. (В спецификации Java, однако, это отдельное состояние не указывается. Выполняющийся поток по-прежнему находится в исполняемом состоянии.)

Если поток запущен, он не обязательно продолжает выполняться. На самом деле даже желательно, чтобы выполняющийся поток периодически приостанавливался, давая возможность выполниться другим потокам. Особенности планирования потоков зависят от конкретных служб, предоставляемых операционной системой. Системы приоритетного планирования выделяют каждому исполняемому потоку по кванту времени для выполнения его задачи. Когда этот квант времени истекает, операционная система выгружает поток и дает возможность выполниться другому потоку (рис. 14.4). Выбирая следующий поток, операционная система принимает во внимание *приоритеты* потоков, как поясняется далее в разделе “Приоритеты потоков”.

Все современные настольные и серверные операционные системы используют приоритетное (вытесняющее) планирование. Но на переносных устройствах вроде мобильных телефонов может применяться кооперативное планирование. В таких устройствах поток теряет управление только в том случае, если он вызывает метод `yield()`, заблокирован или находится в состоянии ожидания.

На машинах с несколькими процессорами каждый процессор может выполнять поток, что позволяет иметь несколько потоков, работающих одновременно. Очевидно, если потоков больше, чем процессоров, планировщик вынужден заниматься разделением времени для их выполнения. Не следует, однако, забывать, что в любой момент времени исполняемый поток может выполняться или не выполняться. Именно поэтому рассматриваемое здесь состояние потока называется исполняемым, а не исполняющимся.

Блокированные и ожидающие потоки

Когда поток находится в состоянии блокировки или ожидания, он временно не активен. Он не выполняет никакого кода и потребляет минимум ресурсов. На планировщике потоков лежит обязанность повторно активизировать его. Подробности зависят от того, как было достигнуто неактивное состояние.

- Когда поток пытается получить встроенную блокировку объектов (но не объект типа `Lock` из библиотеки `java.util.concurrent`), которая в настоящий момент захвачена другим потоком, он становится блокированным. (Блокировки из библиотеки `java.util.concurrent` будут обсуждаться далее в разделе “Объекты блокировки”, а встроенные блокировки объектов — в разделе “Ключевое слово `synchronized`”.) Поток разблокируется, когда все остальные потоки освобождают блокировку и планировщик потоков позволяет захватить ее данному потоку.
- Когда поток ожидает от другого потока уведомления планировщика о наступлении некоторого условия, он входит в состояние ожидания. Эти условия рассматриваются ниже, в разделе “Объекты условий”. Переход в состояние ожидания происходит при вызове метода `Object.wait()` или `Thread.join()` либо в ожидании объекта типа `Lock` или `Condition` из библиотеки `java.util.concurrent`. Но на практике отличия состояний блокировки и ожидания не существенны.
- Несколько методов принимают в качестве параметра время ожидания. Их вызов вводит поток в состояние *временного ожидания*. Это состояние сохраняется до тех пор, пока не истечет заданное время ожидания или не будет получено соответствующее уведомление. К числу методов со временем ожидания относятся `Object.wait()`, `Thread.join()`, `Lock.tryLock()` и `Condition.await()`.

На рис. 14.3 показаны состояния, в которых может находиться поток, а также возможные переходы между ними. Когда поток находится в состоянии блокировки или ожидания (и, конечно, когда он завершается), к запуску планируется другой поток. А когда поток повторно активизируется (например, по истечении времени ожидания или в том случае, если ему удастся захватить блокировку), планировщик потоков сравнивает его приоритет с приоритетом выполняющихся в данный момент потоков. Если приоритет данного потока ниже, он приостанавливается и запускается новый поток.

Завершенные потоки

Поток завершается по одной из следующих причин.

- Прекращает свое существование естественным образом при нормальном завершении метода `run()`.
- Прекращает свое существование внезапно, поскольку неперехваченное исключение прерывает выполнение метода `run()`.

В частности, поток можно уничтожить, вызвав его метод `stop()`. Этот метод генерирует объект ошибки типа `ThreadDeath`, который уничтожает поток. Но метод `stop()` не рекомендован к применению, и поэтому следует избегать его применения в прикладном коде.

`java.lang.Thread 1.0`

- `void join()`
Ожидает завершения указанного потока.
- `void join(long millis)`
Ожидает завершения указанного потока или истечения заданного периода времени.

- Thread.State getState() 5.0

Получает состояние данного потока. Может принимать значения NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING или TERMINATED.

- void stop()

Останавливает поток. Этот метод не рекомендован к применению.

- void suspend()

Временно приостанавливает поток. Этот метод не рекомендован к применению.

- void resume()

Возобновляет поток. Вызывается только после вызова метода suspend(). Этот метод не рекомендован к применению.

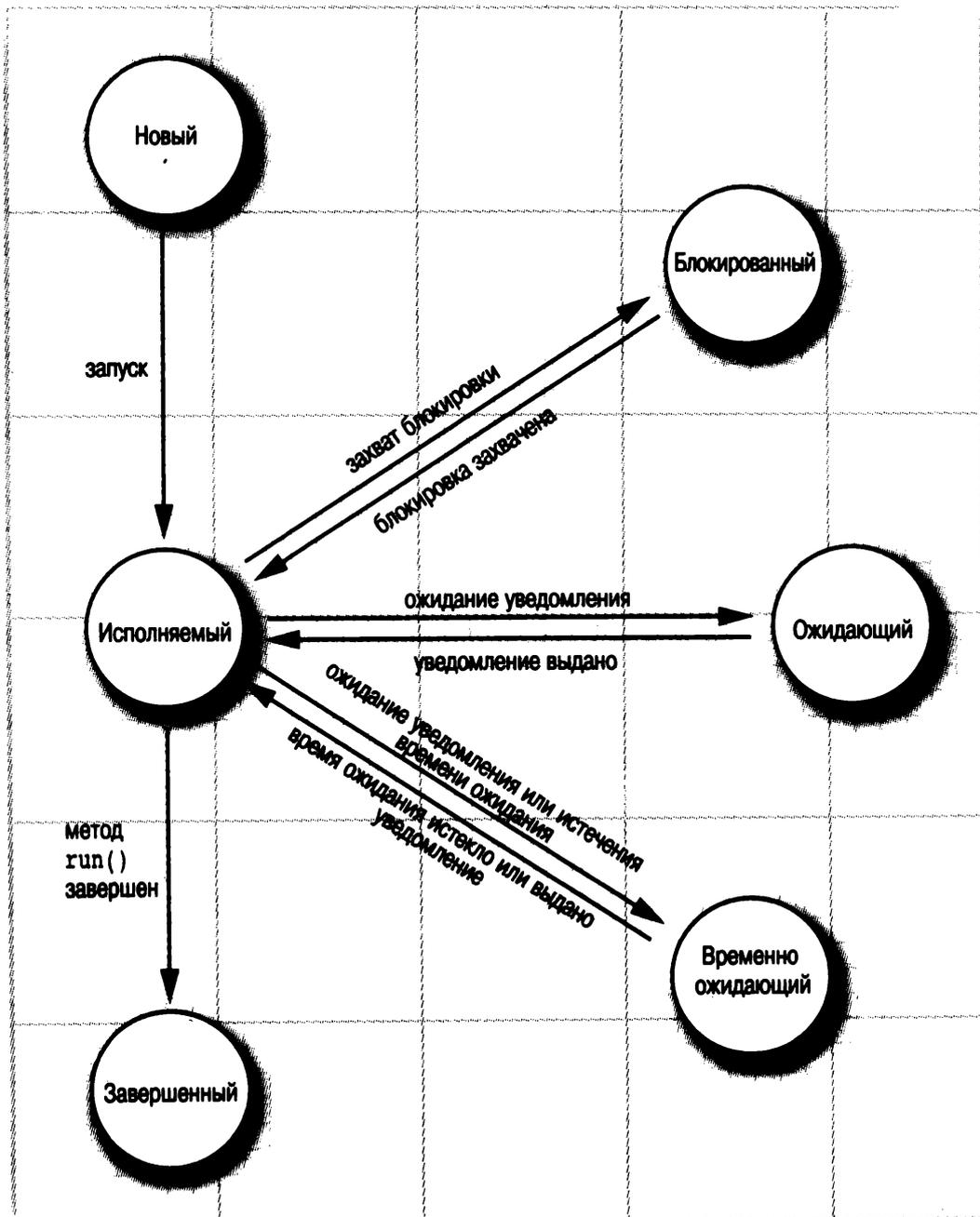


Рис. 14.3. Состояния потока

Свойства потоков

В последующих разделах рассматриваются разнообразные свойства потоков: приоритеты потоков, потоковые демоны, группы потоков и обработчики необрабатываемых исключений.

Приоритеты потоков

В языке программирования Java каждый поток имеет свой *приоритет*. По умолчанию поток наследует приоритет того потока, который его создал. Повысить или понизить приоритет любого потока можно, вызвав метод `setPriority()`. А установить приоритет можно, указав любое значение в пределах от `MIN_PRIORITY` (определено в классе `Thread` равным 1) до `MAX_PRIORITY` (равно 10). Обычному приоритету соответствует значение `NORM_PRIORITY`, равное 5.

Всякий раз, когда планировщик потоков выбирает новый поток для выполнения, он предпочитает потоки с более высоким приоритетом. Но приоритеты потоков *сильно зависят* от системы. Когда виртуальная машина полагается на реализацию потоков средствами главной платформы, на которой она выполняется, приоритеты потоков Java привязываются к уровням приоритетов этой платформы, где может быть больше или меньше уровней приоритетов.

Например, в Windows предусмотрено семь уровней приоритетов. Некоторые приоритеты Java привязываются к тому же самому уровню приоритета операционной системы. В виртуальной машине Oracle для Linux приоритеты потоков вообще игнорируются. Все потоки имеют одинаковый приоритет.

Начинающие программисты иногда злоупотребляют приоритетами потоков. Но имеется не так уж и много причин для того, чтобы манипулировать приоритетами. Поэтому не рекомендуется строить свои программы таким образом, чтобы их правильное функционирование зависело от уровней приоритетов.



ВНИМАНИЕ! Если вы все-таки используете приоритеты в своих программах, вам следует знать об ошибках, распространенных среди начинающих. Так, если имеется несколько потоков с высоким приоритетом, которые не перестают быть активными, то менее приоритетные потоки могут вообще не выполняться. Всякий раз, когда планировщик потоков решает запустить новый поток, он выбирает сначала среди наиболее высокоприоритетных потоков, даже если они могут полностью подавить активность низкоприоритетных потоков.

`java.lang.Thread 1.0`

- `void setPriority(int newPriority)`

Устанавливает приоритет потока. Приоритет должен находиться в пределах от `Thread.MIN_PRIORITY` до `Thread.MAX_PRIORITY`. Для нормального приоритета указывается значение `Thread.NORM_PRIORITY`.

- `static int MIN_PRIORITY`

Минимальный приоритет, который может иметь объект типа `Thread`. Значение минимального приоритета равно 1.

- `static int NORM_PRIORITY`

Приоритет объекта типа `Thread` по умолчанию. Значение нормального приоритета по умолчанию равно 5.

- `static int MAX_PRIORITY`

Максимальный приоритет, который может иметь объект типа `Thread`. Значение максимального приоритета равно 10.

- `static void yield()`

Заставляет текущий выполняющийся поток уступить управление. Если есть другие исполняемые потоки с приоритетом не ниже приоритета данного потока, они будут запланированы для выполнения следующими. Следует иметь в виду, что этот метод — статический.

Потоковые демоны

Превратить поток в *потоковый демон* можно, сделав следующий вызов:

```
t.setDaemon(true);
```

Правда, в таком потоке нет ничего демонического. Демон — это лишь поток, у которого нет других целей, кроме служения другим. В качестве примера можно привести потоки таймера, посылающие регулярные “такты” другим потокам, или же потоки, очищающие устаревшие записи в кеше. Когда остаются только потоковые демоны, виртуальная машина завершает работу. Нет смысла продолжать выполнение программы, когда все оставшиеся потоки являются демонами.

Потоковые демоны иногда ошибочно используются начинающими программистами, которые не хотят думать о действиях, завершающих программу. Но такой подход не безопасен. Потоковый демон вообще не должен обращаться к такому постоянному ресурсу, как файл или база данных, поскольку он может быть прерван в любой момент, даже посередине операции.

java.lang.Thread 1.0

- `void setDaemon(boolean isDaemon)`

Помечает данный поток как демон или пользовательский поток. Этот метод должен вызываться перед запуском потока.

Обработчики необрабатываемых исключений

Метод `run()` потока не может генерировать никаких проверяемых исключений, но может быть прерван непроверяемым исключением. В этом случае поток уничтожается. Но такой конструкции `catch`, куда может распространиться исключение, не существует. Вместо этого, перед тем, как поток прекратит свое существование, исключение передается обработчику необрабатываемых исключений. Такой обработчик должен относиться к классу, реализующему интерфейс `Thread.UncaughtExceptionHandler`. У этого интерфейса имеется следующий единственный метод:

```
void uncaughtException(Thread t, Throwable e)
```

Этот обработчик можно установить в любой поток с помощью метода `setUncaughtExceptionHandler()`. Кроме того, можно установить обработчик по умолчанию для всех потоков с помощью статического метода `setDefaultUncaughtExceptionHandler()` из класса `Thread`. В заменяющем обработчике может использоваться прикладной интерфейс API протоколирования для отправки отчетов о необрабатываемых исключениях в файл протокола.

Если не установить обработчик по умолчанию, то такой обработчик оказывается пустым (`null`). Но если не установить обработчик для отдельного потока, то им становится объект потока типа `ThreadGroup`.



НА ЗАМЕТКУ! Группа потоков — это коллекция потоков, которой можно управлять совместно. По умолчанию все создаваемые потоки относятся к одной и той же группе потоков, но можно устанавливать и другие группы. Теперь в Java имеются более совершенные средства для выполнения операций над коллекциями потоков, поэтому пользоваться группами потоков в собственных программах не рекомендуется.

Класс `ThreadGroup` реализует интерфейс `Thread.UncaughtExceptionHandler`. Его метод `uncaughtException()` выполняет следующие действия.

1. Если у группы потоков имеется родительская группа, то вызывается метод `uncaughtException()` из этой родительской группы.
2. Иначе, если метод `Thread.getDefaultExceptionHandler()` возвращает непустой обработчик (т.е. не `null`), вызывается именно этот обработчик.
3. Иначе, если объект типа `Throwable` является экземпляром класса `ThreadDeath`, ничего не происходит.
4. Иначе имя потока и трассировка стека объекта типа `Throwable` выводятся в поток сообщений об ошибках `System.err`.

Именно такую трассировку стека вы, без сомнения, видели и еще не раз увидите в своих программах.

java.lang.Thread 1.0

- `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) 5.0`
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler() 5.0`
Устанавливают или получают обработчик по умолчанию для необрабатываемых исключений.
- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) 5.0`
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler() 5.0`
Устанавливают или получают обработчик для необрабатываемых исключений. Если обработчик вообще не установлен, таким обработчиком становится объект группы потоков.

java.lang.Thread.UncaughtExceptionHandler 5.0

- void uncaughtException(Thread t, Throwable e)

Определяется для протоколирования специального отчета по завершении потока с необрабатываемым исключением.

Параметры: *t* Поток, который был завершен с необрабатываемым исключением
e Объект необрабатываемого исключения

java.lang.ThreadGroup 1.0

- void uncaughtException(Thread t, Throwable e)

Этот метод вызывается из родительской группы потоков, если таковая имеется, или же вызывается обработчик по умолчанию из класса Thread, если таковой имеется, а иначе выводится трассировка стека в стандартный поток сообщений об ошибках. (Но если *e* — объект типа ThreadDeath, то трассировка стека подавляется. Объекты типа ThreadDeath формируются устаревшим методом stop().)

Синхронизация

В большинстве практических многопоточных приложений двум или более потокам приходится разделять общий доступ к одним и тем же данным. Что произойдет, если два потока имеют доступ к одному объекту и каждый из них вызывает метод, изменяющий состояние этого объекта? Нетрудно догадаться, что потоки станут наступать друг другу на пятки. В зависимости от порядка обращения к данным можно в конечном итоге получить поврежденный объект. Такая ситуация обычно называется *состоянием гонок*.

Пример состояния гонок

Чтобы избежать повреждения данных, совместно используемых многими потоками, нужно научиться *синхронизировать доступ* к ним. В этом разделе будет показано, что произойдет, если не применять синхронизацию. А в следующем разделе поясняется, как синхронизировать обращение к данным.

В следующем примере тестовой программы имитируется банк с многими счетами. В ней будут случайным образом формироваться транзакции, переводящие деньги с одного счета на другой. Каждый счет владеет одним потоком. А каждая транзакция перемещает произвольную сумму денег со счета, обслуживаемого текущим потоком, на другой произвольно выбираемый счет.

Код, имитирующий банк, достаточно прост. В нем определен класс Bank с методом transfer(). Этот метод перемещает некоторую сумму денег с одного счета на другой. (Не будем пока что обращать внимание на возможное появление отрицательных остатков на счетах.) Ниже приведен исходный код метода transfer() из класса Bank.

```
public void transfer(int from, int to, double amount)
    // ВНИМАНИЕ: вызывать этот метод из нескольких потоков небезопасно!
    {
        System.out.print(Thread.currentThread());
        accounts[from] -= amount;
        System.out.printf(" %10.2f from %d to %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
    }
}
```

Ниже приведен исходный код класса `TransferRunnable`. Его метод `run()` выполняет перевод денег с фиксированного банковского счета. В каждой транзакции метод `run()` выбирает случайный целевой счет и произвольную сумму, вызывает метод `transfer()` для объекта `Bank`, а затем переводит поток в состояние ожидания.

```
class TransferRunnable implements Runnable
{
    . . .
    public void run()
    {
        try
        {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = maxAmount * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep((int) (DELAY * Math.random()));
        }
    }
}
```

Когда выполняется данная имитация банка, неизвестно, какая именно сумма находится на любом банковском счете в произвольный момент времени. Но в то же время известно, что общая сумма денег по всем счетам должна оставаться неизменной, поскольку деньги только переводятся с одного счета на другой, а не снимаются окончательно.

В конце каждой транзакции метод `transfer()` заново вычисляет итоговую сумму на счетах и выводит ее. Данная программа вообще не прекращает выполняться. Чтобы удалить ее, следует нажать комбинацию клавиш `<Ctrl+C>`. Ниже приведен типичный результат, выводимый данной программой.

```
. . .
Thread[Thread-11,5,main] 588.48 from 11 to 44 Total Balance: 100000.00
Thread[Thread-12,5,main] 976.11 from 12 to 22 Total Balance: 100000.00
Thread[Thread-14,5,main] 521.51 from 14 to 22 Total Balance: 100000.00
Thread[Thread-13,5,main] 359.89 from 13 to 81 Total Balance: 100000.00
. . .
Thread[Thread-36,5,main] 401.71 from 36 to 73 Total Balance: 99291.06
Thread[Thread-35,5,main] 691.46 from 35 to 77 Total Balance: 99291.06
Thread[Thread-37,5,main] 78.64 from 37 to 3 Total Balance: 99291.06
Thread[Thread-34,5,main] 197.11 from 34 to 69 Total Balance: 99291.06
Thread[Thread-36,5,main] 85.96 from 36 to 4 Total Balance: 99291.06
. . .
Thread[Thread-4,5,main]Thread[Thread-33,5,main] 7.31 from 31 to 32
Total Balance: 99979.24
    627.50 from 4 to 5 Total Balance: 99979.24
. . .
```

Как видите, что-то в этой программе пошло не так. В течение нескольких транзакций общий баланс в имитируемом банке оставался равным сумме \$100000, что совершенно верно, поскольку первоначально было 100 счетов по \$1000 на каждом.

Но через некоторое время общий баланс немного изменился. Запустив эту программу на выполнение, вы можете обнаружить, что ошибка возникнет очень быстро или же общий баланс будет нарушен нескоро. Такая ситуация не внушает доверия, и вы вряд ли захотите положить свои заработанные тяжким трудом денежки в такой банк!

В листингах 14.5–14.7 представлен весь исходный код данной программы. Попробуйте сами найти ошибку в этом коде. А причины ее появления будут раскрыты в следующем разделе.

Листинг 14.5. Исходный код из файла `unsynch/UnsynchBankTest.java`

```
1 package unsynch;
2
3 /**
4  * В этой программе демонстрируется нарушение данных при
5  * произвольном доступе к структуре данных из многих потоков
6  * @version 1.30 2004-08-01*
7  * @author Cay Horstmann
8  */
9 public class UnsynchBankTest
10 {
11     public static final int NACCOUNTS = 100;
12     public static final double INITIAL_BALANCE = 1000;
13
14     public static void main(String[] args)
15     {
16         Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
17         int i;
18         for (i = 0; i < NACCOUNTS; i++)
19         {
20             TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
21             Thread t = new Thread(r);
22             t.start();
23         }
24     }
25 }
```

Листинг 14.6. Исходный код из файла `unsynch/Bank.java`

```
1 package unsynch;
2
3 /**
4  * Имитируемый банк с целым рядом счетов
5  * @version 1.30 2004-08-01
6  * @author Cay Horstmann
7  */
8 public class Bank
9 {
10     private final double[] accounts;
11
12     /**
13      * Конструирует объект банка
14      * @param n Количество счетов
15      * @param initialBalance Первоначальный остаток на каждом счете
16      */
17     public Bank(int n, double initialBalance)
```

```
18 {
19     accounts = new double[n];
20     for (int i = 0; i < accounts.length; i++)
21         accounts[i] = initialBalance;
22 }
23
24 /**
25  * Переводит деньги с одного счета на другой
26  * @param from Счет, с которого переводятся деньги
27  * @param to Счет, на который переводятся деньги
28  * @param amount Сумма перевода
29  */
30 public void transfer(int from, int to, double amount)
31 {
32     if (accounts[from] < amount) return;
33     System.out.print(Thread.currentThread());
34     accounts[from] -= amount;
35     System.out.printf(" %10.2f from %d to %d", amount, from, to);
36     accounts[to] += amount;
37     System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
38 }
39
40 /**
41  * Получает сумму остатков на всех счетах
42  * @return Возвращает общий баланс
43  */
44 public double getTotalBalance()
45 {
46     double sum = 0;
47
48     for (double a : accounts)
49         sum += a;
50
51     return sum;
52 }
53
54 /**
55  * Получает количество счетов в банке
56  * @return Возвращает количество счетов
57  */
58 public int size()
59 {
60     return accounts.length;
61 }
62 }
```

Листинг 14.7. Исходный код из файла unsynch/TransferRunnable.java

```
1 package unsynch;
2
3 /**
4  * Исполняемый поток, в котором деньги переводятся
5  * с одного банковского счета на другой
6  * @version 1.30 2004-08-01
7  * @author Cay Horstmann
8  */
9 public class TransferRunnable implements Runnable
```

```
10 {
11     private Bank bank;
12     private int fromAccount;
13     private double maxAmount;
14     private int DELAY = 10;
15
16     /**
17      * Конструирует объект исполняемого потока
18      * @param b Банк, где деньги переводятся с одного счета на другой
19      * @param from Счет, с которого переводятся деньги
20      * @param max Максимальная сумма перевода в каждой транзакции
21      */
22     public TransferRunnable(Bank b, int from, double max)
23     {
24         bank = b;
25         fromAccount = from;
26         maxAmount = max;
27     }
28
29     public void run()
30     {
31         try
32         {
33             while (true)
34             {
35                 int toAccount = (int) (bank.size() * Math.random());
36                 double amount = maxAmount * Math.random();
37                 bank.transfer(fromAccount, toAccount, amount);
38                 Thread.sleep((int) (DELAY * Math.random()));
39             }
40         }
41         catch (InterruptedException e)
42         {
43         }
44     }
45 }
```

Объяснение причин, приводящих к состоянию гонок

В предыдущем разделе был рассмотрен пример программы, в которой несколько потоков обновляли остатки на банковских счетах. По истечении некоторого времени в ней накапливаются ошибки, а в итоге некоторая сумма теряется или появляется неизвестно откуда. Подобная ошибка возникает в том случае, если два потока пытаются одновременно обновлять один и тот же счет. Допустим, два потока одновременно выполняют следующую операцию:

```
accounts[to] += amount;
```

Дело в том, что такие операции не являются *атомарными*. Приведенная выше операция может быть выполнена поэтапно следующим образом.

1. Загрузить значение из элемента массива `accounts[to]` в регистр.
2. Добавить значение `amount`.
3. Переместить результат обратно в элемент массива `accounts[to]`.

А теперь представим, что в первом потоке выполняются операции в пп. 1 и 2, после чего его выполнение приостанавливается. Допустим, второй поток выходит из состояния ожидания и обновляет тот же самый элемент в массиве `accounts`. Затем выходит из состояния ожидания первый поток и выполняет операцию в п. 3. Такое действие уничтожает изменения, внесенные во втором потоке. В итоге общий баланс оказывается подсчитанным неверно (рис. 14.4). Такое нарушение данных обнаруживается в рассматриваемой здесь тестовой программе. (Разумеется, существует вероятность получить сигнал ложной тревоги, если поток будет прерван во время тестирования!)

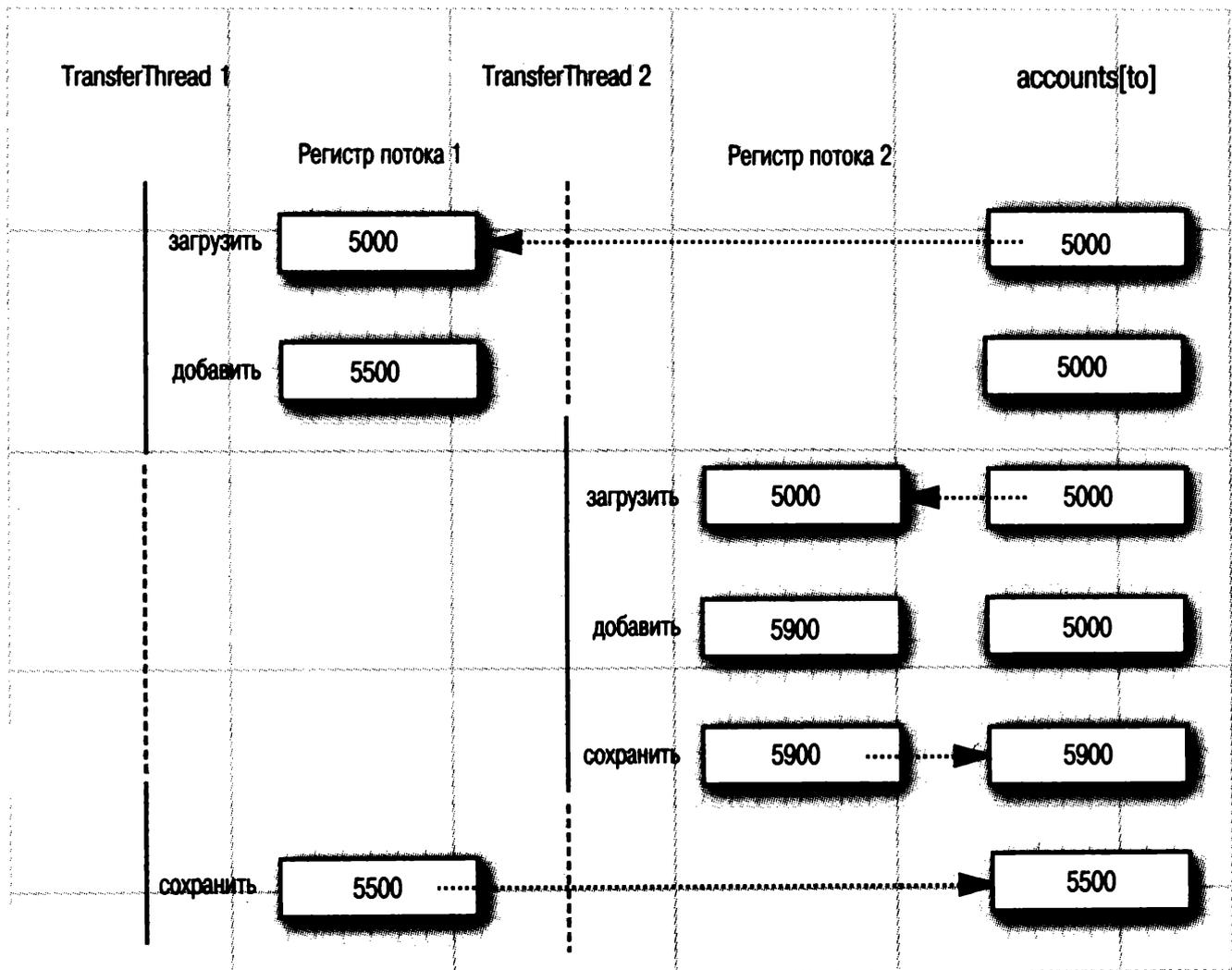


Рис. 14.4. Одновременный доступ из двух потоков



НА ЗАМЕТКУ! Байт-коды, выполняемые виртуальной машиной в классе `Bank`, можно просмотреть. Для этого достаточно ввести следующую команду декомпиляции файла `Bank.class`:

```
javap -c -v Bank
```

Например, следующая строка кода:

```
accounts[to] += amount;
```

транслируется в приведенные ниже байт-коды.

```
aload_0
getfield #2; // поле accounts: [D
iload_2
```


Такая конструкция гарантирует, что только один поток в единицу времени сможет войти в критический раздел кода. Как только один поток заблокирует объект блокировки, никакой другой поток не сможет обойти вызов метода `lock()`. И если другие потоки попытаются вызвать метод `lock()`, они будут деактивизированы до тех пор, пока первый поток не снимет блокировку с объекта блокировки.



ВНИМАНИЕ! Крайне важно разместить вызов метода `unlock()` в блоке `finally`. Если код в критическом разделе сгенерирует исключение, блокировка должна быть непременно снята. В противном случае другие потоки будут заблокированы навсегда.



НА ЗАМЕТКУ! Пользоваться блокировками вместе с оператором `try` с ресурсами нельзя. Ведь метод разблокировки не называется `close()`. Но даже если бы он так назывался, то его все равно нельзя было бы применять вместе с оператором `try` с ресурсами, поскольку в заголовке этого оператора предполагается объявление новой переменной. Но производя блокировку, следует непременно использовать одну и ту же переменную, общую для всех потоков.

Воспользуемся блокировкой для защиты метода `transfer()` из класса `Bank`.

```
public class Bank
{
    private Lock bankLock = new ReentrantLock(); // объект класса
        // ReentrantLock, реализующего интерфейс Lock
    . . .
    public void transfer(int from, int to, int amount)
    {
        bankLock.lock();
        try
        {
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f from %d to %d", amount, from, to);
            accounts[to] += amount;
            System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
        }
        finally
        {
            bankLock.unlock();
        }
    }
}
```

Предположим, что в одном потоке вызывается метод `transfer()` и до его завершения этот поток приостанавливается, а во втором потоке также вызывается метод `transfer()`. Второй поток не сможет захватить блокировку и останется заблокированным при вызове метода `lock()`. Напротив, он будет деактивизирован и вынужден ждать до тех пор, пока выполнение метода `transfer()` не завершится в первом потоке. И только тогда, когда первый поток снимет блокировку, второй поток сможет продолжить свое выполнение (рис. 14.5).

Опробуйте описанный выше механизм синхронизации потоков, введя блокирующий код в метод `transfer()` и запустив рассматриваемую здесь программу снова. Вы можете прогонять ее бесконечно, но общий баланс банка на этот раз не будет нарушен.

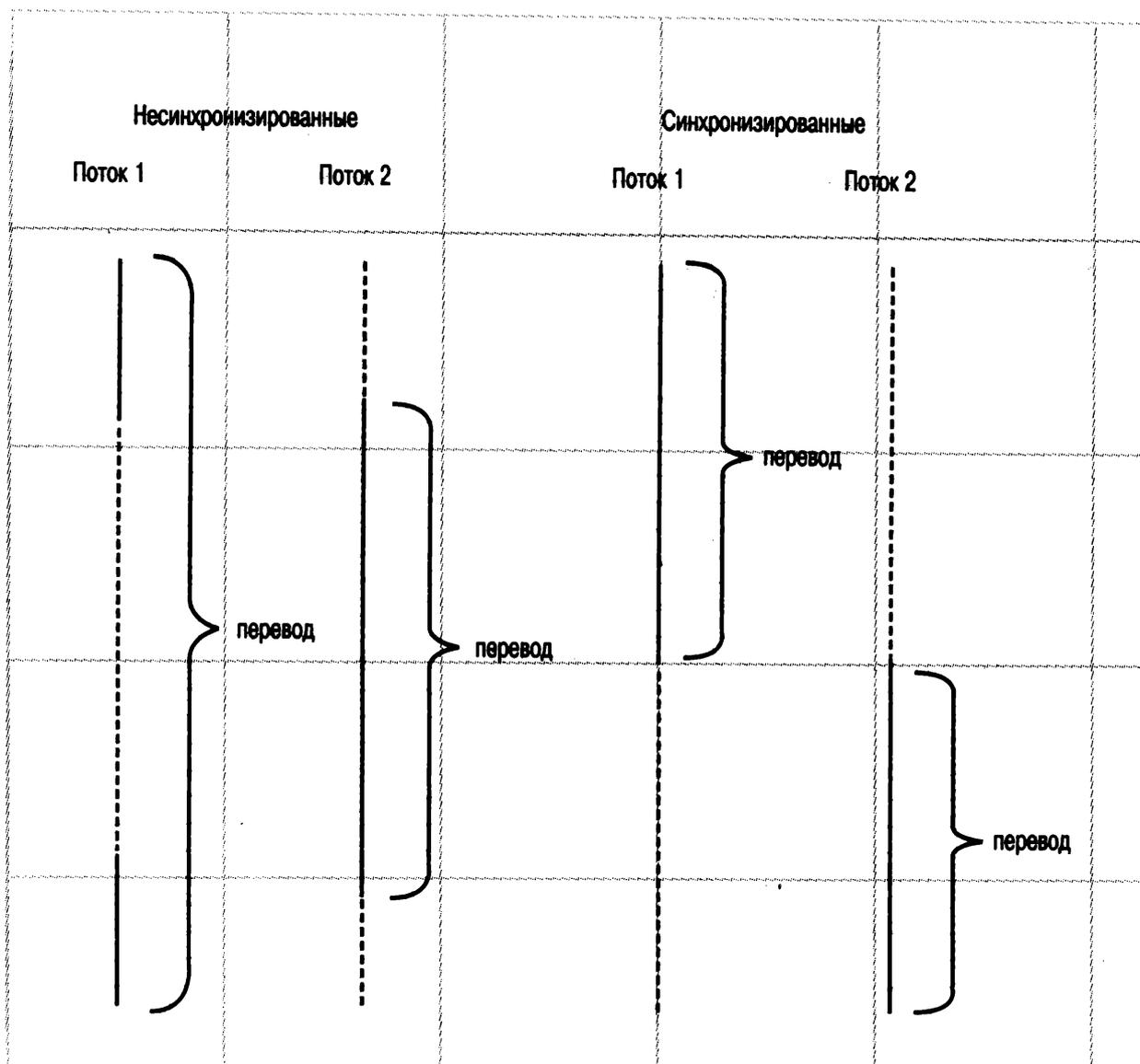


Рис. 14.5. Сравнение синхронизированных и не синхронизированных потоков

Следует, однако, иметь в виду, что у каждого объекта типа `Bank` имеется свой собственный объект типа `ReentrantLock`. Если два потока попытаются обратиться к одному и тому же объекту типа `Bank`, блокировка послужит для сериализации доступа. Но если два потока обращаются к разным объектам типа `Bank`, то каждый из них захватывает свою блокировку и ни один из потоков не блокируется. Так и должно быть, потому что потоки никак не мешают друг другу, оперируя разными экземплярами класса `Bank`.

Блокировка называется *реентерабельной*, потому что поток может повторно захватывать блокировку, которой он уже владеет. Для блокировки предусмотрен счетчик захватов, отслеживающий вложенные вызовы метода `lock()`. И для каждого вызова `lock()` в потоке должен быть вызван метод `unlock()`, чтобы, в конце концов, снять блокировку. Благодаря этому средству код, защищенный блокировкой, может вызывать другой метод, использующий ту же самую блокировку.

Например, метод `transfer()` вызывает метод `getTotalBalance()`, который также блокирует объект `bankLock`, у которого теперь значение счетчика захватов равно 2. Когда метод `getTotalBalance()` завершается, значение счетчика захватов возвращается к 1. При выходе из метода `transfer()` счетчик захватов имеет значение 0, и поток снимает блокировку.

Обычно требуется защищать блоки кода, обновляющие и проверяющие разделяемый объект. Тогда можно не сомневаться, что эти операции завершатся, прежде чем другой поток сможет использовать тот же самый объект.



ВНИМАНИЕ! Будьте внимательны, чтобы код в критическом разделе программы не был пропущен из-за генерирования исключения. Если исключение сгенерировано до конца критического раздела кода, блокировка будет снята в блоке `finally`, но объект может оказаться в поврежденном состоянии.

`java.util.concurrent.locks.Lock 5.0`

- `void lock()`
Захватывает блокировку. Если же в данный момент она захвачена другим потоком, текущий поток блокируется.
- `void unlock()`
Снимает блокировку.

`java.util.concurrent.locks.ReentrantLock 5.0`

- `ReentrantLock()`
Конструирует объект реентерабельной блокировки, которая может быть использована для защиты критического раздела кода.
- `ReentrantLock(boolean fair)`
Конструирует объект реентерабельной блокировки с заданным правилом равноправия. Равноправная блокировка отдает предпочтение потоку, ожидающему дольше всех. Но такое равноправие может отрицательно сказаться на производительности. Поэтому по умолчанию равноправия от блокировок не требуется.



ВНИМАНИЕ! На первый взгляд, лучше, чтобы блокировка была равноправной, но равноправные блокировки действуют *намного* медленнее обычных. Разрешить равноправную блокировку вы можете только в том случае, если точно знаете, что делаете, и имеете на то особые причины. Даже если вы используете равноправную блокировку, у вас все равно нет никаких гарантий, что планировщик потоков будет также соблюдать правило равноправия. Если планировщик потоков решит пренебречь потоком, который длительное время ожидает снятия блокировки, то никакое равноправие блокировок не поможет.

Объекты условий

Нередко поток входит в критический раздел кода только для того, чтобы обнаружить, что он не может продолжить свое выполнение до тех пор, пока не будет соблюдено определенное условие. В подобных случаях для управления потоками, которые захватили блокировку, но не могут выполнить полезные действия, служит *объект условия*. В этом разделе будет представлена реализация объектов условий в

библиотеке Java (по ряду исторических причин объекты условий нередко называются *условными переменными*).

Попробуем усовершенствовать рассматриваемую здесь программу имитации банка. Не будем перемещать деньги со счета, если он не содержит достаточной суммы, чтобы покрыть расходы на перевод. Но для этого не годится код, подобный следующему:

```
if (bank.getBalance(from) >= amount)
    .bank.transfer(from, to, amount)
```

Ведь вполне возможно, что текущий поток будет деактивизирован в промежутке между успешным выполнением проверки и вызовом метода `transfer()`:

```
if (bank.getBalance(from) >= amount)
    // поток может быть деактивирован в этом месте кода
    bank.transfer(from, to, amount);
```

На момент возобновления работы потока остаток на счете может измениться, т.е. уменьшиться ниже допустимого предела. Поэтому нужно каким-то образом гарантировать, что никакой другой поток не сможет изменить остаток на счете между его проверкой и переводом денег. Для этого придется защитить как проверку остатка на счете, так и сам перевод денег с помощью следующей блокировки:

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
        {
            // ожидать
            . . .
        }
        // перевести деньги
        . . .
    }
    finally
    {
        bankLock.unlock();
    }
}
```

Что делать дальше, если на счете нет достаточной суммы? Ожидать до тех пор, пока счет не будет пополнен в каком-то другом потоке. Но ведь данный поток только что получил монопольный доступ к объекту `bankLock`, так что ни в одном другом потоке нет возможности пополнить счет. И здесь на помощь приходит объект условия.

С объектом блокировки может быть связан один или несколько объектов условий, которые получены с помощью метода `newCondition()`. Каждому объекту условия можно присвоить имя, напоминающее об условии, которое он представляет. Например, объект, представляющий условие “достаточных средств”, устанавливается следующим образом:

```
class Bank
{
    private Condition sufficientFunds;
    . . .
    public Bank()
    {
```

```
        . . . . .
        sufficientFunds = bankLock.newCondition();
    }
}
```

Если в методе `transfer()` будет обнаружено, что средств на счете недостаточно, он сделает следующий вызов:

```
sufficientFunds.await();
```

Текущий поток теперь деактивизирован и снимает блокировку. Это дает возможность пополнить счет в другом потоке.

Имеется существенное отличие между потоком, ожидающим возможности захватить блокировку, и потоком, который вызвал метод `await()`. Как только поток вызовет метод `await()`, он войдет в набор ожидания, установленный для данного условия. Поток *не* становится исполняемым, когда оказывается доступной блокировка. Вместо этого он остается деактивизированным до тех пор, пока другой поток не вызовет метод `signalAll()` по тому же условию.

Когда перевод денег будет произведен в другом потоке, в нем должен будет сделан следующий вызов:

```
sufficientFunds.signalAll();
```

При этом вызове активизируются все потоки, ожидающие данного условия. Когда потоки удаляются из набора ожидания, они опять становятся исполняемыми, и в конечном итоге планировщик потоков активизирует их снова. В этот момент они попытаются повторно захватить объект блокировки. И как только он окажется доступным, один из этих потоков захватит блокировку и *продолжит* свое выполнение с *того места*, где он *остановился*, получив управление после вызова метода `await()`.

В этот момент поток снова должен проверить условие. Но нет никаких гарантий, что условие теперь выполнится. Ведь метод `signalAll()` просто сигнализирует ожидающим потокам о том, что условие теперь *может* быть удовлетворено и что его стоит проверить заново.



НА ЗАМЕТКУ! Вообще говоря, вызов метода `await()` должен быть введен в цикл следующей формы:

```
while (!(можно · продолжить))
    условие.await();
```

Крайне важно, чтобы в конечном итоге метод `signalAll()` был вызван в каком-то другом потоке. Когда метод `await()` вызывается в потоке, последний не имеет возможности повторно активизировать самого себя. И здесь он полностью полагается на другие потоки. Если ни один из них не позаботится о повторной активизации ожидающего потока, его выполнение никогда не возобновится. Это может привести к неприятной ситуации *взаимной блокировки*. Если все прочие потоки будут заблокированы, а метод `await()` будет вызван в последнем активном потоке без разблокировки какого-нибудь другого потока, этот поток также окажется заблокированным. И тогда не останется ни одного потока, где можно было бы разблокировать другие потоки, а следовательно, программа зависнет.

Когда же следует вызывать метод `signalAll()`? Существует эмпирическое правило: вызывать этот метод при таком изменении состояния объекта, которое может быть выгодно ожидающим потокам. Например, всякий раз, когда изменяются

остатки на счетах, ожидающим потокам следует давать очередную возможность для проверки остатков на счетах. В данном примере метод `signalAll()` вызывается по завершении перевода денег, как показано ниже.

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
            sufficientFunds.await();
        // перевести средства
        . . .
        sufficientFunds.signalAll();
    }
    finally
    {
        bankLock.unlock();
    }
}
```

Следует, однако, иметь в виду, что вызов метода `signalAll()` не влечет за собой немедленной активизации ожидающего потока. Он лишь разблокирует ожидающие потоки, чтобы они могли соперничать за объект блокировки после того, как текущий поток выйдет из синхронизированного метода.

Другой метод, `signal()`, разблокирует только один поток из набора ожидания, выбирая его случайным образом. Это более эффективно, чем разблокировать все потоки, хотя здесь существует определенная опасность. Если случайно выбранный поток обнаружит, что еще не может продолжить свое выполнение, он вновь заблокируется. И если никакой другой поток не вызовет снова метод `signal()`, то система перейдет в состояние взаимной блокировки.



ВНИМАНИЕ! По условию в потоке может быть вызван только метод: `await()`, `signalAll()` или `signal()`, когда этот поток владеет блокировкой по данному условию.

Запустив на выполнение видоизмененный вариант программы из листинга 14.8, имитирующей банк, вы обнаружите, что теперь она работает правильно. Общий баланс в \$100000 сохраняется неизменным, и ни один счет не имеет отрицательного остатка. (Но для того чтобы прервать выполнение программы, вам снова придется нажать комбинацию клавиш <Ctrl+C>.) Вы можете также заметить, что программа работает немного медленнее — это та цена, которую приходится платить за дополнительные служебные операции, связанные с механизмом синхронизации.

На практике правильно воспользоваться условиями не так-то просто. Поэтому, прежде чем пытаться реализовать собственные объекты условий, стоит рассмотреть применение конструкций, описанных далее в разделе “Синхронизаторы”.

Листинг 14.8. Исходный код из файла `synch/Bank.java`

```
1 package synch;
2
3 import java.util.concurrent.locks.*;
4
5 /**
```

```
6  * Программа, имитирующая банк со счетами и использующая блокировки
7  * для организации последовательного доступа к остаткам на счетах
8  * @version 1.30 2004-08-01
9  * @author Cay Horstmann
10 */
11 public class Bank
12 {
13     private final double[] accounts;
14     private Lock bankLock;
15     private Condition sufficientFunds;
16     /**
17      * Конструирует объект банка
18      * @param n Количество счетов
19      * @param initialBalance Первоначальный остаток на каждом счете
20      */
21     public Bank(int n, double initialBalance)
22     {
23         accounts = new double[n];
24         for (int i = 0; i < accounts.length; i++)
25             accounts[i] = initialBalance;
26         bankLock = new ReentrantLock();
27         sufficientFunds = bankLock.newCondition();
28     }
29     /**
30      * Переводит деньги с одного счета на другой
31      * @param from Счет, с которого переводятся деньги
32      * @param to Счет, на который переводятся деньги
33      * @param amount Сумма перевода
34      */
35     public void transfer(int from, int to, double amount)
36         throws InterruptedException
37     {
38         bankLock.lock();
39         try
40         {
41             while (accounts[from] < amount)
42                 sufficientFunds.await();
43             System.out.print(Thread.currentThread());
44             accounts[from] -= amount;
45             System.out.printf(" %10.2f from %d to %d", amount, from, to);
46             accounts[to] += amount;
47             System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
48             sufficientFunds.signalAll();
49         }
50         finally
51         {
52             bankLock.unlock();
53         }
54     }
55     /**
56      * Получает сумму остатков на всех счетах
57      * @return Возвращает общий баланс
58      */
59     public double getTotalBalance()
60     {
61         bankLock.lock();
62         try
63         {
64             double sum = 0;
```

```
66
67     for (double a : accounts)
68         sum += a;
69
70     return sum;
71 }
72 finally
73 {
74     bankLock.unlock();
75 }
76 }
77 /**
78  * Получает количество счетов в банке
79  * @return Возвращает количество счетов
80  */
81 public int size()
82 {
83     return accounts.length;
84 }
85 }
```

java.util.concurrent.locks.Lock 5.0

- `Condition newCondition()`
Возвращает объект условия, связанный с данной блокировкой.

java.util.concurrent.locks.Condition 5.0

- `void await()`
Вводит поток в набор ожидания по данному условию.
- `void signalAll()`
Разблокирует все потоки в наборе ожидания по данному условию.
- `void signal()`
Разблокирует один произвольно выбранный поток в наборе ожидания по данному условию.

Ключевое слово `synchronized`

В предыдущих разделах было показано, как пользоваться объектами блокировки типа `Lock` и условия типа `Condition`. Прежде чем двигаться дальше, подведем краткие итоги, перечислив ключевые моменты, касающиеся блокировок и условий.

- Блокировка защищает критические разделы кода, позволяя выполнять этот код только одному потоку в единицу времени.
- Блокировка управляет потоками, которые пытаются войти в защищенный раздел кода.
- Каждый объект условия управляет потоками, которые вошли в защищенный раздел кода, но пока еще не в состоянии продолжить свое выполнение.

Интерфейсы `Lock` и `Condition` предоставляют программистам высокую степень контроля над блокировками. Но зачастую такой контроль не требуется, и оказывается достаточно механизма, встроенного в язык `Java`. Еще со времен версии 1.0 *каждый объект* в `Java` обладает встроенной блокировкой. Если метод объявлен с ключевым словом `synchronized`, то блокировка объекта защищает весь этот метод. Следовательно, поток должен захватить встроенную блокировку объекта, чтобы вызвать такой метод.

Иными словами, следующий фрагмент кода:

```
public synchronized void method()
{
    тело метода
}
```

равнозначен приведенному ниже фрагменту кода.

```
public void method()
{
    this.intrinsicLock.lock();
    try
    {
        тело метода
    }
    finally { this.intrinsicLock.unlock(); }
}
```

Например, вместо явной блокировки можно просто объявить метод `transfer()` из класса `Bank` как `synchronized`. Встроенная блокировка объектов блокировки имеет единственное связанное с ней условие. Метод `wait()` вводит поток в набор ожидания, а методы `notifyAll()/notify()` разблокируют ожидающие потоки. Иными словами, вызов метода `wait()` или `notifyAll()` равнозначен следующему коду:

```
встроенноеУсловие.await();
встроенноеУсловие.signalAll();
```



НА ЗАМЕТКУ! Методы `wait()`, `notifyAll()` и `notify()` являются конечными (`final`) методами из класса `Object`. А методы из интерфейса `Condition` должны именоваться `await`, `signalAll` и `signal`, чтобы не вступать в конфликт с этими методами.

Например, класс `Bank` можно реализовать только языковыми средствами `Java` следующим образом:

```
class Bank
{
    private double[] accounts;

    public synchronized void transfer(int from, int to, int amount)
        throws InterruptedException
    {
        while (accounts[from] < amount)
            wait(); // ожидать по единственному условию
                // встроенной блокировки объектов
            accounts[from] -= amount;
            accounts[to] += amount;
            notifyAll(); // уведомить все потоки, ожидающие по данному условию
        }
    public synchronized double getTotalBalance() { . . . }
}
```

Как видите, применение ключевого слова `synchronized` порождает намного более краткий код. Разумеется, чтобы понять такой код, нужно знать, что каждый объект обладает встроенной блокировкой и что эта блокировка имеет встроенное условие. Блокировка управляет потоками, которые пытаются войти в метод `synchronized`. А условие управляет потоками, вызвавшими метод `wait()`.



СОВЕТ. Синхронизированные методы относительно просты. Но начинающие программировать на Java нередко испытывают затруднения в обращении с условиями. Поэтому, прежде чем применять методы `wait()/notifyAll()`, рекомендуется ознакомиться с одной из конструкций, описанных в разделе “Синхронизаторы”.

Статические методы также допускается объявлять синхронизированными. Когда вызывается такой метод, он захватывает встроенную блокировку объекта соответствующего класса. Так, если в классе `Bank` имеется статический синхронизированный метод, при его вызове захватывается блокировка объекта типа `Bank.class`. В результате к этому объекту не может обратиться никакой другой поток и никакой другой синхронизированный статический метод того же класса.

Встроенным блокировкам и условиям присущи некоторые ограничения, в том числе приведенные ниже.

- Нельзя прервать поток, который пытается захватить блокировку.
- Нельзя указать время ожидания, пытаясь захватить блокировку.
- Наличие единственного условия на блокировку может оказаться неэффективным.

Что же лучше использовать в прикладном коде: объекты типа `Lock` и `Condition` или синхронизированные методы? Ниже приведены некоторые рекомендации, которые дают ответ на этот вопрос.

- Лучше не пользоваться ни объектами типа `Lock/Condition`, ни ключевым словом `synchronized`. Зачастую вместо этого можно выбрать подходящий механизм из пакета `java.util.concurrent`, который организует блокировку автоматически. Например, в разделе “Блокирующие очереди” далее в этой главе будет показано, как пользоваться блокирующими очередями для синхронизации потоков, выполняющих общую задачу.
- Если ключевое слово `synchronized` подходит в конкретной ситуации, непременно воспользуйтесь им. В этом случае вам придется написать меньше кода, а следовательно, допустить меньше ошибок. В листинге 14.9 приведен пример очередного варианта программы, имитирующей банк и реализованной на основе синхронизированных методов.
- Пользуйтесь объектами типа `Lock/Condition`, если действительно нуждаетесь в дополнительных возможностях подобных конструкций.

Листинг 14.9. Исходный код из файла `synch2/Bank.java`

```
1 package synch2;
2
3 /**
4  * Программа, имитирующая банк со счетами и использующая
```

```
5  * примитивные языковые конструкции для синхронизации потоков
6  * @version 1.30 2004-08-01
7  * @author Cay Horstmann
8  */
9  public class Bank
10 {
11     private final double[] accounts;
12
13     /**
14      * Конструирует объект банка
15      * @param n Количество счетов
16      * @param initialBalance Первоначальный остаток на каждом счете
17      */
18     public Bank(int n, double initialBalance)
19     {
20         accounts = new double[n];
21         for (int i = 0; i < accounts.length; i++)
22             accounts[i] = initialBalance;
23     }
24
25     /**
26      * Переводит деньги с одного счета на другой
27      * @param from Счет, с которого переводятся деньги
28      * @param to Счет, на который переводятся деньги
29      * @param amount Сумма перевода
30      */
31     public synchronized void transfer(int from, int to, double amount)
32         throws InterruptedException
33     {
34         while (accounts[from] < amount)
35             wait();
36         System.out.print(Thread.currentThread());
37         accounts[from] -= amount;
38         System.out.printf(" %10.2f from %d to %d", amount, from, to);
39         accounts[to] += amount;
40         System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
41         notifyAll();
42     }
43
44     /**
45      * Получает сумму остатков на всех счетах
46      * @return Возвращает общий баланс
47      */
48     public synchronized double getTotalBalance()
49     {
50         double sum = 0;
51
52         for (double a : accounts)
53             sum += a;
54
55         return sum;
56     }
57
58     /**
59      * Получает сумму остатков на всех счетах
60      * @return Возвращает общий баланс
61      */
62     public int size()
```

```

63. {
64     return accounts.length;
65 }
66 }

```

java.lang.Object 1.0

- void notifyAll()

Разблокирует потоки, вызвавшие метод wait() для данного объекта. Этот метод может быть вызван только из тела синхронизированного метода или блока. Метод генерирует исключение типа IllegalMonitorStateException, если поток не владеет блокировкой данного объекта.

- void notify()

Разблокирует один произвольно выбранный поток среди потоков, вызвавших метод wait() для данного объекта. Этот метод может быть вызван только из тела синхронизированного метода или блока. Метод генерирует исключение типа IllegalMonitorStateException, если поток не владеет блокировкой данного объекта.

- void wait()

Вынуждает поток ожидать уведомления в течение указанного периода времени. Этот метод вызывается только из синхронизированного метода или блока. Он генерирует исключение типа IllegalMonitorStateException, если поток не владеет блокировкой данного объекта.

- void wait(long millis)

- void wait(long millis, int nanos)

Вынуждают поток ожидать уведомления в течение указанного периода времени. Эти методы вызываются только из синхронизированного метода или блока. Они генерируют исключение типа IllegalMonitorStateException, если поток не владеет блокировкой данного объекта.

Параметры: *millis* Количество миллисекунд ожидания

nanos Количество наносекунд ожидания, но не более 1000000

Синхронизированные блоки

Как упоминалось выше, у каждого объекта в Java имеется собственная встроенная блокировка. Поток может захватить эту блокировку, вызвав синхронизированный метод. Но есть и другой механизм захвата блокировки — вхождение в *синхронизированный блок*. Когда поток входит в блок в приведенной ниже форме, он захватывает блокировку объекта obj.

```

synchronized (obj) // это синтаксис синхронизированного блока
{
    критический раздел кода
}

```

Иногда в коде встречаются специальные блокировки вроде следующей:

```

public class Bank
{
    private double[] accounts;
    private Object lock = new Object();
    . . .
}

```

```
public void transfer(int from, int to, int amount)
{
    synchronized (lock) // специальная блокировка
    {
        accounts[from] -= amount;
        accounts[to] += amount;
    }
    System.out.println(. . .);
}
}
```

Здесь объект `lock` создается только для использования встроенной блокировки, которая имеется у каждого объекта в Java. Встроенной блокировкой объекта иногда пользуются для реализации дополнительных атомарных операций. Такая практика получила название *клиентской блокировки*. Рассмотрим для примера класс `Vector`, который реализует список и методы которого синхронизированы. А теперь допустим, что остатки на банковских счетах сохранены в объекте типа `Vector<Double>`. Ниже приведена наивная реализация метода `transfer()`.

```
public void transfer(
    Vector<Double> accounts, int from, int to, int amount) // ОШИБКА!
{
    accounts.set(from, accounts.get(from) - amount);
    accounts.set(to, accounts.get(to) + amount);
    System.out.println(. . .);
}
```

Методы `get()` и `set()` из класса `Vector` синхронизированы, но это вряд ли поможет. Вполне возможно, что поток будет приостановлен в методе `transfer()` по завершении первого вызова метода `get()`. В другом потоке может быть затем установлено иное значение на той же позиции. Блокировку можно захватить следующим образом:

```
public void transfer(Vector<Double> accounts, int from, int to, int amount)
{
    synchronized (accounts)
    {
        accounts.set(from, accounts.get(from) - amount);
        accounts.set(to, accounts.get(to) + amount);
    }
    System.out.println(. . .);
}
```

Такой подход вполне работоспособен, но он полностью зависит от того факта, что встроенная блокировка используется в классе `Vector` для всех его модифицирующих методов. Но так ли это на самом деле? В документации на класс `Vector` этого не обещается. Поэтому следует очень тщательно проанализировать его исходный код и надеяться, что в последующие его версии не будут внедрены несинхронизированные модифицирующие методы. Как видите, клиентская блокировка — весьма ненадежный прием, и поэтому он обычно не рекомендуется для применения.

Принцип монитора

Блокировки и условия — эффективные инструментальные средства синхронизации потоков, но они не слишком объектно-ориентированы. В течение многих лет исследователи ищут способы обеспечения безопасности многопоточной обработки, чтобы избавить программистов от необходимости думать о явных блокировках.

Одно из наиболее успешных решений — принцип монитора, который был впервые предложен Пером Бринчем Хансеном (Per Brinch Hansen) и Тони Хоаром (Tony Hoare) в 1970-х годах. В терминологии Java монитор обладает следующими свойствами.

- Монитор — это класс, имеющий только закрытые поля.
- У каждого объекта такого класса имеется связанная с ним блокировка.
- Все методы блокируются этой блокировкой. Иными словами, если клиент вызывает метод `obj.method()`, блокировка объекта `obj` автоматически захватывается в начале этого метода и снимается по его завершении. А поскольку все поля класса монитора закрыты, то такой подход гарантирует, что к ним нельзя будет обратиться ни в одном из потоков до тех пор, пока ими манипулирует какой-то другой поток.
- У блокировки может быть любое количество связанных с ней условий.

Ранние версии мониторов имели единственное условие с довольно изящным синтаксисом. Так, можно было просто сделать вызов `await accounts[from] >= balance`, не указывая явную условную переменную. Но исследования показали, что неразборчивая повторная проверка условий может оказаться неэффективной. Проблема была решена благодаря применению явных условных переменных, каждая из которых управляет отдельным рядом потоков.

Создатели Java вольно адаптировали принцип монитора. *Каждый* объект в Java обладает встроенной блокировкой и встроенным условием. Если метод объявлен с ключевым словом `synchronized`, он действует как метод монитора. Переменная условия доступна через вызовы методов `wait()`, `notifyAll()`, `notify()`.

Но объекты в Java отличаются от мониторов в следующих трех важных отношениях, нарушающих безопасность потоков.

- Поля не обязательно должны быть закрытыми (`private`).
- Методы не обязаны быть синхронизированными (`synchronized`).
- Встроенная блокировка доступна клиентам.

Это — явное пренебрежение требованиями безопасности, изложенными Пером Бринчем Хансеном. В уничижительном обзоре, посвященном примитивам многозадачной обработки в Java, он пишет: “Для меня является непостижимым тот факт, что небезопасный параллелизм столь серьезно принят сообществом программистов, и это спустя четверть века после изобретения мониторов и языка `Concurrent Pascal`. Этому нет оправданий”. (*Java’s Insecure Parallelism, ACM SIGPLAN Notices 34:38–45, April 1999*).

Поля и переменные типа `volatile`

Плата за синхронизацию кажется порой непомерной, когда нужно просто читать или записать данные в одно или два поля экземпляра. В конце концов, что такого страшного может при этом произойти? К сожалению, современные процессоры и компиляторы оставляют широкие возможности для появления ошибок.

- Компьютеры с несколькими процессорами могут временно удерживать значения из памяти в регистрах или локальных кешах. Вследствие этого потокам, выполняющимся на разных процессорах, могут быть доступны разные значения в одной и той же области памяти!
- Компиляторы могут изменять порядок выполнения команд для достижения максимальной производительности. Они не меняют этот порядок таким образом, чтобы изменился смысл кода, но делают предположения, что значения в памяти меняются только явными командами в коде. Но значение в памяти может быть изменено в другом потоке!

Если вы пользуетесь блокировками для защиты кода, который может выполняться в нескольких потоках, то вряд ли столкнетесь с подобными затруднениями. Компиляторы обязаны соблюдать блокировки, очищая при необходимости локальные кеши и не изменяя порядок следования команд. Подробнее об этом можно узнать из документа *Java Memory Model and Thread Specification* (Спецификация модели памяти и потоков в Java), разработанного экспертной группой JSR 133 (www.jcp.org/en/jsr/detail?id=133). Большая часть этого документа довольно сложна и полна технических подробностей, но в нем приведен также целый ряд наглядных примеров. Более доступный обзор данной темы, автором которого является Брайан Гоецц, доступен по адресу <http://www.ibm.com/developerworks/ru/library/j-jtp02244/>.



НА ЗАМЕТКУ! Брайан Гоецц предложил следующий “девич синхронизации”: если вы записываете переменную, которая может быть затем прочитана в другом потоке, или же читаете переменную, которая была записана в другом потоке, то обязаны использовать синхронизацию.

Ключевое слово `volatile` обозначает неблокирующий механизм синхронизированного доступа к полю экземпляра. Если поле объявляется как `volatile`, то компилятор и виртуальная машина принимают во внимание тот факт, что поле может быть параллельно обновлено в другом потоке.

Допустим, у объекта имеется поле признака `done` типа `boolean`, который устанавливается в одном потоке и опрашивается в другом. Как пояснялось выше, для этой цели можно организовать встроенную блокировку следующим образом:

```
private boolean done;
public synchronized boolean isDone() { return done; }
public synchronized void setDone() { done = true; }
```

Возможно, применять встроенную блокировку объекта — не самая лучшая идея. Ведь методы `isDone()` и `setDone()` могут блокироваться, если другой поток заблокировал объект. В таком случае можно воспользоваться отдельным объектом типа `Lock` только для данной переменной. Но это повлечет за собой немало хлопот. Поэтому в данном случае имеет смысл объявить поле как `volatile` следующим образом:

```
private volatile boolean done;
public boolean isDone() { return done; }
public void setDone() { done = true; }
```



ВНИМАНИЕ! Переменные типа `volatile` не гарантируют никакой атомарности операций. Например, приведенный ниже метод не гарантирует смены значения поля на противоположное.

```
public void flipDone() { done = !done; } // не атомарная операция!
```

Поля и переменные типа `final`

Как было показано в предыдущем разделе, благополучно прочитать содержимое поля из нескольких потоков не удастся, если не применить блокировки или модификатор доступа `volatile`. Но имеется еще одна возможность получить надежный доступ к разделяемому полю, если оно объявлено как `final`. Рассмотрим следующую строку кода:

```
final Map<String, Double> accounts = new HashMap<>();
```

Переменная `accounts` станет доступной из других потоков по завершении конструктора. Если не объявить ее как `final`, то нет никакой гарантии, что обновленное значение переменной `accounts` окажется доступным из других потоков. Ведь если конструктор класса `HashMap` не завершится нормально, значение этой переменной может оказаться пустым (`null`). Разумеется, операции над отображением не являются потокобезопасными. Если содержимое отображения видоизменяется или читается в нескольких потоках, то по-прежнему требуется их синхронизация.

Атомарность операций

Разделяемые переменные могут быть объявлены как `volatile`, при условии, что над ними не выполняется никаких операций, кроме присваивания. В пакете `java.util.concurrent.atomic` имеется целый ряд классов, в которых эффективно используются команды машинного уровня, гарантирующие атомарность других операций без применения блокировок. Например, в классе `AtomicInteger` имеются методы `incrementAndGet()` и `decrementAndGet()`, атомарно инкрементирующие или декрементирующие целое значение. Объект этого класса можно благополучно использовать в качестве разделяемого счетчика без дополнительной синхронизации потоков.

В данном пакете имеются также классы `AtomicBoolean`, `AtomicLong`, `AtomicReference` и атомарные массивы логических, целых, длинных целых значений и ссылок. Эти классы предназначены в основном для системных программистов, разрабатывающих утилиты параллелизма, а не для прикладных программистов.

Взаимные блокировки

Блокировки и условия не могут решить всех проблем, которые возникают в многопоточных приложениях. Рассмотрим следующую ситуацию:

Счет 1: сумма \$200

Счет 2: сумма \$300

Поток 1: переводит сумму \$300 со счета 1 на счет 2

Поток 2: переводит сумму \$400 со счета 2 на счет 1

Как следует из рис. 14.6, потоки 1 и 2 явно блокируются. Ни один из них не выполняется, поскольку остатков на счетах 1 и 2 недостаточно для выполнения транзакции. Может так случиться, что все потоки будут заблокированы, поскольку каждый из них будет ожидать пополнения счета. Такая ситуация называется *взаимной блокировкой*.

В рассматриваемой здесь программе взаимная блокировка не может произойти по следующей простой причине: сумма каждого перевода не превышает \$1000. А поскольку имеется всего 100 счетов с общим балансом \$100000, то как минимум на одном счете в любой момент времени должна быть сумма больше \$1000. Поэтому поток, переводящий деньги с этого счета, может продолжить свое выполнение.

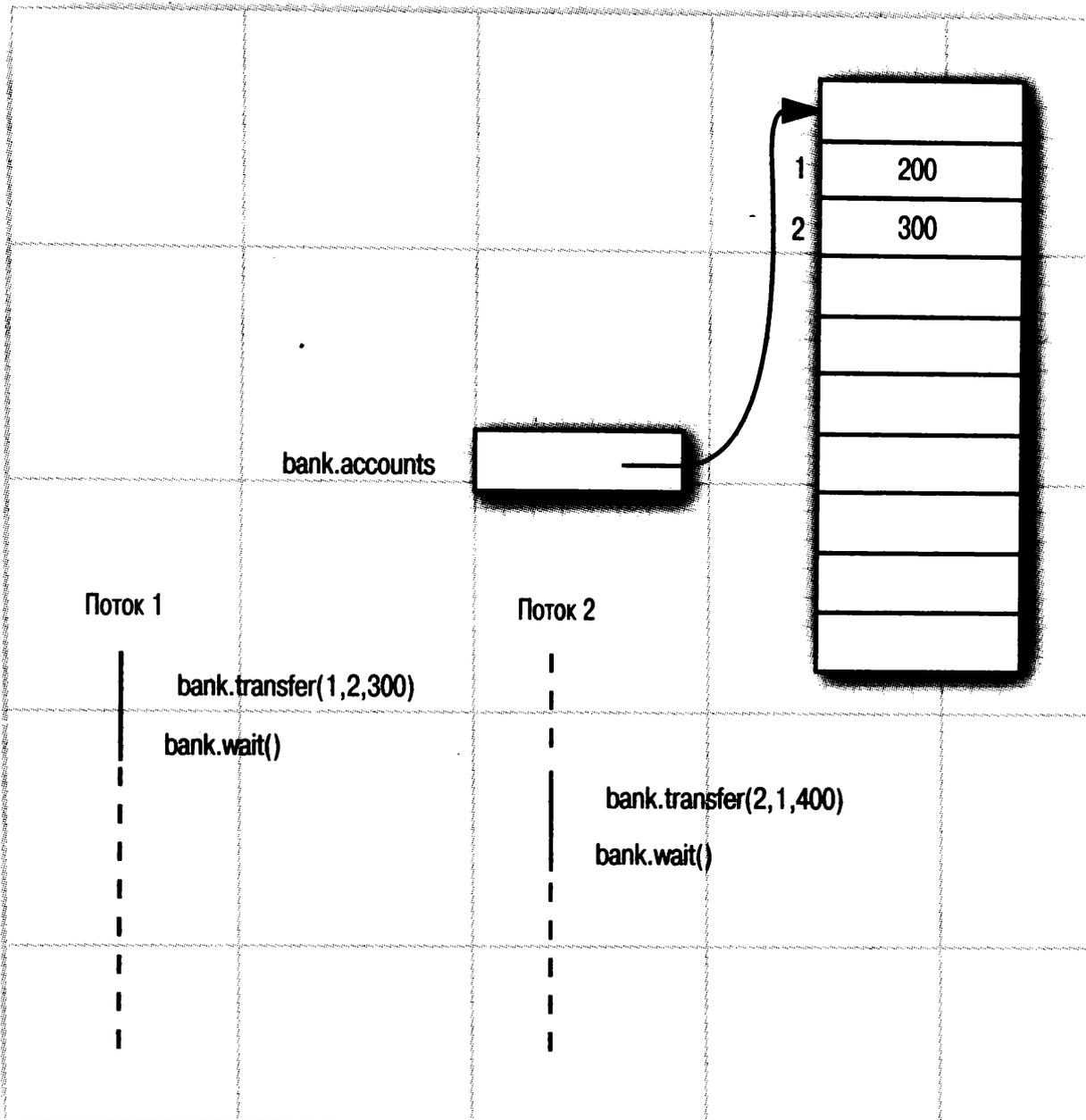


Рис. 14.6. Ситуация взаимной блокировки

Но если внести изменение в метод `run()`, исключив лимит \$1000 на транзакцию, то взаимная блокировка произойдет очень скоро. Можете убедиться в этом сами. Для этого установите значение константы `NACCOUNTS` равным 10, сконструируйте объект типа `Runnable` со значением поля `max`, равным $2 * \text{INITIAL_BALANCE}$, и запустите программу на выполнение. Она будет работать довольно долго, но в конечном итоге зависнет.



СОВЕТ. Когда программа зависнет, нажмите комбинацию клавиш `<Ctrl+\>`. В итоге будет выведено содержимое памяти с перечислением всех лотоков. Для каждого потока выполняется трассировка стека, из которой видно, где именно произошла блокировка. В качестве альтернативы запустите утилиту `jconsole`, как описано в главе 11, и просмотрите панель `Threads (Потоки)`, приведенную на рис. 14.7.

Другой способ создать взаимную блокировку — сделать i -й поток ответственным за размещение денег на i -м счете вместо их снятия с i -го счета. В этом случае

имеется вероятность, что все потоки набросятся на один и тот же счет и каждый из них будет пытаться снять с него деньги. Попробуйте смоделировать подобную ситуацию. Для этого обратитесь в программе `SynchBankTest` к методу `run()` из класса `TransferRunnable`, а в вызове метода `transfer()` поменяйте местами параметры `fromAccount` и `toAccount`. После запуска программы вы почти сразу же обнаружите взаимную блокировку.

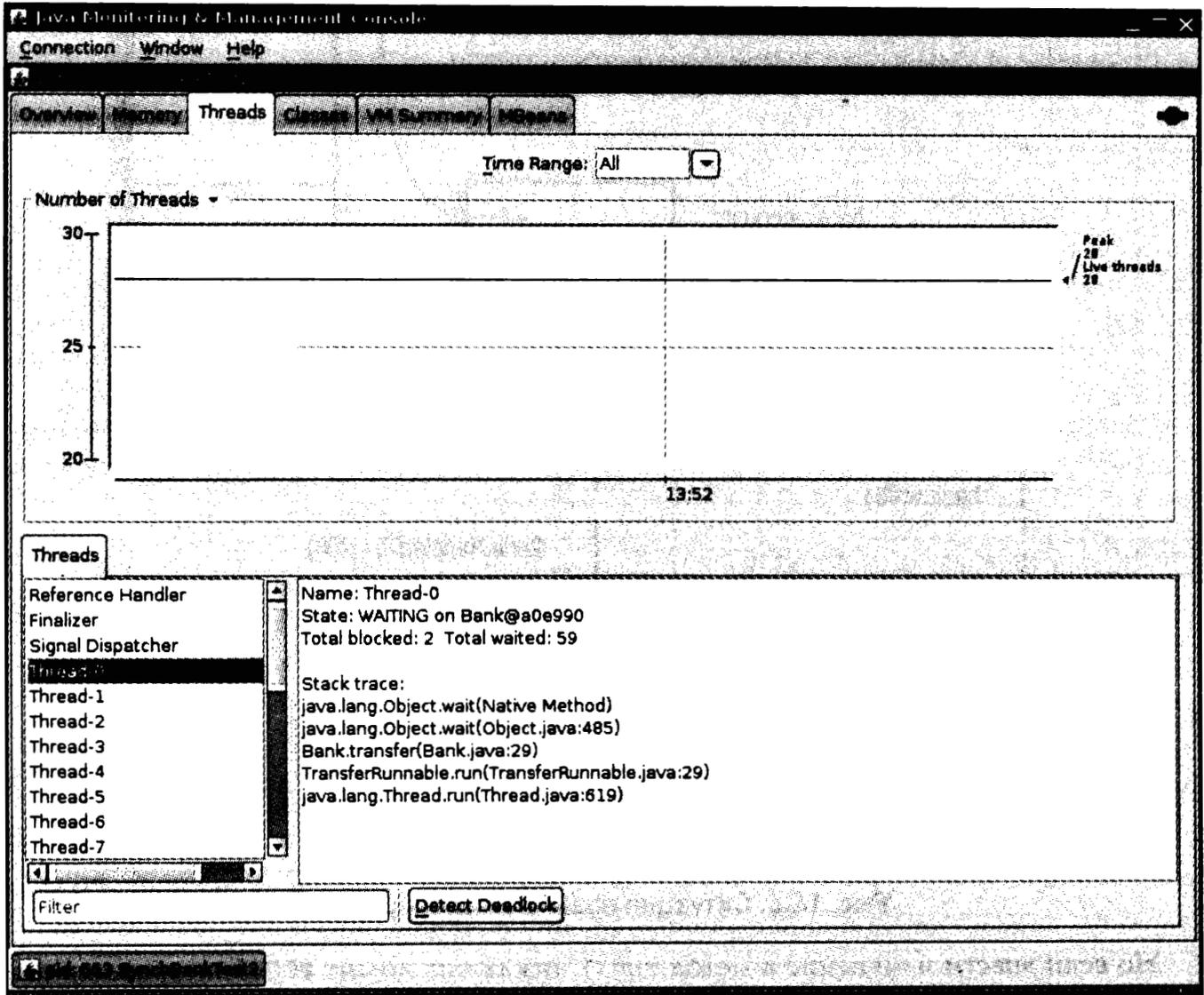


Рис. 14.7. Вид панели Threads в окне утилиты jconsole

Взаимная блокировка может легко возникнуть в том случае, если заменить метод `signalAll()` на `signal()` в программе `SynchBankTest`. Сделав это, вы обнаружите, что программа в конечном итоге зависнет. (И в этом случае лучше установить значение 10 константы `NACCOUNTS`, чтобы как можно скорее добиться желаемого результата.) В отличие от метода `signalAll()`, который извещает все потоки, ожидающие пополнения счета, метод `signal()` разблокирует только один поток. Если этот поток не может продолжить свое выполнение, то все потоки могут оказаться заблокированными. Рассмотрим следующий простой сценарий создания взаимной блокировки:

Счет 1: \$1,990

Все прочие счета: сумма \$990 на каждом

Поток 1: переводит сумму \$995 со счета 1 на счет 2

Все прочие потоки: переводят сумму \$995 со своего счета на другой

Ясно, что все потоки, кроме потока 1, заблокированы, поскольку на их счетах недостаточно денег. Поток 1 выполняет перевод денег, после чего возникает следующая ситуация:

Счет 1: сумма \$995

Счет 2: сумма \$1,985

Все прочие счета: сумма \$990 на каждом

Затем в потоке 1 вызывается метод `signal()`, который произвольным образом выбирает поток для разблокировки. Допустим, он выбирает поток 3. Этот поток активизируется, обнаруживает, что на его счете недостаточно денег, и снова вызывает метод `await()`. Но поток 1 все еще выполняется. В итоге формируется новая произвольная транзакция вроде следующей:

Поток 1: переводит сумму \$997 со счета 1 на счет 2

Теперь метод `await()` вызывается и в потоке 1, а в итоге все потоки блокируются. Система входит во взаимную блокировку. И виновником всему оказывается вызов метода `signal()`. Он разблокирует только один поток, а таким потоком может оказаться совсем не тот, который позволит программе выполняться дальше. (В данном случае поток 2 должен выполнить перевод денег со счета 2.)

К сожалению, в Java отсутствуют средства, исключающие или снимающие взаимные блокировки. Поэтому вам придется разрабатывать свои программы, чтобы полностью исключить ситуации взаимных блокировок.

Локальные переменные потоков

В предыдущих разделах обсуждались риски совместного использования переменных, разделяемых между потоками. Иногда можно избежать такого разделения общих ресурсов, предоставляя каждому потоку свой экземпляр с помощью вспомогательного класса `ThreadLocal`. Например, класс `SimpleDateFormat` не является потокобезопасным. Допустим, имеется следующая статическая переменная:

```
public static final SimpleDateFormat dateFormat =
    new SimpleDateFormat("yyyy-MM-dd");
```

Если в двух потоках выполняется приведенная ниже операция, то ее результат может превратиться в "мусор", поскольку внутренние структуры данных, используемые переменной `dateFormat`, могут быть повреждены в параллельно выполняющемся потоке.

```
String dateStamp = dateFormat.format(new Date());
```

Во избежание этого можно было бы, с одной стороны, организовать синхронизацию потоков, но это недешевое удовольствие, а с другой стороны, сконструировать локальный объект типа `SimpleDateFormat` по мере надобности в нем, но и это расточительство. Поэтому для построения одного экземпляра на каждый поток лучше воспользоваться следующим фрагментом кода:

```
public static final ThreadLocal<SimpleDateFormat> dateFormat =
    new ThreadLocal<SimpleDateFormat>()
    {
        protected SimpleDateFormat initialValue()
        {
            return new SimpleDateFormat("yyyy-MM-dd");
        }
    };
```

Для доступа к конкретному средству форматирования делается следующий вызов:

```
String dateStamp = dateFormat.get().format(new Date());
```

При первом вызове метода `get()` в данном потоке вызывается также метод `initialValue()`. А по его завершении метод `get()` возвращает экземпляр, принадлежащий текущему потоку.

Аналогичные трудности вызывает генерирование случайных чисел в нескольких потоках. Для этой цели служит класс `java.util.Random`, который является потокобезопасным. Но и он оказывается недостаточно эффективным, если нескольким потокам приходится ожидать доступа к единственному разделяемому между ними генератору случайных чисел.

Для предоставления каждому потоку отдельного генератора случайных чисел можно было бы воспользоваться вспомогательным классом `ThreadLocal`, но в версии Java SE 7 для этой цели предоставляется служебный класс `ThreadLocalRandom`. Достаточно сделать приведенный ниже вызов `ThreadLocalRandom.current()`, и в результате возвратится экземпляр класса `Random`, однозначный для текущего потока.

```
int random = ThreadLocalRandom.current().nextInt(upperBound);
```

`java.lang.ThreadLocal<T>` 1.2

- `T get()`
Получает текущее значение в данном потоке. Если этот метод вызывается в первый раз, то значение получается путем вызова метода `initialize()`.
- `protected initialize()`
Этот метод следует переопределить, чтобы он предоставил исходное значение. По умолчанию этот метод возвращает пустое значение `null`.
- `void set(T t)`
Устанавливает новое значение для данного потока.
- `void remove()`
Удаляет значение из потока.

`java.util.concurrent.ThreadLocalRandom` 7

- `static ThreadLocalRandom current()`
Возвращает экземпляр класса `Random`, однозначный для текущего потока.

Проверка блокировок и время ожидания

Поток блокируется на неопределенное время, когда он вызывает метод `lock()` для захвата блокировки, которой владеет другой поток. Поэтому следует проявлять осторожность, пытаясь захватить блокировку. Метод `tryLock()` пытается захватить блокировку и возвращает логическое значение `true`, если ему удастся это сделать. В противном случае он немедленно возвращает логическое значение `false`, и тогда

поток может переключиться на выполнение каких-нибудь других полезных действий, как показано ниже.

```
if (myLock.tryLock())
    // теперь поток владеет блокировкой
    try { . . . }
    finally { myLock.unlock(); }
else
    // сделать что-нибудь еще полезное
```

Метод `tryLock()` можно вызвать, указав время ожидания в качестве параметра, как показано в приведенной ниже строке кода, где `TimeUnit` — значение перечисления из ряда `SECONDS`, `MILLISECONDS`, `MICROSECONDS` и `NANOSECONDS`.

```
if (myLock.tryLock(100, TimeUnit.MILLISECONDS)) . . .
```

Метод `lock()` не может быть прерван. Если поток прерван во время ожидания захвата блокировки, прерванный поток остается заблокированным до тех пор, пока доступна блокировка. Если же возникнет взаимная блокировка, то метод `lock()` вообще не завершится.

Но если вызвать метод `tryLock()` с указанием времени ожидания в качестве параметра, то при попытке прерывания потока, находящегося в состоянии ожидания, будет сгенерировано исключение типа `InterruptedException`. Это, безусловно, полезное средство, поскольку оно дает возможность снимать взаимные блокировки в программе.

Кроме того, можно вызвать метод `lockInterruptibly()`. У него то же самое назначение, что и у метода `tryLock()`, но только с бесконечным временем ожидания.

Когда же ожидается выполнение некоторого условия, то и в этом случае можно указать время ожидания, как показано ниже. Метод `await()` возвращает управление, если другой поток активизирует данный поток вызовом метода `signalAll()` или `signal()`, по истечении времени ожидания или в связи с тем, что поток был прерван.

```
myCondition.await(100, TimeUnit.MILLISECONDS)
```

Метод `await()` генерирует исключение типа `InterruptedException`, если ожидающий поток прерывается. В том (скорее, маловероятном) случае, если требуется продолжить ожидание, вместо этого можно вызвать метод `awaitUninterruptibly()`.

java.util.concurrent.locks.Lock 5.0

- `boolean tryLock()`
Пытается захватить блокировку без блокирования. Возвращает логическое значение `true` при удачном исходе. Этот метод захватывает блокировку, если она доступна, даже при условии, что соблюдается правило равноправного блокирования и ожидания в других потоках.
- `boolean tryLock(long time, TimeUnit unit)`
Пытается захватить блокировку, но блокируя не дольше заданного времени. Возвращает логическое значение `true` при удачном исходе.
- `void lockInterruptibly()`
Захватывает блокировку, но блокируя на неопределенное время. Если поток прерывается, генерирует исключение типа `InterruptedException`.

java.util.concurrent.locks.Condition 5.0

- `boolean await(long time, TimeUnit unit)`

Входит в набор ожидания по данному условию, блокируя до тех пор, пока поток не будет удален из набора ожидания или же до тех пор, пока не истечет заданное время. Возвращает логическое значение `false`, если завершается по истечении времени ожидания, а иначе — логическое значение `true`.

- `void awaitUninterruptibly()`

Входит в набор ожидания по данному условию, блокируя до тех пор, пока поток не будет удален из набора ожидания. Если поток прерван, этот метод не генерирует исключение типа `InterruptedException`.

Блокировки чтения/записи

В пакете `java.util.concurrent.locks` определяются два класса блокировок: уже рассмотренный ранее класс `ReentrantLock`, а также класс `ReentrantReadWriteLock`. Последний удобен в тех случаях, когда имеется больше потоков для чтения из структуры данных и поменьше потоков для записи в нее. В подобных случаях имеет смысл разрешить совместный доступ читающим потокам. Разумеется, записывающий поток должен по-прежнему иметь исключительный доступ.

Ниже описаны действия, которые следует предпринять для организации блокировок чтения/записи.

1. Сконструируйте объект типа `ReentrantReadWriteLock`:

```
private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
```

2. Извлеките блокировки чтения и записи:

```
private Lock readLock = rwl.readLock();
private Lock writeLock = rwl.writeLock();
```

3. Используйте блокировку чтения для всех потоков, получающих доступ только для чтения данных:

```
public double getTotalBalance()
{
    readLock.lock();
    try { . . . }
    finally { readLock.unlock(); }
}
```

4. Используйте блокировку записи для всех потоков, получающих доступ для видоизменения данных:

```
public void transfer(. . .)
{
    writeLock.lock();
    try { . . . }
    finally { writeLock.unlock(); }
}
```

java.util.concurrent.locks.ReentrantReadWriteLock 5.0

- Lock readLock()
Получает блокировку чтения, которая может быть захвачена многими читающими потоками, но исключая все записывающие потоки.
- Lock writeLock()
Получает блокировку записи, исключаящую доступ для всех читающих и остальных записывающих потоков.

Причины, по которым методы `stop()` и `suspend()` не рекомендованы к применению

В первоначальной версии Java был определен метод `stop()`, который просто останавливал поток, а также метод `suspend()`, который блокировал поток до тех пор, пока другой поток не вызывал метод `resume()`. У методов `stop()` и `suspend()` имеется нечто общее: оба пытаются контролировать поведение данного потока без учета взаимодействия потоков.

Оба эти метода были объявлены как не рекомендованные к применению, начиная с версии Java SE 1.2. Метод `stop()`, по существу, небезопасен, а что касается метода `suspend()`, то, как показывает опыт, он часто приводит к взаимным блокировкам. В этом разделе поясняется, почему применение этих методов проблематично и что нужно делать, чтобы избежать проблем, которые они вызывают.

Обратимся сначала к методу `stop()`. Он прекращает выполнение любых незавершенных методов, включая и `run()`. Когда поток останавливается, данный метод немедленно снимает блокировки со всех объектов, которые он блокировал. Это может привести к тому, что объекты останутся в несогласованном состоянии. Допустим, метод `TransferThread()` остановлен посередине перевода денег с одного счета на другой: после снятия денег с одного счета, но перед переносом их на другой. В этом случае объект имитируемого банка оказывается *поврежденным*. А поскольку блокировка снята, этот поврежденный объект будет доступен и другим потокам, которые не были остановлены.

Когда одному потоку требуется остановить другой поток, он никоим образом не может знать, когда вызов метода `stop()` безопасен, а когда он может привести к повреждению объектов. Поэтому данный метод был объявлен не рекомендованным к применению. Когда требуется остановить поток, его нужно прервать. Прерванный поток может затем остановиться сам, когда это можно будет сделать безопасно.



НА ЗАМЕТКУ! Некоторые авторы требовали объявления метода `stop()` нежелательным, поскольку он мог приводить к появлению объектов, заблокированных остановленным потоком навсегда. Но подобное требование некорректно. Остановленный поток выходит из всех синхронизированных методов, которые он вызывал, генерируя исключение типа `ThreadDeath`. Как следствие, поток освобождает все встроенные блокировки объектов, которые он удерживает.

А теперь рассмотрим, что же не так делает метод `suspend()`. В отличие от метода `stop()`, метод `suspend()` не повреждает объекты. Но если приостановить поток, владеющий блокировкой, то эта блокировка останется недоступной до тех пор, пока

поток не будет возобновлен. Если поток, вызвавший метод `suspend()`, попытается захватить ту же самую блокировку, программа перейдет в состояние взаимной блокировки: приостановленный поток ожидает, когда его возобновят, а поток, который приостановил его, ожидает снятия блокировки.

Подобная ситуация часто возникает в ГПИ. Допустим, имеется графический имитатор банка. Кнопка `Pause` приостанавливает потоки денежных переводов, а кнопка `Resume` возобновляет их, как показано ниже.

```
pauseButton.addActionListener(new
    ActionListener()
    .{
        public void actionPerformed(ActionEvent event)
        {
            for (int i = 0; i < threads.length; i++)
                threads[i].suspend(); // не рекомендуется делать!
        }
    });
resumeButton.addActionListener(. . .); // возобновляет все потоки переводов
```

Допустим также, что метод `paintComponent()` рисует график каждого счета, вызывая метод `getBalances()` для получения массива остатков на счетах. Как поясняется далее в разделе “Потоки и библиотека `Swing`”, действия обеих кнопок и перерисовка происходят в одном и том же потоке, называемом *потоком диспетчеризации событий*. Рассмотрим следующий сценарий.

1. Один из потоков, выполняющих денежные переводы, захватывает блокировку объекта `bank`.
2. Пользователь щелкает на кнопке `Pause`.
3. Все потоки денежных переводов приостанавливаются, но один из них продолжает удерживать блокировку для объекта `bank`.
4. По той же причине график счета должен быть перерисован.
5. Метод `paintComponent()` вызывает метод `getBalance()`.
6. Этот метод пытается захватить блокировку объекта `bank`.

В итоге программа зависает. Поток диспетчеризации событий не может продолжить свое выполнение, поскольку блокировкой владеет один из приостановленных потоков. Поэтому пользователь не может активизировать кнопку `Resume`, чтобы возобновить выполнение потоков.

Если требуется безопасно приостановить поток, следует ввести переменную `suspendRequested` и проверить ее в безопасном месте метода `run()`, т.е. там, где данный поток не блокирует объекты, необходимые другим потокам. Когда в данном потоке обнаружится, что переменная `suspendRequested` установлена, ему придется подождать до тех пор, пока она станет вновь доступной. Подобный прием демонстрируется в следующем фрагменте кода:

```
private volatile boolean suspendRequested = false;
private Lock suspendLock = new ReentrantLock();
private Condition suspendCondition = suspendLock.newCondition();

public void run()
{
    while (. . .)
```

```
{
    . . .
    if (suspendRequested)
    {
        suspendLock.lock();
        try { while (suspendRequested) suspendCondition.await(); }
        finally { suspendLock.unlock(); }
    }
}

public void requestSuspend() { suspendRequested = true; }

public void requestResume()
{
    suspendRequested = false;
    suspendLock.lock();
    try { suspendCondition.signalAll(); }
    finally { suspendLock.unlock(); }
}
```

Блокирующие очереди

Теперь вы знакомы со всеми основными низкоуровневыми составляющими параллельного программирования на Java. Но в практике программирования вы, скорее всего, предпочтете держаться как можно дальше от низкоуровневых конструкций. Ведь намного проще и безопаснее работать со структурами более высокого уровня, реализованными знатоками параллелизма.

Многие затруднения, связанные с потоками, можно изящно и безопасно сформулировать, применив одну или больше очередей. В частности, поставляющий поток вводит элементы в очередь, а потребляющие потоки извлекают их. Таким образом, очередь позволяет безопасно передавать данные из одного потока в другой. Обратимся снова к примеру программы банковских переводов. Вместо того чтобы обращаться к объекту банка напрямую, потоки, выполняющие денежные переводы, вводят объекты команд на денежный перевод в очередь. А другой поток удаляет эти объекты из очереди и сам выполняет денежные переводы. И только этот поток имеет доступ к внутреннему содержимому объекта банка. В итоге никакой синхронизации не требуется. (Конечно, разработчики потокобезопасных классов очередей должны позаботиться о блокировках и условиях, но это их забота, а не ваша как пользователя таких классов.)

Блокирующая очередь вынуждает поток блокироваться при попытке ввести элемент в переполненную очередь или удалить элемент из пустой очереди. Блокирующие очереди — удобное инструментальное средство для координации работы множества потоков. Одни рабочие потоки могут периодически размещать промежуточные результаты в блокирующей очереди, а другие рабочие потоки — удалять промежуточные результаты и видоизменять их далее.

Такая очередь автоматически уравнивает нагрузку. Если первая группа потоков работает медленнее, чем вторая, то вторая блокируется, ожидая результатов. Если же первая группа работает быстрее, она притормаживается до тех пор, пока вторая не догонит ее. В табл. 14.1 перечислены методы блокирующих очередей.

Таблица 14.1. Методы блокирующих очередей

Метод	Обычное действие	Действие в особых случаях
<code>add()</code>	Вводит элемент в очередь	Генерирует исключение типа <code>IllegalStateException</code> , если очередь заполнена
<code>element()</code>	Возвращает элемент, находящийся в голове очереди	Генерирует исключение типа <code>NoSuchElementException</code> , если очередь пуста
<code>offer()</code>	Вводит элемент в очередь и возвращает логическое значение <code>true</code>	Возвращает логическое значение <code>false</code> , если очередь заполнена
<code>peek()</code>	Возвращает элемент, находящийся в голове очереди	Возвращает пустое значение <code>null</code> , если очередь пуста
<code>poll()</code>	Возвращает элемент, находящийся в голове очереди, удаляя его из очереди	Возвращает пустое значение <code>null</code> , если очередь пуста
<code>put()</code>	Вводит элемент в очередь	Блокирует, если очередь пуста
<code>remove()</code>	Возвращает элемент, находящийся в голове очереди, удаляя его из очереди	Генерирует исключение типа <code>NoSuchElementException</code> , если очередь пуста
<code>take()</code>	Возвращает элемент, находящийся в голове очереди, удаляя его из очереди	Блокирует, если очередь пуста

Методы блокирующих очередей разделяются на три категории, в зависимости от выполняемых действий, когда очередь заполнена или пуста. Для применения блокирующей очереди в качестве инструментального средства управления потоками понадобятся методы `put()` и `take()`. Методы `add()`, `remove()` и `element()` генерируют исключение при попытке ввести элемент в заполненную очередь или получить элемент из головы пустой очереди. Разумеется, в многопоточной программе очередь может заполниться или опустеть в любой момент, поэтому вместо этих методов, возможно, потребуются методы `offer()`, `poll()` и `peek()`. Эти методы просто возвращают признак сбоя вместо исключения, если они не могут выполнить свои функции.



НА ЗАМЕТКУ! Методы `poll()` и `peek()` возвращают пустое значение `null` для обозначения неудачного исхода. Поэтому пустые значения `null` недопустимо вводить в блокирующие очереди.

Существуют варианты методов `offer()` и `poll()` с указанием времени ожидания. Например, при приведенном ниже вызове в течение 100 миллисекунд предпринимается попытка ввести элемент в хвост очереди. Если это удастся сделать, то возвращается логическое значение `true`, в противном случае — логическое значение `false` по истечении времени ожидания.

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

Аналогично при приведенном ниже вызове в течение 100 миллисекунд предпринимается попытка удалить элемент из головы очереди. Если это удастся сделать, то возвращается логическое значение `true`, в противном случае — логическое значение `false` по истечении времени ожидания.

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

Метод `put ()` блокирует, когда очередь заполнена. А метод `take ()` блокирует, когда очередь пуста. Существуют также эквиваленты методов `offer ()` и `put ()` без указания времени ожидания.

В пакете `java.util.concurrent` предоставляется несколько вариантов блокирующих очередей. По умолчанию у очереди типа `LinkedBlockingQueue` отсутствует верхняя граница емкости, но такая граница емкости может быть указана. `LinkedBlockingDeque` — это вариант блокирующей двухсторонней очереди. А блокирующая очередь типа `ArrayBlockingQueue` конструируется с заданной емкостью и признаком равноправия блокировки в качестве необязательного параметра. Если этот параметр указан, то предпочтение отдается очередям, дольше всего находящимся в состоянии ожидания. Как всегда, от этого существенно страдает производительность. Потому правило равноправия блокировки следует применять только в том случае, если оно действительно разрешает затруднения, возникающие при многопоточной обработке.

`PriorityBlockingQueue` — это блокирующая очередь по приоритету, а не просто действующая по принципу “первым пришел — первым обслужен”. Элементы удаляются из такой очереди по их приоритетам. Эта очередь имеет неограниченную емкость, но при попытке извлечь элемент из пустой очереди происходит блокирование. (Подробнее об очередях по приоритету см. в главе 13.)

И, наконец, блокирующая очередь типа `DelayQueue` содержит объекты, реализующие интерфейс `Delayed`, объявляемый следующим образом:

```
interface Delayed extends Comparable<Delayed>
{
    long getDelay(TimeUnit unit);
}
```

Метод `getDelay ()` возвращает оставшееся время задержки объекта. Отрицательное значение указывает, что время задержки истекло. Элементы могут быть удалены из очереди типа `DelayQueue` только в том случае, если истечет время их задержки. Кроме того, придется реализовать метод `compareTo ()`, который используется в очереди типа `DelayQueue` для сортировки ее элементов.

В версии Java SE 7 внедрен интерфейс `TransferQueue`, дающий поставляющему потоку возможность ожидать до тех пор, пока потребляющий поток не будет готов принять элемент из очереди. Когда в поставляющем потоке делается приведенный ниже вызов, блокировка устанавливается до тех пор, пока она не будет снята в другом потоке. Данный интерфейс реализуется в классе `LinkedTransferQueue`.

```
q.transfer(item);
```

В примере программы, приведенном в листинге 14.10, демонстрируется применение блокирующей очереди для управления многими потоками. Эта программа осуществляет поиск среди всех файлов в каталоге и его подкаталогах, выводя строки кода, содержащие заданное ключевое слово.

В поставляющем потоке перечисляются все файлы во всех подкаталогах, а затем они размещаются в блокирующей очереди. Эта операция выполняется быстро, и поэтому очередь быстро заполняется всеми файлами из файловой системы, если не установлена верхняя граница ее емкости.

Кроме того, запускается огромное количество поисковых потоков. В каждом таком потоке файл извлекается из очереди, открывается, а затем из него выводятся все строки, содержащие ключевое слово, после чего из очереди извлекается следующий файл. Для того чтобы прекратить выполнение данной программы, когда никакой

другой обработки файлов больше не требуется, применяется специальный прием: из перечисляющего потока в очередь вводится фиктивный объект, чтобы уведомить о завершении потока. (Это похоже на муляж чемодана с надписью “последний чемодан” на ленточном транспортере в зале выдачи багажа.) Когда поисковый поток получает такой объект, он возвращает его обратно и завершается.

Следует также заметить, что в данном примере программы не требуется никакой явной синхронизации потоков. А в качестве синхронизирующего механизма используется сама структура очереди.

Листинг 14.10. Исходный код из файла `blockingQueue/BlockingQueueTest.java`

```
1 package blockingQueue;
2
3 import java.io.*;
4 import java.util.*;
5 import java.util.concurrent.*;
6
7 /**
8  * @version 1.01 2012-01-26
9  * @author Cay Horstmann
10 */
11 public class BlockingQueueTest
12 {
13     public static void main(String[] args)
14     {
15         Scanner in = new Scanner(System.in);
16         System.out.print(
17             "Enter base directory (e.g. /usr/local/jdk1.6.0/src): ");
18         String directory = in.nextLine();
19         System.out.print("Enter keyword (e.g. volatile): ");
20         String keyword = in.nextLine();
21
22         final int FILE_QUEUE_SIZE = 10;
23         final int SEARCH_THREADS = 100;
24
25         BlockingQueue<File> queue =
26             new ArrayBlockingQueue<>(FILE_QUEUE_SIZE);
27
28         FileEnumerationTask enumerator =
29             new FileEnumerationTask(queue, new File(directory));
30         new Thread(enumerator).start();
31         for (int i = 1; i <= SEARCH_THREADS; i++)
32             new Thread(new SearchTask(queue, keyword)).start();
33     }
34 }
35
36 /**
37  * Перечисляет все файлы в каталоге и его подкаталогах
38 */
39 class FileEnumerationTask implements Runnable
40 {
41     public static File DUMMY = new File("");
42     private BlockingQueue<File> queue;
43     private File startingDirectory;
44
45     /**
46      * Конструирует объект типа FileEnumerationTask
```

```
47  * @param queue Блокирующая очередь, в которую вводятся
48  *             перечисляемые файлы
49  * @param startingDirectory Каталог, с которого начинается
50  *             перечисление файлов
51  */
52  public FileEnumerationTask(
53      BlockingQueue<File> queue, File startingDirectory)
54  {
55      this.queue = queue;
56      this.startingDirectory = startingDirectory;
57  }
58
59  public void run()
60  {
61      try
62      {
63          enumerate(startingDirectory);
64          queue.put(DUMMY);
65      }
66      catch (InterruptedException e)
67      {
68      }
69  }
70
71  /**
72   * Рекурсивно перечисляет все файлы в
73   * данном каталоге и его подкаталогах
74   * @param directory Исходный каталог
75   */
76  public void enumerate(File directory) throws InterruptedException
77  {
78      File[] files = directory.listFiles();
79      for (File file : files)
80      {
81          if (file.isDirectory()) enumerate(file);
82          else queue.put(file);
83      }
84  }
85  }
86  }
87
88  /**
89   * Осуществляет поиск заданного ключевого слова в файлах
90   */
91  class SearchTask implements Runnable
92  {
93      private BlockingQueue<File> queue;
94      private String keyword;
95
96      /**
97       * Конструирует объект типа SearchTask
98       * @param queue Очередь, из которой извлекаются файлы
99       * @param keyword Искомое ключевое слово
100      */
101      public SearchTask(BlockingQueue<File> queue, String keyword)
102      {
103          this.queue = queue;
104          this.keyword = keyword;
105      }
106  }
```

```
107 public void run()
108 {
109     try
110     {
111         boolean done = false;
112         while (!done)
113         {
114             File file = queue.take();
115             if (file == FileEnumerationTask.DUMMY)
116             {
117                 queue.put(file);
118                 done = true;
119             }
120             else search(file);
121         }
122     }
123     catch (IOException e)
124     {
125         e.printStackTrace();
126     }
127     catch (InterruptedException e)
128     {
129     }
130 }
131
132 /**
133  * Осуществляет поиск в файлах заданного ключевого
134  * слова и выводит все совпадающие строки
135  * @param file Файл для поиска ключевого слова
136  */
137
138 public void search(File file) throws IOException
139 {
140     try (Scanner in = new Scanner(file))
141     {
142         int lineNumber = 0;
143         while (in.hasNextLine())
144         {
145             lineNumber++;
146             String line = in.nextLine();
147             if (line.contains(keyword))
148                 System.out.printf(
149                     "%s:%d:%s\n", file.getPath(), lineNumber, line);
150         }
151     }
152 }
153 }
```

java.util.concurrent.ArrayBlockingQueue<E> 5.0

- `ArrayBlockingQueue(int capacity)`
- `ArrayBlockingQueue(int capacity, boolean fair)`

Конструируют блокирующую очередь заданной емкости с установленным правилом равноправия блокировки, реализованную в виде циклического массива.

java.util.concurrent.LinkedBlockingQueue<E> 5.0

java.util.concurrent.LinkedBlockingDeque<E> 6

- `LinkedBlockingQueue()`

- `LinkedBlockingDeque()`

Конструируют неограниченную блокирующую одностороннюю или двустороннюю очередь, реализованную в виде связанного списка.

- `LinkedBlockingQueue(int capacity)`

- `LinkedBlockingDeque(int capacity)`

Конструируют ограниченную блокирующую одно- или двустороннюю очередь, реализованную в виде связанного списка.

java.util.concurrent.DelayQueue<E extends Delayed> 5.0

- `DelayQueue()`

Конструирует неограниченную блокирующую очередь элементов типа `Delayed`. Из очереди могут быть удалены только элементы, время задержки которых истекло.

java.util.concurrent.Delayed 5.0

- `long getDelay(TimeUnit unit)`

Получает задержку для данного объекта, измеряемую в заданных единицах времени.

java.util.concurrent.PriorityBlockingQueue<E> 5.0

- `PriorityBlockingQueue()`

- `PriorityBlockingQueue(int initialCapacity)`

- `PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)`

Конструируют неограниченную блокирующую очередь по приоритетам, реализованную в виде "кучи".

Параметры: `initialCapacity` Исходная емкость очереди по приоритетам.

По умолчанию равна **11**

`comparator`

Компаратор, используемый для сравнения элементов очереди. Если он указан, то элементы очереди должны реализовывать интерфейс `Comparable`

java.util.concurrent.BlockingQueue<E> 5.0

- `void put(E element)`
Вводит элемент в очередь, устанавливая, если требуется, блокировку.
- `E take()`
Удаляет элемент из головы очереди и возвращает его, устанавливая, если требуется, блокировку.
- `boolean offer(E element, long time, TimeUnit unit)`
Вводит заданный элемент в очередь и возвращает логическое значение `true` при удачном исходе, устанавливая, если требуется, блокировку на время ввода элемента в очередь или до истечения времени ожидания.
- `E poll(long time, TimeUnit unit)`
Удаляет элемент из головы очереди и возвращает его, устанавливая, если требуется, блокировку до тех пор, пока элемент доступен, или же до тех пор, пока не истечет время ожидания. При неудачном исходе возвращает пустое значение `null`.

java.util.concurrent.BlockingDeque<E> 6

- `void putFirst(E element)`
- `void putLast(E element)`
Вводят элемент в очередь, устанавливая, если требуется, блокировку.
- `E takeFirst()`
- `E takeLast()`
Удаляют элемент из головы очереди и возвращают его, устанавливая, если требуется, блокировку.
- `boolean offerFirst(E element, long time, TimeUnit unit)`
- `boolean offerLast(E element, long time, TimeUnit unit)`
Вводят заданный элемент в очередь и возвращают логическое значение `true` при удачном исходе, устанавливая, если требуется, блокировку на время ввода элемента или до истечения времени ожидания.
- `E pollFirst(long time, TimeUnit unit)`
- `E pollLast(long time, TimeUnit unit)`
Удаляют и возвращают элемент из головы очереди, устанавливая, если требуется, блокировку до тех пор, пока элемент доступен, или же до тех пор, пока не истечет время ожидания. При неудачном исходе возвращают пустое значение `null`.

java.util.concurrent.TransferQueue<E> 7

- `void transfer(E element)`
- `boolean tryTransfer(E element, long time, TimeUnit unit)`
Передают значение или пытаются передать его в течение заданного времени ожидания, устанавливая, если требуется, блокировку до тех пор, пока элемент не будет удален из очереди в другом потоке. Второй метод возвращает логическое значение `true` при удачном исходе.

Потокобезопасные коллекции

Если во многих потоках одновременно вносятся изменения в структуру данных вроде хеш-таблицы, то эту структуру данных очень легко повредить. (Подробнее о хеш-таблицах см. в главе 13.) Например, в одном потоке может быть начат ввод нового элемента. Допустим, этот поток приостанавливается в процессе переназначения ссылок между группами хеш-таблицы. Если другой поток начнет проход по тому же самому списку, он может последовать по неправильным ссылкам, внося полный беспорядок, а возможно, сгенерировав попутно исключение или войдя в бесконечный цикл.

Структуру данных, разделяемую между потоками, можно защитить, установив блокировку, но обычно проще выбрать потокобезопасную реализацию такой структуры данных. Блокирующие очереди, о которых шла речь в предыдущем разделе, безусловно, являются потокобезопасными коллекциями. В последующих разделах будут рассмотрены другие потокобезопасные коллекции, предоставляемые в библиотеке Java.

Эффективные отображения, множества и очереди

В пакете `java.util.concurrent` предоставляются следующие эффективные реализации отображений, отсортированных множеств и очередей: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet` и `ConcurrentLinkedQueue`. В этих коллекциях используются изолированные алгоритмы, сводящие к минимуму вероятность соперничества, обеспечивая параллельный доступ к разным частям структуры данных.

В отличие от большинства коллекций, метод `size()` не обязательно выполняется в течение постоянного промежутка времени. Для определения размера одной из перечисленных выше коллекций обычно требуется обратиться ко всем ее элементам. Коллекции возвращают *слабо совместные* итераторы. Это означает, что итератор может отражать все изменения, внесенные в коллекцию после его создания, а может и не отражать их. Но такие итераторы никогда не возвращают одно и то же значение дважды и не генерируют исключение типа `ConcurrentModificationException`.



НА ЗАМЕТКУ! Напротив, итератор коллекции из пакета `java.util` генерирует исключение типа `ConcurrentModificationException`, если в коллекцию внесены изменения после создания этого итератора.

Хеш-отображение параллельного действия способно эффективно поддерживать большое количество читающих потоков и фиксированное количество записывающих потоков. По умолчанию допускается до 16 параллельно действующих записывающих потоков. Таких потоков может быть намного больше, но если запись одновременно выполняется в более чем 16 потоках, то остальные временно блокируются. В конструкторе такой коллекции можно, конечно, указать большее количество потоков, но вряд ли это вообще понадобится.

В состав классов `ConcurrentHashMap` и `ConcurrentSkipListMap` входят полезные методы для атомарных операций ввода и удаления взаимосвязей ключей и значений, называемых иначе ассоциациями. В частности, метод `putIfAbsent()` атомарно вводит новую ассоциацию, которой не было раньше. Это удобно для организации кеша, к которому происходит обращение во многих потоках, чтобы гарантировать,

что только один из них может вводить элемент в кеш в любой момент времени, как показано ниже.

```
cache.putIfAbsent(key, value);
```

Обратную атомарную операцию выполняет метод `remove()`, который, в свою очередь, должен вызывать метод `removeIfPresent()`. Так, при следующем вызове автоматически удаляются ключ и значение, если они присутствуют в отображении:

```
cache.remove(key, value)
```

И, наконец, в результате следующего вызова старое значение автоматически заменяется новым, находя его по ключу:

```
cache.replace(key, oldValue, newValue)
```

java.util.concurrent.ConcurrentLinkedQueue<E> 5.0

- `ConcurrentLinkedQueue<E>()`
Конструирует неограниченную и неблокирующую очередь, безопасно доступную из многих потоков.

java.util.concurrent.ConcurrentSkipListSet<E> 6

- `ConcurrentSkipListSet<E>()`
- `ConcurrentSkipListSet<E>(Comparator<? super E> comp)`
Конструируют отсортированное множество, безопасно доступное из многих потоков. Первый конструктор требует, чтобы элементы множества относились к классу, реализующему интерфейс `Comparable`.

java.util.concurrent.ConcurrentHashMap<K, V> 5.0

java.util.concurrent.ConcurrentSkipListMap<K, V> 6

- `ConcurrentHashMap<K, V>()`
- `ConcurrentHashMap<K, V>(int initialCapacity)`
- `ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)`

Конструируют отсортированное хеш-отображение, безопасно доступное из многих потоков.

Параметры: *initialCapacity*

Исходная емкость данной коллекции.

По умолчанию равна **16**

loadFactor

Управляет изменением размера коллекции: если средняя загрузка на группу превышает этот показатель, то размер хеш-таблицы изменяется

concurrencyLevel

Ожидаемое количество конкурирующих потоков, выполняющих запись

- `ConcurrentSkipListMap<K, V>()`
- `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`
Конструируют отсортированное хеш-отображение, безопасно доступное из многих потоков. Первый конструктор требует, чтобы элементы множества относились к классу, реализующему интерфейс `Comparable`.
- `V putIfAbsent(K key, V value)`
Если ключ отсутствует в отображении, то связывает заданное значение с указанным ключом и возвращает пустое значение `null`. В противном случае возвращает присутствующее значение, связанное с ключом.
- `boolean remove(K key, V value)`
Если указанный ключ связан с данным значением, то удаляет ключ и значение, возвращая логическое значение `true`, а иначе возвращает логическое значение `false`.
- `boolean replace(K key, V oldValue, V newValue)`
Если указанный ключ связан со старым значением `oldValue`, то связывает его с новым значением `newValue`. В противном случае возвращает логическое значение `false`.

Копируемые при записи массивы

Классы `CopyOnWriteArrayList` и `CopyOnWriteArraySet` представляют потокобезопасные коллекции, при любых изменениях в которых создается копия базового массива. Это удобно, если количество потоков, выполняющих обход коллекции, значительно превышает количество потоков, изменяющих ее. Когда конструируется итератор, он содержит ссылку на текущий массив. Если в дальнейшем массив изменяется, итератор по-прежнему ссылается на старую копию массива, а текущий массив коллекции заменяется новым. Как следствие, старому итератору доступно согласованное (хотя и потенциально устаревшее) представление коллекции, не требующее дополнительных издержек на синхронизацию.

Устаревшие потокобезопасные коллекции

С самых первых версий Java классы `Vector` и `Hashtable` предоставляли потокобезопасные реализации динамического массива и хеш-таблицы. Теперь эти классы считаются устаревшими и заменены классами `ArrayList` и `HashMap`. Но эти последние классы не являются потокобезопасными, хотя в библиотеке коллекций предусмотрен другой механизм для обеспечения безопасности потоков. Любой класс может быть сделан потокобезопасным благодаря *синхронизирующей оболочке* следующим образом:

```
List<E> synchArrayList = Collections.synchronizedList(new ArrayList<E>());  
Map<K, V> synchHashMap = Collections.synchronizedMap(new HashMap<K, V>());
```

Методы полученной в результате коллекции защищены блокировкой, обеспечивая потокобезопасный доступ. Но при этом необходимо гарантировать, что ни один поток не обращается к данным через исходные, не синхронизированные методы. Это проще всего сделать, не сохраняя никаких ссылок на исходный объект. Достаточно сконструировать коллекцию и сразу же передать ее оболочке, как показано в приведенных ранее примерах. Но если требуется сделать обход коллекции в то время, как в другом потоке имеется возможность изменить ее, то придется установить клиентскую блокировку, как показано ниже.

```
synchronized (synchHashMap)
{
    Iterator<K> iter = synchHashMap.keySet().iterator();
    while (iter.hasNext()) . . . ;
}
```

Такой же код следует использовать и при организации цикла в стиле `for each`, поскольку в этом цикле применяется итератор. Следует, однако, иметь в виду, что при попытке изменить коллекцию из другого потока, когда происходит ее обход, итератор генерирует исключение типа `ConcurrentModificationException`, свидетельствующее о неудачном исходе обхода коллекции. Кроме того, требуется синхронизация, чтобы просто и надежно обнаружить попытки внести изменения в данные из параллельно действующих потоков.

Но, как правило, вместо синхронизирующих оболочек лучше пользоваться коллекциями, определенными в пакете `java.util.concurrent`. В частности, хеш-отображение типа `ConcurrentHashMap` тщательно реализовано с таким расчетом, чтобы к нему можно было обращаться из многих потоков, не блокирующих друг друга, если они работают с разными группами данных в хеш-таблице. Исключением из этого правила является часто обновляемый списочный массив. В этом случае вместо синхронизированного списочного массива типа `ArrayList` лучше выбрать списочный массив типа `CopyOnWriteArrayList`.

java.util.Collections 1.2

- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
 - `static <E> List synchronizedList(List<E> c)`
 - `static <E> Set synchronizedSet(Set<E> c)`
 - `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
 - `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
 - `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`
- Конструируют представление коллекции с синхронизированными методами.

Интерфейсы Callable и Future

Интерфейс `Runnable` инкапсулирует задачу, выполняющуюся асинхронно. Его можно рассматривать как асинхронный метод без параметров и возвращаемого значения. А интерфейс `Callable` подобен интерфейсу `Runnable`, но в нем предусмотрен возврат значения. Интерфейс `Callable` относится к параметризованному типу и имеет единственный метод `call()`, как показано ниже.

```
public interface Callable<V>
{
    V call() throws Exception;
}
```

Параметр типа обозначает тип возвращаемого значения. Например, интерфейс `Callable<Integer>` представляет асинхронное вычисление, в результате которого возвращается объект типа `Integer`. А сохранение результатов асинхронного

вычисления обеспечивает интерфейс Future. В частности, вычисление можно начать, предоставив кому-нибудь другому объект типа Future, а затем просто забыть о нем. Владелец объекта типа Future может получить результат, когда он будет готов. У интерфейса Future объявляются следующие методы:

```
public interface Future<V>
{
    V get() throws . . .;
    V get(long timeout, TimeUnit unit) throws . . .;
    void cancel(boolean mayInterrupt);
    boolean isCancelled();
    boolean isDone();
}
```

При вызове первого метода get() блокировка устанавливается до тех пор, пока не завершится вычисление. Второй метод get() генерирует исключение типа TimeoutException, если время ожидания истекает до завершения вычислений. Если же прерывается поток, в котором выполняется вычисление, оба метода генерируют исключение типа InterruptedException. А если вычисление уже завершено, то метод get() сразу же возвращает управление. Метод isDone() возвращает логическое значение false, если вычисление продолжается, и логическое значение true, если оно завершено.

Вычисление можно прервать, вызвав метод cancel(). Если вычисление еще не начато, оно отменяется и вообще не начнется. Если же вычисление уже выполняется, оно прерывается, когда параметр метода mayInterrupt() имеет логическое значение true.

Класс-оболочка FutureTask служит удобным механизмом для превращения интерфейса Callable одновременно в интерфейсы Future и Runnable, реализуя оба эти интерфейса. В приведенном ниже коде демонстрируется характерный тому пример.

```
Callable<Integer> myComputation = . . .;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = new Thread(task); // это интерфейс Runnable
t.start();
. . .
Integer result = task.get(); // а это интерфейс Future
```

В примере программы из листинга 14.11 описанные выше понятия демонстрируются непосредственно в коде. Эта программа действует аналогично программе из предыдущего примера, где находятся файлы, содержащие ключевое слово. Но в данном случае просто подсчитывается количество файлов, соответствующих критерию поиска. Таким образом, выполняется долгосрочная задача, результатом которой является целочисленное значение, например, типа Callable<Integer>, как показано ниже.

```
class MatchCounter implements Callable<Integer>
{
    public MatchCounter(File directory, String keyword) { . . . }
    public Integer call() { . . . } // возвращает количество совпадающих
    файлов
}
```

Далее из объекта класса MatchCounter конструируется объект типа FutureTask, который служит для запуска потока следующим образом:

```
FutureTask<Integer> task = new FutureTask<Integer>(counter);
Thread t = new Thread(task);
t.start();
```

И, наконец, выводится результат, как показано ниже. Разумеется, при вызове метода `get()` блокировка устанавливается до тех пор, пока не будет готов результат.

```
System.out.println(task.get() + " matching files.");
```

В теле метода `call()` тот же самый механизм применяется рекурсивно. Для каждого подкаталога создается новый объект типа `MatchCounter`, и для него запускается поток. Кроме того, объекты типа `FutureTask` накапливаются в списочном массиве типа `ArrayList<Future<Integer>>`. И, наконец, все полученные результаты складываются следующим образом:

```
for (Future<Integer> result : results)
    count += result.get();
```

Как пояснялось выше, при каждом вызове метода `get()` блокировка устанавливается до тех пор, пока не будет готов результат. Разумеется, потоки выполняются параллельно, поэтому не исключено, что результаты будут готовы почти одновременно.

Листинг 4.11. Исходный код из файла `future/FutureTest.java`

```
1 package future;
2
3 import java.io.*;
4 import java.util.*;
5 import java.util.concurrent.*;
6 /**
7  * @version 1.01 2012-01-26
8  * @author Cay Horstmann
9  */
10 public class FutureTest
11 {
12     public static void main(String[] args)
13     {
14         Scanner in = new Scanner(System.in);
15         System.out.print(
16             "Enter base directory (e.g. /usr/local/jdk5.0/src): ");
17         String directory = in.nextLine();
18         System.out.print("Enter keyword (e.g. volatile): ");
19         String keyword = in.nextLine();
20
21         MatchCounter counter = new MatchCounter(new File(directory), keyword);
22         FutureTask<Integer> task = new FutureTask<>(counter);
23         Thread t = new Thread(task);
24         t.start();
25         try
26         {
27             System.out.println(task.get() + " matching files.");
28         }
29         catch (ExecutionException e)
30         {
31             e.printStackTrace();
32         }
33         catch (InterruptedException e)
34         {
35         }
36     }
37 }
38
39 /**
```

```
40 * Подсчитывает файлы, содержащие заданное ключевое слово,
41 * в каталоге и его подкаталогах
42 */
43 class MatchCounter implements Callable<Integer>
44 {
45     private File directory;
46     private String keyword;
47     private int count;
48
49     /**
50      * Конструирует объект типа MatchCounter
51      * @param directory Каталог, с которого начинается поиск
52      * @param keyword Искомое ключевое слово
53      */
54     public MatchCounter(File directory, String keyword)
55     {
56         this.directory = directory;
57         this.keyword = keyword;
58     }
59
60     public Integer call()
61     {
62         count = 0;
63         try
64         {
65             File[] files = directory.listFiles();
66             List<Future<Integer>> results = new ArrayList<>();
67
68             for (File file : files)
69                 if (file.isDirectory())
70                 {
71                     MatchCounter counter = new MatchCounter(file, keyword);
72                     FutureTask<Integer> task = new FutureTask<>(counter);
73                     results.add(task);
74                     Thread t = new Thread(task);
75                     t.start();
76                 }
77             else
78             {
79                 if (search(file)) count++;
80             }
81
82             for (Future<Integer> result : results)
83                 try
84                 {
85                     count += result.get();
86                 }
87                 catch (ExecutionException e)
88                 {
89                     e.printStackTrace();
90                 }
91             }
92             catch (InterruptedException e)
93             {
94             }
95             return count;
96         }
97
98     /**
99      * Осуществляет поиск заданного ключевого слова в файлах
```

```
100 * @param file Файл для поиска ключевого слова
101 * @return Возвращает логическое значение true, если
102 *         ключевое слово содержится в файле
103 */
104 public boolean search(File file)
105 {
106     try
107     {
108         try (Scanner in = new Scanner(file))
109         {
110             boolean found = false;
111             while (!found && in.hasNextLine())
112             {
113                 String line = in.nextLine();
114                 if (line.contains(keyword)) found = true;
115             }
116             return found;
117         }
118     }
119     catch (IOException e)
120     {
121         return false;
122     }
123 }
124 }
```

java.util.concurrent.Callable<V> 5.0

- `V call()`
Запускает задачу, выдающую результат.

java.util.concurrent.Future<V> 5.0

- `V get()`
- `V get(long time, TimeUnit unit)`
Получают результат, устанавливая блокировку до момента его готовности или истечения заданного промежутка времени. Второй метод генерирует исключение типа `TimeoutException` при неудачном исходе.
- `boolean cancel(boolean mayInterrupt)`
Пытается отменить выполнение задачи. Если задача уже запущена и параметр `mayInterrupt` принимает логическое значение `true`, она прерывается. Возвращает логическое значение `true` при удачном исходе операции отмены.
- `boolean isCancelled()`
Возвращает логическое значение `true`, если задача была отменена до ее завершения.
- `boolean isDone()`
Возвращает логическое значение `true`, если задача завершена нормально, отменена или прервана вследствие исключения.

java.util.concurrent.FutureTask<V> 5.0

- `FutureTask(Callable<V> task)`
- `FutureTask(Runnable task, V result)`

Конструируют объект, реализующий одновременно интерфейсы `Future<V>` и `Runnable`.

Исполнители

Создание нового потока — довольно дорогостоящая операция с точки зрения потребляемых ресурсов, поскольку она включает в себя взаимодействие с операционной системой. Если в программе создается большое количество кратковременных потоков, то имеет смысл использовать *пул потоков*. В пуле потоков содержится целый ряд простаивающих потоков, готовых к запуску. Так, исполняемый поток типа `Runnable` размещается в пуле, а один из остальных потоков вызывает его метод `run()`. Когда метод `run()` завершается, его поток не уничтожается, но остается в пуле готовым обслужить новый запрос.

Другая причина для использования пула потоков — необходимость ограничить количество параллельно выполняющихся потоков. Создание огромного числа потоков может отрицательно сказаться на производительности и даже привести к полному отказу виртуальной машины. Поэтому, если применяется алгоритм, создающий большое количество потоков, следует установить фиксированный пул потоков, чтобы ограничить общее количество параллельно выполняющихся потоков.

В состав класса `Executors` исполнителей входит целый ряд статических фабричных методов для построения пулов потоков. Их перечень и краткое описание приводятся в табл. 14.2.

Таблица 14.2. Фабричные методы из класса `Executors`

Метод	Описание
<code>newCachedThreadPool()</code>	Новые потоки создаются по мере необходимости, а простаивающие потоки сохраняются в течение 60 секунд
<code>newFixedThreadPool()</code>	Пул содержит фиксированное множество набор потоков, а простаивающие потоки сохраняются в течение 60 секунд
<code>newSingleThreadExecutor()</code>	Пул с единственным потоком, выполняющим переданные ему задачи поочередно (подобно потоку диспетчеризации событий в <code>Swing</code>)
<code>newScheduledThreadPool()</code>	Пул фиксированных потоков для планового запуска; заменяет класс <code>java.util.Timer</code>
<code>newSingleThreadScheduledExecutor()</code>	Пул с единственным потоком для планового запуска на выполнение

Пулы потоков

Рассмотрим первые три метода из табл. 14.2, а остальные методы будут обсуждаться далее, в разделе “Плановое выполнение потоков”. Метод `newCachedThreadPool()`

строит пул потоков, выполняющий каждую задачу немедленно, используя существующий простаивающий поток, если он доступен, а иначе — создавая новый поток. Метод `newFixedThreadPool()` строит пул потоков фиксированного размера. Если количество задач превышает количество простаивающих потоков, то лишние задачи ставятся в очередь на обслуживание, чтобы быть запущенными, когда завершатся текущие выполняющиеся задачи. Метод `newSingleThreadExecutor()` создает выродившийся пул размером в один поток. Задачи, передаваемые в единственный поток, выполняются по очереди. Все три упомянутых метода возвращают объект класса `ThreadPoolExecutor`, реализующего интерфейс `ExecutorService`.

Передать задачу типа `Runnable` или `Callable` объекту типа `ExecutorService` можно одним из следующих вариантов метода `submit()`:

```
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
Future<T> submit(Callable<T> task)
```

Пул запускает переданную задачу при первом удобном случае. В результате вызова метода `submit()` возвращается объект типа `Future`, который можно использовать для опроса состояния задачи. Первый вариант метода `submit()` возвращает объект необычного типа `Future<?>`. Он служит для вызова метода `isDone()`, `cancel()` или `isCancelled()`. Но по завершении метода `get()` возвращается только пустое значение `null`. Второй вариант метода `submit()` также принимает объект типа `Runnable`, а по завершении метода `get()`, вызываемого для объекта типа `Future`, возвращается заданный объект `result`. И третий вариант метода `submit()` принимает объект типа `Callable`, а возвращаемый объект типа `Future` получает результат вычисления по его готовности.

По завершении работы с пулом потоков следует вызвать метод `shutdown()`. Этот метод инициирует последовательность закрытия пула, после чего новые задачи не принимаются на выполнение. По завершении всех задач потоки в пуле уничтожаются. В качестве альтернативы можно вызвать метод `shutdownNow()`. В этом случае пул отменяет все задачи, которые еще не запущены, пытаясь прервать выполняющиеся потоки.

Ниже перечислены действия для организации пула потоков.

1. Вызовите статический метод `newCachedThreadPool()` или `newFixedThreadPool()` из класса `Executors`.
2. Вызовите метод `submit()` для передачи объектов `Runnable` или `Callable` в пул потоков.
3. Полагайтесь на возвращаемые объекты типа `Future`, чтобы иметь возможность прервать задачу или передать объекты типа `Callable`.
4. Вызовите метод `shutdown()`, если больше не собираетесь запускать новые задачи.

В программе из предыдущего примера создается большое количество кратковременных потоков — по одному на каждый каталог. А в примере программы из листинга 14.12 для запуска задач вместо этого используется пул потоков. Следует иметь в виду, что эта программа выводит максимальный размер пула, достигнутый во время выполнения. Эти сведения недоступны через интерфейс `ExecutorService`, и поэтому приходится выполнять приведение типа объекта пула к классу `ThreadPoolExecutor`.

Листинг 14.12. Исходный код из файла threadPool/ThreadPoolTest.java

```
1 package threadPool;
2
3 import java.io.*;
4 import java.util.*;
5 import java.util.concurrent.*;
6
7 /**
8  * @version 1.01 2012-01-26
9  * @author Cay Horstmann
10 */
11 public class ThreadPoolTest
12 {
13     public static void main(String[] args) throws Exception
14     {
15         Scanner in = new Scanner(System.in);
16         System.out.print(
17             "Enter base directory (e.g. /usr/local/jdk5.0/src): ");
18         String directory = in.nextLine();
19         System.out.print("Enter keyword (e.g. volatile): ");
20         String keyword = in.nextLine();
21
22         ExecutorService pool = Executors.newCachedThreadPool();
23
24         MatchCounter counter =
25             new MatchCounter(new File(directory), keyword, pool);
26         Future<Integer> result = pool.submit(counter);
27
28         try
29         {
30             System.out.println(result.get() + " matching files.");
31         }
32         catch (ExecutionException e)
33         {
34             e.printStackTrace();
35         }
36         catch (InterruptedException e)
37         {
38         }
39         pool.shutdown();
40
41         int largestPoolSize =
42             ((ThreadPoolExecutor) pool).getLargestPoolSize();
43         System.out.println("largest pool size=" + largestPoolSize);
44     }
45 }
46 /**
47  * Подсчитывает файлы, содержащие заданное ключевое слово,
48  * в каталоге и его подкаталогах
49 */
50 class MatchCounter implements Callable<Integer>
51 {
52     private File directory;
53     private String keyword;
54     private ExecutorService pool;
55     private int count;
```

```
56
57  /**
58   * Конструирует объект типа MatchCounter
59   * @param directory Каталог, с которого начинается поиск
60   * @param keyword Искомое ключевое слово
61   * @param pool Пул потоков для передачи подзадач на выполнение
62   */
63  public MatchCounter(
64      File directory, String keyword, ExecutorService pool)
65  {
66      this.directory = directory;
67      this.keyword = keyword;
68      this.pool = pool;
69  }
70
71  public Integer call()
72  {
73      count = 0;
74      try
75      {
76          File[] files = directory.listFiles();
77          List<Future<Integer>> results = new ArrayList<>();
78
79          for (File file : files)
80              if (file.isDirectory())
81                  {
82                      MatchCounter counter = new MatchCounter(file, keyword, pool);
83                      Future<Integer> result = pool.submit(counter);
84                      results.add(result);
85                  }
86              else
87                  {
88                      if (search(file)) count++;
89                  }
90
91          for (Future<Integer> result : results)
92              try
93                  {
94                      count += result.get();
95                  }
96              catch (ExecutionException e)
97                  {
98                      e.printStackTrace();
99                  }
100          }
101          catch (InterruptedException e)
102          {
103          }
104          return count;
105      }
106  /**
107   * Осуществляет поиск заданного ключевого слова в файлах
108   * @param File для поиска ключевого слова
109   * @return Возвращает логическое значение true, если
110   *         ключевое слово содержится в файле
111   */
112  public boolean search(File file)
113  {
```

```
114     try
115     {
116         try (Scanner in = new Scanner(file))
117         {
118             boolean found = false;
119             while (!found && in.hasNextLine())
120             {
121                 String line = in.nextLine();
122                 if (line.contains(keyword)) found = true;
123             }
124             return found;
125         }
126     }
127     catch (IOException e)
128     {
129         return false;
130     }
131 }
```

java.util.concurrent.Executors 5.0

- `ExecutorService newCachedThreadPool()`
Возвращает кешированный пул потоков, создающий потоки по мере необходимости и закрывающий потоки, которые простаивают более 60 секунд.
- `ExecutorService newFixedThreadPool(int threads)`
Возвращает пул потоков, использующий заданное количество потоков для запуска задач.
- `ExecutorService newSingleThreadExecutor()`
Возвращает исполнитель, поочередно запускающий задачи на выполнение в одном потоке.

java.util.concurrent.ExecutorService 5.0

- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`
- `Future<?> submit(Runnable task)`
Передают указанную задачу на выполнение.
- `void shutdown()`
Останавливает службу, завершая все запущенные задачи и не принимая новых.

java.util.concurrent.ThreadPoolExecutor 5.0

- `int getLargestPoolSize()`
Возвращает максимальный размер пула потоков, который был достигнут в течение срока его существования.

Плановое выполнение потоков

В состав интерфейса `ScheduledExecutorService` входят методы для планирования или многократного выполнения задач. Это — обобщение класса `java.util.Timer`, позволяющее организовать пул потоков. Методы `newScheduledThreadPool()` и `newSingleThreadScheduledExecutor()` из класса `Executors` возвращают объекты, реализующие интерфейс `ScheduledExecutorService`. Имеется возможность запланировать однократный запуск задач типа `Runnable` или `Callable` по истечении некоторого времени, а также периодический запуск задач типа `Runnable`. Более подробно средства планового выполнения потоков представлены ниже в описании соответствующего прикладного интерфейса API.

`java.util.concurrent.Executors 5.0`

- `ScheduledExecutorService newScheduledThreadPool(int threads)`
Возвращает пул потоков, использующий заданное их количество для планирования запуска задач.
- `ScheduledExecutorService newSingleThreadScheduledExecutor()`
Возвращает исполнитель, планирующий поочередный запуск задач в одном потоке.

`java.util.concurrent.ScheduledExecutorService 5.0`

- `ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)`
- `ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)`
Планируют запуск указанной задачи по истечении заданного промежутка времени.
- `ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`
Планирует периодический запуск указанной задачи через каждые `period` промежутков времени в единицах `unit` по истечении первоначального времени задержки `initialDelay`.
- `ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`
Планирует периодический запуск указанной задачи со временем задержки `delay` в единицах `unit` между запусками по истечении первоначального времени задержки `initialDelay`.

Управление группами задач

Ранее было показано, как пользоваться службой исполнителей в виде пула потоков для повышения эффективности выполнения задач. Но иногда исполнитель меняется скорее по причинам тактического характера только для управления группами взаимосвязанных задач. Например, все задачи можно прервать в исполнителе, вызвав его метод `shutdownNow()`.

Метод `invokeAny()` передает исполнителю все объекты типа `Callable` из коллекции и возвращает результат выполненной задачи. Но заранее неизвестно, к какой именно задаче относится возвращаемый результат. Можно лишь предположить, что это будет результат выполнения задачи, которая завершилась быстрее всех. Поэтому данный метод вызывается для поиска любого приемлемого решения поставленной задачи. Допустим, требуется разложить на множители большие целые числа — вычисление, необходимое для взлома шифра RSA. С этой целью можно запустить целый ряд задач, в каждой из которых будет предпринята попытка найти множители в отдельном диапазоне чисел. Как только в одной из задач будет найдено решение, все остальные вычисления можно прекратить.

Метод `invokeAll()` запускает все объекты типа `Callable` из коллекции и возвращает список объектов типа `Future`, представляющих результаты выполнения всех задач. Обработку этих результатов можно организовать по мере их готовности следующим образом:

```
List<Callable<T>> tasks = . . .;
List<Future<T>> results = executor.invokeAll(tasks);
for (Future<T> result : results)
    processFurther(result.get());
```

Недостаток такого подхода состоит в том, что ждать, возможно, придется слишком долго, если на выполнение первой задачи уйдет довольно много времени. Поэтому целесообразнее получать результаты по мере их готовности. Это можно сделать с помощью класса `ExecutorCompletionService`.

С этой целью запустите исполнитель, полученный обычным способом. Затем сконструируйте объект типа `ExecutorCompletionService`. Передайте задачи данному экземпляру службы исполнителей. Эта служба управляет блокирующей очередью объектов типа `Future`, содержащих результаты переданных на выполнение задач, причем очередь заполняется по мере готовности результатов. Таким образом, вычисления можно организовать более эффективно, как показано ниже.

```
ExecutorCompletionService service = new ExecutorCompletionService(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++)
    processFurther(service.take().get());
```

java.util.concurrent.ExecutorService 5.0

- `T invokeAny(Collection<Callable<T>> tasks)`
- `T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`
Выполняют указанные задачи и возвращают результат выполнения одной из них. Второй метод генерирует исключение типа `TimeoutException` по истечении времени ожидания.
- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks)`
- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`
Выполняют указанные задачи и возвращают результаты выполнения всех задач. Второй метод генерирует исключение типа `TimeoutException` по истечении времени ожидания.

java.util.concurrent.ExecutorCompletionService 5.0

- `ExecutorCompletionService(Executor e)`
- Конструирует службу завершения исполнителя, которая собирает результаты работы заданного исполнителя.
- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`
Передают на выполнение задачу базовому исполнителю.
- `Future<T> take()`
Удаляет следующий готовый результат, устанавливая блокировку, если готовых результатов нет.
- `Future<T> poll()`
- `Future<T> poll(long time, TimeUnit unit)`
Удаляют следующий готовый результат или пустое значение `null`, если готовых результатов нет. Второй метод ожидает в течение указанного промежутка времени.

Архитектура вилочного соединения

В одних приложениях используется большое количество потоков, которые в основном простаивают. Примером тому может служить веб-сервер, использующий по одному потоку на каждое соединение. А в других приложениях на каждое ядро процесса приходится по одному потоку для выполнения таких задач, требующих интенсивных вычислений, как обработка изображений и видеозаписей. Для поддержки именно таких приложений в версии Java SE 7 появилась архитектура вилочного соединения. Допустим, имеется задача обработки, естественно разделяемая на подзадачи следующим образом:

```
if (problemSize < threshold)
    решить задачу непосредственно
else
{
    разделить задачу на подзадачи, решить каждую подзадачу
    рекурсивно и объединить полученные результаты
}
```

Примером такой задачи служит обработка изображений. Для увеличения изображения можно преобразовать по отдельности верхнюю и нижнюю его половины. Если для подобных операций имеется достаточно свободных процессоров, то их выполнение можно распараллелить. (Безусловно, для разделения обрабатываемого изображения на две половины потребуются дополнительные действия, но это уже технические подробности, не имеющие отношения к делу.)

Обратимся к более простому примеру. Предположим, требуется подсчитать количество элементов в массиве с определенным свойством. Для этого массив разделяется на две половины, подсчитываются соответствующие элементы в каждой из них, а результаты затем складываются.

Чтобы облечь рекурсивные вычисления в форму, пригодную для архитектуры вилочного соединения, сначала предоставляется класс, расширяющий класс `RecursiveTask<T>` (если результат вычисления относится к типу `T`) или же класс

RecursiveAction (если получение результата не предполагается). Затем переопределяется метод `compute()` для формирования и вызова подзадач, а также объединения результатов их выполнения. Ниже приведен соответствующий пример кода.

```
class Counter extends RecursiveTask<Integer>
{
    . . .
    protected Integer compute()
    {
        if (to - from < THRESHOLD)
        {
            решить задачу непосредственно
        }
        else
        {
            int mid = (from + to) / 2;
            Counter first = new Counter(values, from, mid, filter);
            Counter second = new Counter(values, mid, to, filter);
            invokeAll(first, second);
            return first.join() + second.join();
        }
    }
}
```

В данном примере метод `invokeAll()` получает ряд задач и блоков до тех пор, пока их выполнение не будет завершено, а метод `join()` объединяет полученные результаты. В частности, метод `join()` вызывается для каждой подзадачи, а в итоге возвращается сумма результатов их выполнения.



НА ЗАМЕТКУ! Имеется также метод `get()` для получения текущего результата, но он менее привлекателен, поскольку может сгенерировать проверяемые исключения, которые не допускается генерировать в методе `compute()`.

Весь исходный код рассматриваемого здесь примера приведен в листинге 14.13. В самой архитектуре вилочного соединения применяется эффективный эвристический алгоритм для уравнивания рабочей нагрузки на имеющиеся потоки, называемый *перехватом работы*. Для каждого рабочего потока организуется двухсторонняя очередь выполняемых задач. Рабочий поток размещает подзадачи в голове своей двухсторонней очереди, причем только один поток имеет доступ к голове этой очереди, благодаря чему исключается потребность в блокировке потоков. Когда рабочий процесс простаивает, он пытается перехватить задачу из хвоста другой двухсторонней очереди, но поскольку в хвосте очереди обычно располагаются крупные подзадачи, то потребность в перехвате задач возникает редко.

Листинг 14.13. Исходный код из файла `forkJoin/forkJoinTest.java`

```
1 package forkJoin;
2
3 import java.util.concurrent.*;
4
5 /**
6  * В этой программе демонстрируется архитектура вилочного соединения
7  * @version 1.00 2012-05-20
8  * @author Cay Horstmann
```

```
9 */
10 public class ForkJoinTest
11 {
12     public static void main(String[] args)
13     {
14         final int SIZE = 10000000;
15         double[] numbers = new double[SIZE];
16         for (int i = 0; i < SIZE; i++) numbers[i] = Math.random();
17         Counter counter = new Counter(numbers, 0, numbers.length,
18             new Filter()
19             {
20                 public boolean accept(double x) { return x > 0.5; }
21             });
22         ForkJoinPool pool = new ForkJoinPool();
23         pool.invoke(counter);
24         System.out.println(counter.join());
25     }
26 }
27 interface Filter
28 {
29     boolean accept(double t);
30 }
31
32 class Counter extends RecursiveTask<Integer>
33 {
34     public static final int THRESHOLD = 1000;
35     private double[] values;
36     private int from;
37     private int to;
38     private Filter filter;
39
40     public Counter(double[] values, int from, int to, Filter filter)
41     {
42         this.values = values;
43         this.from = from;
44         this.to = to;
45         this.filter = filter;
46     }
47     protected Integer compute()
48     {
49         if (to - from < THRESHOLD)
50         {
51             int count = 0;
52             for (int i = from; i < to; i++)
53             {
54                 if (filter.accept(values[i])) count++;
55             }
56             return count;
57         }
58         else
59         {
60             int mid = (from + to) / 2;
61             Counter first = new Counter(values, from, mid, filter);
62             Counter second = new Counter(values, mid, to, filter);
63             invokeAll(first, second);
64             return first.join() + second.join();
65         }
66     }
67 }
```

Синхронизаторы

В состав пакета `java.util.concurrent` входит несколько классов, упрощающих управление рядом взаимодействующих задач (табл. 14.3). Эти механизмы имеют “готовые” функциональные возможности для часто встречающихся образцов взаимодействия потоков. Так, если имеется ряд взаимодействующих потоков, которые следуют одному из этих образцов поведения, то нужно воспользоваться соответствующим библиотечным классом вместо того, чтобы пытаться составлять самодельную коллекцию блокировок и условий.

Таблица 14.3. Синхронизаторы

Класс	Назначение	Применение
<code>CyclicBarrier</code>	Дает набору потоков возможность ожидать до тех пор, пока их счетчик достигнет общего барьера, а затем дополнительно выполнить некоторое барьерное действие	Когда должно быть завершено определенное количество потоков, прежде чем можно будет воспользоваться результатами их выполнения. Барьер может быть использован повторно, как только освободятся ожидающие потоки
<code>Phaser</code>	Аналогичен предыдущему классу, но с подсчетом изменяемых участников барьера	Внедрен в версии Java SE 7
<code>CountDownLatch</code>	Дает набору потоков возможность ожидать до тех пор, пока значение счетчика уменьшится до нуля	Когда одному или нескольким потокам приходится ожидать наступления заданного количества событий
<code>Exchanger</code>	Дает двум потокам возможность обмениваться объектами, когда они к этому готовы	Когда два потока работают с двумя экземплярами одной и той же структуры данных, причем один из них заполняет один экземпляр, а второй — опустошает другой
<code>Semaphore</code>	Дает набору потоков возможность ожидать до тех пор, пока им будет разрешено продолжить свое выполнение	Для ограничения общего числа потоков, имеющих доступ к ресурсу. Если допустимое число равно одному потоку, то все остальные потоки блокируются до тех пор, пока владеющий ресурсом поток не даст им свое разрешение
<code>SynchronousQueue</code>	Дает одному потоку возможность передавать объект другому потоку	Для передачи объекта из одного потока в другой, когда оба потока готовы, не требуя явной синхронизации

Семафоры

По существу, семафоры управляют множеством *разрешений*. Чтобы пройти семафор, поток запрашивает разрешение, вызывая метод `acquire()`. Доступно только фиксированное число разрешений, ограничивающее количество потоков, которые могут пройти семафор. Другие потоки могут выдавать разрешения, вызывая метод `release()`. Объектов разрешений на самом деле не существует. Семафор просто поддерживает счетчик. Более того, разрешение не может быть снято потоком, который получил его. В действительности любой поток может выдать любое число

разрешений. Но если их количество превысит допустимый максимум, то семафор просто установит максимальное значение в своем счетчике.

Семафоры были изобретены Эдсгером Дейкстрой (Edsger Dijkstra) в 1968 году для применения в качестве *примитивов синхронизации*. Дейкстра доказал, что семафоры могут быть эффективно реализованы и вполне способны разрешить многие затруднения, часто возникающие при синхронизации. Почти во всей литературе по операционным системам можно найти реализации ограниченных очередей, использующих семафоры. Разумеется, прикладные программисты не должны заново изобретать ограниченные очереди. А семафоры обычно не годятся для разработки прикладных программ.

Защелки с обратным отсчетом

Класс `CountDownLatch` дает ряду потоков возможность ожидать до тех пор, пока значение счетчика достигнет нуля. Защелка с обратным отсчетом срабатывает только один раз. Как только счетчик достигнет нуля, увеличить его снова нельзя. Полезным частным случаем является защелка со счетчиком, равным 1. Она реализует односторонний вентиль. Один поток задерживается вентиляем до тех пор, пока другой поток не установит счетчик в нуль. Представьте, например, ряд потоков, которым для своего выполнения требуются некоторые начальные данные. Один рабочий поток запускается и ожидает открытия вентиля, а другой — подготавливает данные. Когда данные готовы, он вызывает метод `countDown()` и тем самым открывает вентиль для всех остальных рабочих потоков.

Далее можно воспользоваться другой защелкой, чтобы выяснить, когда завершатся все рабочие потоки. Для этого защелка инициализируется количеством потоков. Каждый рабочий поток перед своим завершением уменьшает значение счетчика защелки, а другой поток, собирающий результаты выполнения других потоков, ожидает на защелке и продолжает свое выполнение, как только все рабочие потоки завершатся.

Барьеры

Класс `CyclicBarrier` реализует рандеву, называемое *барьером*. Рассмотрим ряд потоков, каждый из которых выполняет отдельную часть общих вычислений. Когда все части готовы, их результаты должны быть объединены. Если поток завершает свою часть вычислений, ему разрешается достичь барьера. Как только все потоки достигнут барьера, барьер открывается, и потоки могут выполняться дальше. Обратимся к конкретному примеру. Сначала конструируется барьер, которому передается количество участвующих потоков:

```
CyclicBarrier barrier = new CyclicBarrier(nthreads);
```

Каждый поток выполняет некоторую работу и вызывает метод `await()` на барьере до завершения, как показано ниже.

```
public void run()
{
    doWork();
    barrier.await();
    . . .
}
```

Метод `await()` принимает время ожидания в качестве необязательного параметра, как показано в приведенной ниже строке кода.

```
barrier.await(100, TimeUnit.MILLISECONDS);
```

Если любой из потоков, ожидающих барьера, покинет барьер, то барьер *нарушается*. (Поток может покинуть барьер, если он вызвал метод `await()` со временем ожидания или из-за прерывания.) В этом случае метод `await()` для всех прочих потоков генерирует исключение типа `BrokenBarrierException`. Вызов метода `await()` в уже ожидающих потоках немедленно прекращается. Кроме того, можно применить необязательное *барьерное действие*, которое выполняется, как только все потоки достигают барьера:

```
Runnable barrierAction = . . . ;
CyclicBarrier barrier = new CyclicBarrier(nthreads, barrierAction);
```

Такое действие позволяет собрать результаты выполнения индивидуальных потоков. Барьер называется *циклическим*, потому что он может быть повторно использован после того, как все ожидающие потоки будут освобождены. В этом отношении он отличается от защелки с обратным отсчетом, которая может быть использована лишь однократно. Еще больше удобств для организации барьеров предоставляет класс `Phaser`, позволяя изменять число участвующих потоков на разных стадиях процесса синхронизации.

Обменники

Класс обменников `Exchanger` используется в тех случаях, когда два потока работают с двумя экземплярами одного и того же буфера данных. Как правило, один из потоков заполняет буфер, а другой потребляет его содержимое. Когда оба потока завершают свое выполнение, происходит обмен буферами.

Синхронные очереди

Синхронная очередь — это механизм, объединяющий поставляющий и потребляющий потоки. Когда в потоке вызывается метод `put()` для объекта типа `SynchronousQueue`, он блокируется до тех пор, пока в другом потоке не будет вызван метод `take()`, и наоборот. В отличие от класса `Exchanger`, данные передаются только в одном направлении — от поставляющего потока к потребляющему. Несмотря на то что класс `SynchronousQueue` реализует интерфейс `BlockingQueue`, он, по существу, не организует очередь и не содержит никаких элементов, а его метод `size()` всегда возвращает нулевое значение.

Потоки и библиотека Swing

Как упоминалось в самом начале этой главы, одной из причин для использования потоков в программе является необходимость повысить ее реакцию на внешние воздействия. Если прикладная программа должна выполнять какую-нибудь длительную задачу, ее следует запускать в новом рабочем потоке вместо того, чтобы блокировать пользовательский интерфейс.

Следует, однако, соблюдать осторожность в отношении того, что делается в рабочем потоке, поскольку библиотека `Swing`, как ни странно, *не* является потокобезопасной. Так, если попытаться манипулировать элементами пользовательского интерфейса из нескольких потоков, то можно повредить этот пользовательский интерфейс.

Чтобы лучше понять существо рассматриваемого здесь вопроса, запустите текстовую программу из листинга 14.14. Как только вы щелкнете на кнопке `Bad` (Плохо),

запустится новый поток, метод `run()` которого буквально истязает комбинированный список, случайным образом вводя и удаляя из него элементы, как показано ниже.

```
public void run()
{
    try
    {
        while (true)
        {
            int i = Math.abs(generator.nextInt());
            if (i % 2 == 0)
                combo.insertItemAt(new Integer(i), 0);
            else if (combo.getItemCount() > 0)
                combo.removeItemAt(i % combo.getItemCount());
            sleep(1);
        }
    } catch (InterruptedException e) {}
}
```

Итак, попробуйте щелкнуть на кнопке `Bad`, затем несколько раз щелкните на комбинированном списке. Перетащите бегунок на полосе прокрутки и переместите окно. Еще раз щелкните на кнопке `Bad`. Продолжайте щелкать на комбинированном списке. В конце концов, вы увидите отчет об исключении (рис. 14.8).

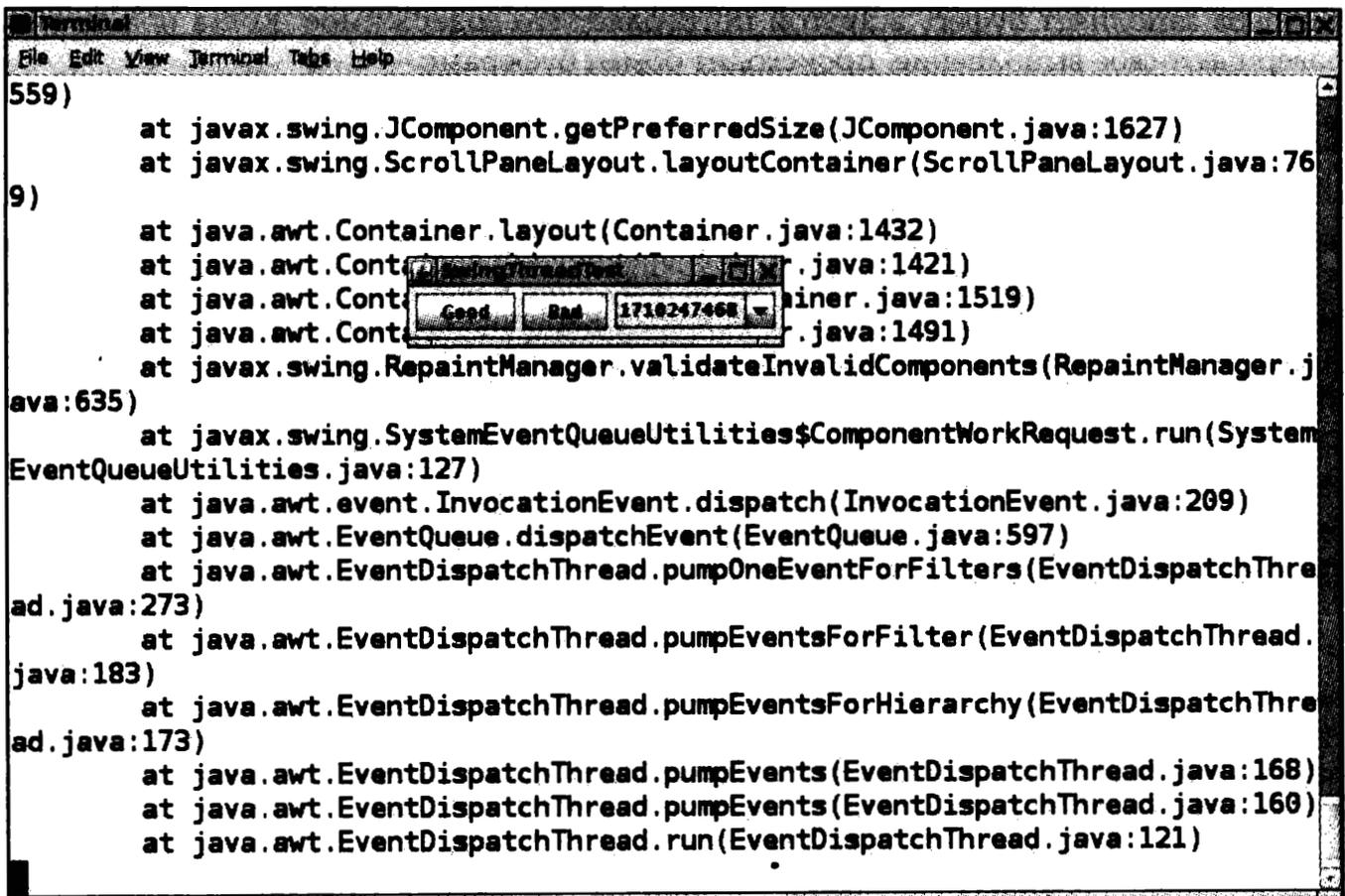


Рис. 14.8. Отчет об исключении, выводимый на консоль

Что же происходит? Когда элемент вводится в комбинированный список, последний инициирует событие для обновления своего отображения в окне. Затем вступает в действие код отображения, считывая текущий размер комбинированного списка и

подготавливая отображение соответствующих значений. Но рабочий поток продолжает работать и в какой-то момент уменьшает количество элементов в комбинированном списке. Код отображения полагает, что значений в модели больше, чем есть на самом деле, запрашивает несуществующие значения и генерирует исключение типа `ArrayIndexOutOfBoundsException`.

Такой ситуации можно было бы избежать, позволив прикладным программистам блокировать объект комбинированного списка на время его отображения. Но по двум причинам разработчики Swing решили не тратить усилий, чтобы сделать эту библиотеку потокобезопасной. Прежде всего, синхронизация требует времени, а никому не хочется еще больше замедлять Swing. Но важнее другое: создатели Swing изучили опыт других разработчиков, которые имели дело с потокобезопасными наборами инструментальных средств пользовательского интерфейса. То, что они обнаружили, не внушало оптимизма. Прикладные программисты, пользующиеся потокобезопасными наборами инструментальных средств, путались в требованиях синхронизации и нередко создавали программы, склонные входить в состояние взаимной блокировки.

Выполнение продолжительных задач

Пользуясь потоками вместе с библиотекой Swing, соблюдайте два простых правила.

1. Если действие требует много времени, выполняйте его в отдельном рабочем потоке и никогда — в потоке диспетчеризации событий.
2. Ни в коем случае не трогайте компоненты Swing ни в одном из потоков, кроме потока диспетчеризации событий.

Причину соблюдения первого правила понять нетрудно. Если надолго перейти в поток диспетчеризации событий, то пользователю покажется, что приложение зависло, поскольку оно не в состоянии реагировать ни на какие события. В частности, ни в коем случае не следует выполнять вызовы ввода-вывода в потоке диспетчеризации событий, поскольку они могут заблокировать его навсегда, а также вызывать в нем метод `sleep()`. (Если требуется организовать ожидание в течение некоторого промежутка времени, применяйте события таймера.) Второе правило часто называют *правилом единственного потока* для программирования в Swing. Речь о нем пойдет далее.

На первый взгляд эти два правила противоречат друг другу. Предположим, что отдельный поток запущен для выполнения продолжительной задачи. Обычно ГПИ требуется обновлять для отражения хода выполнения задачи в этом потоке. Когда задача завершится, ГПИ придется обновить снова. Но ведь трогать компоненты Swing в своем потоке нельзя. Так, если придется обновить индикатор выполнения задачи или текст метки, то сделать это из своего потока просто не удастся.

В качестве выхода из этого положения можно воспользоваться двумя служебными методами в любом из потоков, чтобы ввести произвольные действия в очередь событий. Допустим, требуется периодически обновлять метку в потоке, чтобы отобразить ход выполнения задачи. Вызывать метод `label.setText()` из своего потока нельзя. Вместо этого следует воспользоваться методами `invokeLater()` и `invokeAndWait()` из класса `EventQueue`, чтобы осуществить этот вызов в потоке диспетчеризации событий.

С этой целью код Swing вводится в тело метода `run()` того класса, который реализует интерфейс `Runnable`. Затем создается объект этого класса, который передается

статическому методу `invokeLater()` или `invokeAndWait()`. В приведенном ниже примере кода показано, как обновить текст метки.

```
EventQueue.invokeLater(new
    Runnable()
    {
        public void run()
        {
            label.setText(percentage + "% complete");
        }
    });
```

Метод `invokeLater()` возвратит управление, как только событие будет отправлено в очередь событий. Метод `run()` выполняется асинхронно, тогда как метод `invokeAndWait()` ожидает до тех пор, пока фактически завершится выполнение метода `run()`.

Для обновления метки хода выполнения задачи подходит метод `invokeLater()`. Пользователям важнее, чтобы выполнение задачи продвинулось в рабочем потоке как можно дальше, чем иметь более точный индикатор этого процесса. Оба рассматриваемых здесь метода выполняют метод `run()` в потоке диспетчеризации событий. Никакого дополнительного потока не создается.

В примере программы из листинга 14.14 демонстрируется применение метода `invokeLater()` для безопасного внесения изменений в комбинированный список. Если щелкнуть на кнопке Good (Хорошо), числа будут введены и удалены в отдельном потоке. Но сами изменения произойдут непосредственно в потоке диспетчеризации событий.

Листинг 14.14. Исходный код из файла `swing/SwingThreadTest.java`

```
1 package swing;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.util.*;
6 import javax.swing.*;
7
8 /**
9  * В этой программе демонстрируется, что поток, выполняющийся
10 * параллельно с потоком диспетчеризации событий, может вызвать
11 * ошибки в работе компонентов Swing
12 * @version 1.23 2007-05-17
13 * @author Cay Horstmann
14 */
15 public class SwingThreadTest
16 {
17     public static void main(String[] args)
18     {
19         EventQueue.invokeLater(new Runnable()
20         {
21             public void run()
22             {
23                 JFrame frame = new SwingThreadFrame();
24                 frame.setTitle("SwingThreadTest");
25                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26                 frame.setVisible(true);
27             }
28         });
29     }
30 }
```

```
28     });
29   }
30 }
31
32 /**
33  * В этом фрейме имеются две кнопки для заполнения комбинированного
34  * списка из отдельного потока. Кнопка Good обновляет этот список
35  * через очередь событий, а кнопка Bad – непосредственно
36  */
37 class SwingThreadFrame extends JFrame
38 {
39   public SwingThreadFrame()
40   {
41     final JComboBox<Integer> combo = new JComboBox<>();
42     combo.insertItemAt(Integer.MAX_VALUE, 0);
43     combo.setPrototypeDisplayValue(combo.getItemAt(0));
44     combo.setSelectedIndex(0);
45
46     JPanel panel = new JPanel();
47
48     JButton goodButton = new JButton("Good");
49     goodButton.addActionListener(new ActionListener()
50     {
51       public void actionPerformed(ActionEvent event)
52       {
53         new Thread(new GoodWorkerRunnable(combo)).start();
54       }
55     });
56     panel.add(goodButton);
57     JButton badButton = new JButton("Bad");
58     badButton.addActionListener(new ActionListener()
59     {
60       public void actionPerformed(ActionEvent event)
61       {
62         new Thread(new BadWorkerRunnable(combo)).start();
63       }
64     });
65     panel.add(badButton);
66
67     panel.add(combo);
68     add(panel);
69     pack();
70   }
71 }
72
73 /**
74  * В этом исполняемом потоке комбинированный список видоизменяется
75  * путем произвольного ввода и удаления чисел. Это может привести к
76  * ошибкам, так как методы видоизменения комбинированного списка не
77  * синхронизированы, и поэтому этот список доступен как из рабочего
78  * потока, так из потока диспетчеризации событий
79  */
80 class BadWorkerRunnable implements Runnable
81 {
82   private JComboBox<Integer> combo;
83   private Random generator;
84
85   public BadWorkerRunnable(JComboBox<Integer> aCombo)
```

```
86  {
87      combo = aCombo;
88      generator = new Random();
89  }
90
91  public void run()
92  {
93      try
94      {
95          while (true)
96          {
97              int i = Math.abs(generator.nextInt());
98              if (i % 2 == 0) combo.insertItemAt(i, 0);
99              else if (combo.getItemCount() > 0)
100                 combo.removeItemAt(i % combo.getItemCount());
101              Thread.sleep(1);
102          }
103      }
104      catch (InterruptedException e)
105      {
106      }
107  }
108 }
109 /**
110  * Этот исполняемый поток видоизменяет комбинированный список,
111  * произвольно вводя и удаляя числа. Чтобы исключить нарушение
112  * содержимого этого списка, редактирующие операции направляются
113  * в поток диспетчеризации событий
114  */
115 class GoodWorkerRunnable implements Runnable
116 {
117     private JComboBox<Integer> combo;
118     private Random generator;
119
120     public GoodWorkerRunnable(JComboBox<Integer> aCombo)
121     {
122         combo = aCombo;
123         generator = new Random();
124     }
125
126     public void run()
127     {
128         try
129         {
130             while (true)
131             {
132                 EventQueue.invokeLater(new Runnable()
133                 {
134                     public void run()
135                     {
136                         int i = Math.abs(generator.nextInt());
137                         if (i % 2 == 0) combo.insertItemAt(i, 0);
138                         else if (combo.getItemCount() > 0)
139                             combo.removeItemAt(i % combo.getItemCount());
140                     }
141                 });
142             }
143         }
144     }
145 }
```

```
141     Thread.sleep(1);
142     }
143 }
144 catch (InterruptedException e)
145 {
146 }
147 }
148 }
```

java.awt.EventQueue 1.1

- static void invokeLater(Runnable runnable) 1.2

Вынуждает метод run() объекта runnable выполняться в потоке диспетчеризации событий после того, как будут обработаны все ожидающие события.

- static void invokeAndWait(Runnable runnable) 1.2

Вынуждает метод run() объекта runnable выполняться в потоке диспетчеризации событий после того, как будут обработаны все ожидающие события. При вызове этого метода блокировка устанавливается до тех пор, пока не завершится метод run().

- static boolean isDispatchThread() 1.2

Возвращает логическое значение true, если этот метод выполняется в потоке диспетчеризации событий.

Применение класса SwingWorker

Когда пользователь выдает команду, для выполнения которой требуется много времени, запускается новый поток, чтобы выполнить эту команду. Как было показано в предыдущем разделе, в этом потоке должен вызываться метод EventQueue.invokeLater() для обновления GUI.

В примере программы из листинга 14.15 предоставляются команды для загрузки текстового файла и отмены процесса загрузки. Попробуйте запустить ее вместе с крупным текстовым файлом, например, файлом, содержащим полный англоязычный текст романа "Граф Монте-Кристо" и находящимся в каталоге gutenberг загружаемого кода примеров к этой книге. Такой файл загружается в отдельном потоке, и в этот момент пункт меню Open (Открыть) остается недоступным, а пункт меню Cancel (Отмена) — наоборот, доступным (рис. 14.9). После ввода из файла каждой текстовой строки обновляется счетчик в строке состояния. По завершении процесса загрузки пункт меню Open становится доступным, тогда как пункт меню Cancel — недоступным, а в строке состояния появляется сообщение "Done" (Готово).

Этот пример демонстрирует следующие типичные действия в пользовательском интерфейсе для выполнения фоновой задачи.

- После каждой единицы работы пользовательский интерфейс обновляется, чтобы показать ход ее выполнения.
- По завершении всей работы в пользовательский интерфейс вносятся окончательные изменения.

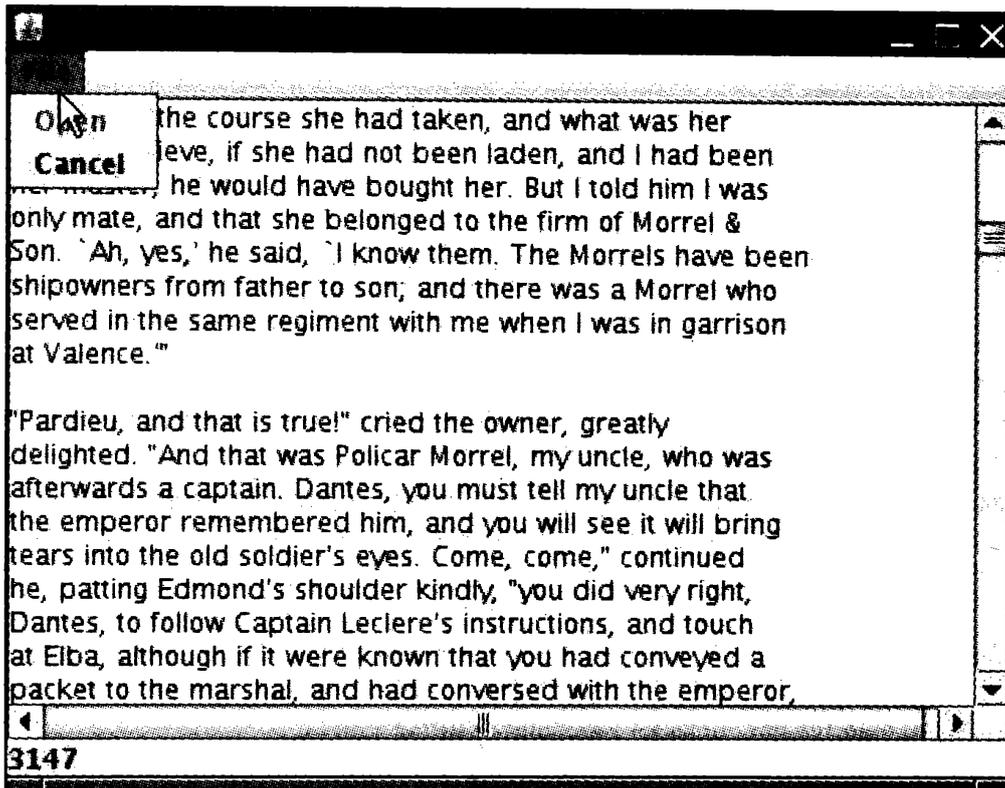


Рис. 14.9. Загрузка файла в отдельном потоке

Класс `SwingWorker` облегчает реализацию данной задачи. Для этого переопределяется метод `doInBackground()`, чтобы выполнять продолжительную задачу и периодически вызывать метод `publish()` для отображения хода ее выполнения. Этот метод выполняется в рабочем потоке. А метод `publish()`, в свою очередь, вызывает метод `process()`, чтобы вынудить поток диспетчеризации событий обработать данные о ходе выполнения задачи. По завершении задачи в потоке диспетчеризации событий вызывается метод `done()`, производящий завершающее обновление пользовательского интерфейса.

Всякий раз, когда требуется выполнить какую-нибудь задачу в рабочем потоке, следует сконструировать новый экземпляр объекта типа `SwingWorker`. (Каждый такой рабочий объект предназначен для однократного применения.) Затем вызывается метод `execute()`. Обычно этот метод вызывается в потоке диспетчеризации событий, но делать этого не рекомендуется.

Предполагается, что рабочий объект производит некоторый результат, и поэтому класс `SwingWorker<T, V>` реализует интерфейс `Future<T>`. Этот результат может быть получен с помощью метода `get()` из интерфейса `Future`. Метод `get()` устанавливает блокировку до тех пор, пока не будет доступен результат, поэтому вряд стоит вызывать его сразу же после метода `execute()`. Это лучше сделать, когда уже известно, что задача завершена. Обычно метод `get()` вызывается из метода `done()`. (Вызывать метод `get()` необязательно. Иногда достаточно и обработки данных о ходе выполнения задачи.)

Как промежуточные данные хода выполнения задачи, так и конечный результат могут иметь произвольные типы. Класс `SwingWorker` получает эти типы через параметры типа. Так, класс `SwingWorker<T, V>` выдает результат типа `T` и данные о ходе выполнения типа `V`. Чтобы прервать выполнение задачи, достаточно вызвать метод `cancel()` из интерфейса `Future`. Как только задача будет отменена, метод `get()` генерирует исключение типа `CancellationException`.

Как упоминалось выше, вызов метода `publish()` из рабочего потока повлечет за собой вызов метода `process()` в потоке диспетчеризации событий. Ради повышения эффективности результаты нескольких вызовов метода `publish()` могут стать причиной только одного вызова метода `process()`. Метод `process()` принимает в качестве параметра список типа `List<V>`, содержащий все промежуточные результаты.

Обратимся к практическому применению этого механизма на примере чтения из текстового файла. Оказывается, что компонент `JTextArea` достаточно медленный. Так, для ввода строк из длинного текстового файла (вроде романа “Графа Монте-Кристо”) потребуется немало времени. Чтобы показать пользователю ход выполнения данного процесса, требуется отображать в строке состояния количество прочитанных текстовых строк. Таким образом, данные о ходе чтения из текстового файла будут состоять из текущего номера строки и самой текстовой строки. Эти данные можно упаковать в тривиальный внутренний класс следующим образом:

```
private class ProgressData
{
    public int number;
    public String line;
}
```

В конечном итоге получается текст, прочитанный и сохраненный в объекте типа `StringBuilder`. А для его обработки в рабочем потоке понадобится класс `SwingWorker<StringBuilder, ProgressData>`.

В методе `doInBackground()` осуществляется построчное чтение из текстового файла. После каждой прочитанной текстовой строки вызывается метод `publish()` для передачи номера прочитанной строки и ее содержимого, как показано ниже.

```
@Override public StringBuilder doInBackground() throws IOException,
InterruptedException
{
    int lineNumber = 0;
    Scanner in = new Scanner(new FileInputStream(file));
    while (in.hasNextLine())
    {
        String line = in.nextLine();
        lineNumber++;
        text.append(line);
        text.append("\n");
        ProgressData data = new ProgressData();
        data.number = lineNumber;
        data.line = line;
        publish(data);
        Thread.sleep(1); // только для проверки отмены; а в
                        // конкретных программах не требуется
    }
    return text;
}
```

После каждой прочитанной строки чтение из текстового файла приостанавливается на одну миллисекунду, чтобы благополучно проверить возможность отмены, но выполнение конкретных прикладных программ вряд ли потребует замедлять подобного рода задержками. Если закомментировать строку кода `Thread.sleep(1);`, то обнаружится, что текст романа “Графа Монте-Кристо” загружается достаточно быстро — всего за несколько пакетных обновлений пользовательского интерфейса.



НА ЗАМЕТКУ! Выполнение рассматриваемой здесь программы можно сделать более плавным, если обновлять текстовую область из рабочего потока, но такое не допускается в большинстве других компонентов Swing. Здесь демонстрируется общий подход, когда все обновления компонента происходят в потоке диспетчеризации событий.

В методе `process()` игнорируются все номера строк, за исключением последней, а все прочитанные текстовые строки сцепляются для единого обновления текстовой области, как показано ниже.

```
@Override public void process(List<ProgressData> data)
{
    if (isCancelled()) return;
    StringBuilder b = new StringBuilder();
    statusLine.setText("" + data.get(data.size() - 1).number);
    for (ProgressData d : data) { b.append(d.line); b.append("\n"); }
    textArea.append(b.toString());
}
```

В методе `done()` текстовая область обновляется полным текстом, а пункт меню `Cancel` становится недоступным. Следует также иметь в виду, что рабочий поток запускается в приемнике событий от пункта меню `Open`. Описанная выше простая методика позволяет выполнять продолжительные задачи, сохраняя для ГПИ возможность реагировать на действия пользователя.

Листинг 14.15. Исходный код из файла `swingWorker/SwingWorkerTest.java`

```
1 package swingWorker;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.util.*;
7 import java.util.List;
8 import java.util.concurrent.*;
9
10 import javax.swing.*;
11
12 /**
13  * В этой программе демонстрируется рабочий поток, в котором
14  * выполняется потенциально продолжительная задача
15  * @version 1.1 2007-05-18
16  * @author Cay Horstmann
17  */
18 public class SwingWorkerTest
19 {
20     public static void main(String[] args) throws Exception
21     {
22         EventQueue.invokeLater(new Runnable()
23         {
24             public void run()
25             {
26                 JFrame frame = new SwingWorkerFrame();
27                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28                 frame.setVisible(true);
29             }
30         });
31     }
```

```
32 }
33
34 /**
35  * Этот фрейм содержит текстовую область для отображения содержимого
36  * текстового файла, меню для открытия файла и отмены его открытия, а
37  * также строку состояния для отображения процесса загрузки файла
38  */
39 class SwingWorkerFrame extends JFrame
40 {
41     private JFileChooser chooser;
42     private JTextArea textArea;
43     private JLabel statusLine;
44     private JMenuItem openItem;
45     private JMenuItem cancelItem;
46     private SwingWorker<String, ProgressData> textReader;
47     public static final int TEXT_ROWS = 20;
48     public static final int TEXT_COLUMNS = 60;
49
50     public SwingWorkerFrame()
51     {
52         chooser = new JFileChooser();
53         chooser.setCurrentDirectory(new File("."));
54
55         textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
56         add(new JScrollPane(textArea));
57
58         statusLine = new JLabel(" ");
59         add(statusLine, BorderLayout.SOUTH);
60
61         JMenuBar menuBar = new JMenuBar();
62         setJMenuBar(menuBar);
63         JMenu menu = new JMenu("File");
64         menuBar.add(menu);
65
66         openItem = new JMenuItem("Open");
67         menu.add(openItem);
68
69         openItem.addActionListener(new ActionListener()
70         {
71             public void actionPerformed(ActionEvent event)
72             {
73                 // показать диалоговое окно для выбора файлов
74                 int result = chooser.showOpenDialog(null);
75
76                 // если файл выбран, выбрать его в качестве
77                 // пиктограммы для метки
78                 if (result == JFileChooser.APPROVE_OPTION)
79                 {
80                     textArea.setText("");
81                     openItem.setEnabled(false);
82                     textReader = new TextReader(chooser.getSelectedFile());
83                     textReader.execute();
84                     cancelItem.setEnabled(true);
85                 }
86             }
87         });
88
89         cancelItem = new JMenuItem("Cancel");
90         menu.add(cancelItem);
91         cancelItem.setEnabled(false);

```

```
92     cancelItem.addActionListener(new ActionListener()
93     {
94         public void actionPerformed(ActionEvent event)
95         {
96             textReader.cancel(true);
97         }
98     });
99     pack();
100 }
101
102 private class ProgressData
103 {
104     public int number;
105     public String line;
106 }
107 private class TextReader extends SwingWorker<StringBuilder, ProgressData>
108 {
109     private File file;
110     private StringBuilder text = new StringBuilder();
111
112     public TextReader(File file)
113     {
114         this.file = file;
115     }
116
117     // Следующий метод выполняется в рабочем потоке,
118     // не затрагивая компоненты Swing
119
120     @Override
121     public StringBuilder doInBackground()
122         throws IOException, InterruptedException
123     {
124         int lineNumber = 0;
125         try (Scanner in = new Scanner(new FileInputStream(file)))
126         {
127             while (in.hasNextLine())
128             {
129                 String line = in.nextLine();
130                 lineNumber++;
131                 text.append(line);
132                 text.append("\n");
133                 ProgressData data = new ProgressData();
134                 data.number = lineNumber;
135                 data.line = line;
136                 publish(data);
137                 Thread.sleep(1); // только для проверки отмены; а в
138                                 // конкретных программах не требуется
139             }
140         }
141         return text;
142     }
143
144     // Следующие методы выполняются в потоке диспетчеризации событий
145
146     @Override
147     public void process(List<ProgressData> data)
148     {
149         if (isCancelled()) return;
150         StringBuilder b = new StringBuilder();
151         statusLine.setText("" + data.get(data.size() - 1).number);
152     }
```

```
153     for (ProgressData d : data)
154     {
155         b.append(d.line);
156         b.append("\n");
157     }
158     textArea.append(b.toString());
159 }
160 @Override
161 public void done()
162 {
163     try
164     {
165         StringBuilder result = get();
166         textArea.setText(result.toString());
167         statusLine.setText("Done");
168     }
169     catch (InterruptedException ex)
170     {
171     }
172     catch (CancellationException ex)
173     {
174         textArea.setText("");
175         statusLine.setText("Cancelled");
176     }
177     catch (ExecutionException ex)
178     {
179         statusLine.setText("" + ex.getCause());
180     }
181
182     cancelItem.setEnabled(false);
183     openItem.setEnabled(true);
184 }
185 };
186 }
```

javax.swing.SwingWorker<T, V> 6

- `abstract T doInBackground()`
Этот метод переопределяется, чтобы выполнять фоновую задачу и возвращать результат ее выполнения.
- `void process(List<V> data)`
Этот метод переопределяется, чтобы обрабатывать промежуточные данные о ходе выполнения задачи в потоке диспетчеризации событий.
- `void publish(V... data)`
Направляет промежуточные данные о ходе выполнения задачи в поток диспетчеризации событий. Вызывается из метода `doInBackground()`.
- `void execute()`
Планирует запуск данного рабочего задания в рабочем потоке.
- `SwingWorker.StateValue getState()`
Получает состояние рабочего потока, которое может принимать одно из значений: `PENDING`, `STARTED` или `DONE`.

Правило единственного потока

Каждая прикладная программа на Java начинается с метода `main()`, который выполняется в главном потоке. А в Swing-программе главный поток действует кратковременно. Он планирует построение ГПИ в потоке диспетчеризации событий и завершает свою работу. После построения ГПИ в потоке диспетчеризации событий обрабатываются уведомления о событиях вроде вызовов метода `actionPerformed()` или `paintComponent()`. А другие потоки вроде потока, направляющего события в очередь, действуют подспудно и поэтому недоступны для прикладного программирования.

Ранее в этой главе упоминалось следующее правило единственного потока: “не затрагивать компоненты Swing из любого потока, кроме потока диспетчеризации событий”. А теперь рассмотрим это правило подробнее. Существует несколько исключений из этого правила.

- В любом потоке можно совершенно безопасно вводить и удалять приемники событий. Разумеется, методы приемников событий будут вызываться из потока диспетчеризации событий.
- Лишь небольшое количество методов в Swing являются потокобезопасными. Они специально помечены в документации на прикладной интерфейс API следующим образом: “This method is thread safe, although most Swing methods are not” (Этот метод потокобезопасный, хотя большинство методов Swing таковыми не являются). Ниже перечислены наиболее полезные из потокобезопасных методов.

```
JTextComponent.setText()  
JTextArea.insert()  
JTextArea.append()  
JTextArea.replaceRange()  
JComponent.repaint()  
JComponent.revalidate()
```



НА ЗАМЕТКУ! В примерах программ, представленных в этой книге, неоднократно использовался метод `repaint()`, тогда как метод `revalidate()` применяется реже. Его назначение — инициировать компоновку компонента после того, как изменилось содержимое. В состав традиционной библиотеки AWT входит метод `validate()` для инициирования компоновки отдельного компонента. А для компонентов Swing вместо него следует вызывать метод `revalidate()`. (Но для того чтобы инициировать компоновку такого компонента, как `JFrame`, придется вызывать метод `validate()`, потому что класс `JFrame` является производным от класса `Component`, а не `JComponent`.)

Раньше правило единственного потока было более либеральным. В любом потоке можно было конструировать компоненты, устанавливать их свойства и вводить в контейнеры до тех пор, пока ни один из компонентов не был *реализован*. Компонент считается реализованным, когда он начинает принимать события перерисовки и проверки достоверности. Это происходит, как только выполняется вызов метода `setVisible(true)` или `pack()` для этого компонента, или же когда компонент вводится в уже реализованный контейнер.

Такой вариант правила единственного потока был удобным. Ведь он позволял строить ГПИ в методе `main()` и затем вызывать метод `setVisible(true)` во фрейме верхнего уровня прикладной программы. И не было никакой необходимости в трудоемком планировании исполняемого потока типа `Runnable` в потоке диспетчеризации событий.

К сожалению, некоторые реализаторы компонентов не уделили должного внимания особенностям исходного правила единственного потока. Они запускали действия в потоке диспетчеризации событий, не заботясь даже о проверке того, реализован ли компонент. Так, если вызывается метод `setSelectionStart()` или `setSelectionEnd()` для компонента типа `JTextComponent`, то перемещения знака вставки планируются в потоке диспетчеризации событий, даже если компонент невидим.

Обнаружить и исправить эти недостатки совсем не трудно, но разработчики Swing пошли по пути наименьшего сопротивления. Они объявили, что обращаться к компонентам из любого потока, который не является потоком диспетчеризации событий, небезопасно. А поэтому ГПИ приходится строить в потоке диспетчеризации событий, делая вызов метода `EventQueue.invokeLater()`, как уже не раз демонстрировалось в рассмотренных ранее примерах программ.

Разумеется, существует немало программ, в которых по неосторожности соблюдается старый вариант правила единственного потока, когда ГПИ инициализируется в главном потоке. Такие программы подвержены определенному риску, связанному с тем, что некоторые действия по инициализации ГПИ могут вступить в конфликт с действиями в главном потоке. Как упоминалось в главе 7, вы вряд ли захотите оказаться одним из тех немногих "счастливых", которые столкнутся с подобными неприятностями и будут вынуждены тратить время и силы на отладку нерегулярно воспроизводящихся ошибок, связанных с потоками. Поэтому лучше просто строго следовать правилу единственного потока в его нынешнем варианте.

На этом первый том данной книги завершается. Он был посвящен основам языка программирования Java и тем компонентам стандартной библиотеки, которые требуются в большинстве программных проектов. Надеемся, что этот экскурс в основы Java оказался для вас интересным и вы извлекли из него немало полезного для себя. А более сложные вопросы, в том числе сетевое программирование, расширенные средства библиотек AWT и Swing, безопасность и интернационализация, будут обсуждаться во втором томе.

Ключевые слова Java

Ключевое слово	Что означает	См. главу
<code>abstract</code>	Абстрактный класс или метод	5
<code>assert</code>	Поиск внутренних ошибок в программе	11
<code>boolean</code>	Логический тип данных	3
<code>break</code>	Прерывание оператора <code>switch</code> или цикла	3
<code>byte</code>	8-разрядный целочисленный тип	3
<code>case</code>	Ветвь перехода в операторе <code>switch</code>	3
<code>catch</code>	Оператор блока <code>try</code> для перехвата исключений	11
<code>char</code>	Символьный тип в уникоде	3
<code>class</code>	Тип класса	4
<code>const</code>	Не используется	
<code>continue</code>	Продолжение выполнения кода в конце цикла	3
<code>default</code>	Ветвь перехода по умолчанию в операторе <code>switch</code>	3
<code>do</code>	Начало цикла <code>do-while</code>	3
<code>double</code>	Числовой тип данных с плавающей точкой двойной точности	3
<code>else</code>	Альтернативная ветвь условного оператора <code>if</code>	3
<code>enum</code>	Перечислимый тип	3
<code>extends</code>	Родительский класс для данного класса	4
<code>final</code>	Константа, класс или метод, который нельзя переопределить	5
<code>finally</code>	Часть блока <code>try</code> , которая выполняется всегда	11
<code>float</code>	Числовой тип данных с плавающей точкой одинарной точности	3
<code>for</code>	Разновидность цикла	3
<code>goto</code>	Не используется	
<code>if</code>	Условный оператор	3

Окончание таблицы

Ключевое слово	Что означает	См. главу
implements	Один или более интерфейс, реализуемый в классе	6
import	Импорт пакета	4
instanceof	Проверка, является ли объект экземпляром класса	5
int	32-разрядный целочисленный тип	3
interface	Абстрактный тип с методами, который может быть реализован в классе	6
long	64-разрядный длинный целочисленный тип	3
native	Метод, реализуемый на уровне операционной системы	11 (второй том)
new	Выделение памяти для нового объекта или массива	3
null	Пустая ссылка (формально null является константой, а не ключевым словом)	3
package	Пакет классов	4
private	Модификатор доступа только для методов данного класса	4
protected	Модификатор доступа только для методов данного класса, производных от него классов и других классов из того же самого пакета	5
public	Модификатор открытого доступа для методов всех классов	4
return	Возврат из метода	3
short	16-разрядный короткий целочисленный тип	3
static	Модификатор однозначности компонентов класса, а не его объектов	4
strictfp	Строгие правила для вычислений с плавающей точкой	3
super	Объект или конструктор суперкласса	5
switch	Оператор переключения	3
synchronized	Метод или блок кода, атомарный для потока	14
this	Неявный аргумент метода или конструктора класса	4
throw	Генерирование исключения	11
throws	Исключения, которые может генерировать метод	11
transient	Обозначение данных, которые не должны быть постоянными	1 (второй том)
try	Блок кода, в котором могут возникнуть обрабатываемые исключения	11
void	Обозначение метода, не возвращающего никаких значений	3
volatile	Указание на то, что поле может быть доступно из нескольких потоков	14
while	Разновидность цикла	3

Предметный указатель

А

- Автоупаковка и распаковка примитивных типов, 240
- Алгоритмы
 - решето Эратосфена, реализация, 750
 - сжатия данных, 526
- Аплеты
 - взаимодействие, 564
 - выполнение
 - в браузере, 29; 54
 - в утилите appletviewer, 53
 - иерархия наследования классов, 549
 - история развития, 549
 - контекст, 564
 - назначение, 28; 525
 - обращение
 - к звуковым файлам, 563
 - к файлам изображений, 563
 - передача данных через параметры, 557
 - построение и выполнение, 53
 - преобразование приложений, 553
 - применение, 549
 - реализация средствами Swing, 549
- Архитектура вилочного соединения
 - алгоритм перехвата работы, 829
 - назначение, 828
- Атомарность операций, обеспечение, 796

Б

- Библиотеки
 - AWT
 - иерархия событий, 392
 - компоненты для обработки событий, 394
 - механизм обработки событий, 360
 - назначение, 314
 - недостатки, 314
 - IFC
 - разработка, 314
 - состав, 315
 - Java 2D, назначение, 334
 - Swing
 - архитектура компонентов, 398
 - визуальные стили ГПИ, 315
 - выполнение компонентов в потоке диспетчеризации событий, 320
 - исключения из правила единственного потока, 846
 - назначение, 315
 - отличия от AWT, 315
 - потокбезопасные методы, 846
 - правила многопоточной обработки, 835
 - преимущества, 315
 - разработка, 314
 - коллекций Java
 - архитектура классов коллекций, 725
 - итераторы, особенности, 688
 - классы, 692

разделение интерфейсов и реализации, 684

рефлексии, 246

Битовые маски, назначение, 385

Блоки

вложенные, 97

инициализации, 175

назначение, 97

область действия, 97

операторов, 98

синхронизированные, 792

Большие числа

математические операции, 114

определение, 114

В

Ввод-вывод

консольный, 88

файловый, 95

форматирование выводимых данных, 90

чтение вводимых данных, 88

Взаимная блокировка

возникновение, 785

как явление, 796

условия возникновения, 798

Визуальные стили ГПИ

изменение, способы, 370

разновидности, 315

Встраивание кода, назначение, 210

Г

Глобальные параметры настройки

каталог для хранения, 567

прикладной интерфейс API для хранения, 572

сохранение и восстановление, 567

таблицы свойств, 567

центральное хранилище, 572

Д

Двухмерные формы

построение, 338

рисование, 334

Действия

активизация и запрет, 378

изменяющие цвет, внедрение, 379

назначение, 377

предопределенные имена, 378

привязка, 381

Дескрипторы

<applet>

атрибуты, 554

назначение, 554

<object>

атрибуты, 557

назначение, 557

<param>, назначение, 557

Диалоговые окна
 безрежимные
 для установки цвета фона, создание, 518
 назначение, 486
 для выбора разных вариантов
 вывод сообщений, 487
 компоненты, 486
 создание, 488
 типы вариантов, 487
 для выбора файлов
 вспомогательный компонент, 511
 создание, 507
 для обмена данными, создание, 501
 документно-модальные, назначение, 497
 инструментально-модальные,
 назначение, 497
 кнопка по умолчанию, установка, 502
 признак модальности, указание, 497
 режимные
 для выбора цвета, создание, 518
 назначение, 486
 создание, 496
 с текстовыми компонентами, компоновка
 в Swing GUI Builder, 470
 селектора шрифтов
 компоновка, 460
 порядок обхода компонентов, 485
 фрейм-владелец, указание, 497

Диспетчеры
 безопасности, назначение, 537
 компоновки
 граничной, 408
 групповой, 460; 470
 поточной, 405
 по умолчанию, 407
 сеточно-контейнерной, 460
 собственные, установка, 407
 специальные, создание, 481
 функции, 406
 протоколирования
 инициализация, 611
 смена, 611

Документация, создание и
 комментирование, 189

И

Изображения
 ввод из файлов, 354
 вывод
 в окне, 354
 рядами, 354

Импорт
 классов, 180
 статический, 181

Инициализация в двойных фигурных
 скобках, прием, 303

Инкапсуляция
 основной принцип, 133
 поля экземпляра и методы, 133
 преимущества, 156

Интегрированные среды разработки
 Eclipse
 выполнение апплетов, 55
 выявление ошибок компиляции, 50
 отладка программ, 640
 применение, 47

NetBeans
 групповая компоновка ГПИ, 460
 построитель ГПИ Swing GUI Builder, 470
 назначение, 44

Интерфейсы
 Action
 методы, 378
 реализация, 379
 расширение, 378

ActionListener
 метод actionPerformed(), реализация, 287;
 363
 объявление, 287
 отслеживаемые события,
 разновидности, 363
 реализация, 287; 361

AppletContext, реализация и методы, 564

AutoCloseable, реализация и метод, 596

ButtonModel
 методы, 427
 реализация, 403
 свойства, 403

Callable
 метод, 816; 820
 назначение, 816
 реализация, 817

ChangeListener, реализация и метод, 435

Cloneable
 назначение, 282
 реализация, 283

Collection
 методы, 686; 691
 обобщенные служебные методы, 689
 расширение, 687

Comparable
 метод compareTo(), 274
 обобщение, 272
 определение, 272
 реализация, 708

Comparator, реализация и методы, 708

Condition, методы, 788; 802

Deque, реализация и методы, 714

Enumeration, реализация и методы, 746

ExecutorService
 методы, 825; 827
 реализация, 822

Filter, реализация и метод, 616

Future
 методы, 817; 820
 назначение, 817
 реализация, 817

InvocationHandler
 метод обработки вызовов, 308
 реализация, 308

Iterable, реализация и метод, 687

Iterator
 методы, 687; 692
 обобщенные служебные методы, 689
 реализация, 687

LayoutManager, реализация и методы, 481

List, реализация и методы, 700; 702; 727

ListIterator, реализация и методы, 695; 701

Lock, методы, 783; 788; 801

Map
 методы, 719
 реализация, 716

- MenuItemListener, методы, 451
 - MouseListener, реализация и методы, 388
 - MouseMotionListener, реализация и методы, 387
 - NavigableMap, реализация и методы, 728; 736
 - NavigableSet, реализация и методы, 710; 712; 731; 735
 - Queue
 - методы, 713
 - реализация, способы, 685
 - RandomAccess, назначение и реализация, 727
 - Runnable
 - методы, 764
 - реализация, 760; 817
 - ScheduledExecutorService
 - методы, 826
 - реализация, 826
 - Set, реализация и методы, 727
 - Shape, реализация, 335
 - SortedMap
 - методы, 736
 - назначение, 727
 - SortedSet
 - методы, 712; 731; 735
 - назначение, 727
 - SwingConstants
 - константы, 416
 - реализация, 417
 - Thread.UncaughtExceptionHandler
 - реализация и метод, 772
 - TransferQueue
 - методы, 812
 - реализация, 807
 - Type, иерархия наследования, 677
 - WindowListener
 - методы, 373
 - реализация, 373
 - для отслеживания и обработки событий, 394
 - коллекций
 - применение, 738
 - разделение, 684
 - константы, объявление, 278
 - назначение, 272
 - очередей, 684
 - помеченные, назначение, 283
 - порядок реализации, 273
 - приемников событий, 360; 394
- Исключения**
- возникновение, 583
 - генерирование
 - объявление, 584
 - повторное, 591
 - порядок, 586
 - условия, 249; 584
 - заключение в оболочку, 592
 - классификация, 582
 - необработываемые, обработчики, 772
 - непроверяемые
 - определение, 249; 583
 - отличие от проверяемых, 602
 - обработка
 - механизм, назначение, 581
 - организация, 249
 - рекомендации, 600
 - описание, 584
 - освобождение ресурсов в блоке finally, 592
 - передача на обработку, 603
 - перехват
 - в блоке операторов try/catch, 588
 - порядок, 588
 - проверяемые
 - обработка, 602
 - объявление, 583
 - определение, 249; 583
 - преодоление ограничений на обработку, 663
 - собственных типов, генерирование, 587
 - цепочки, 591
- Исполнители**
- организация пулов потоков, 821
 - управление группами задач, 826
- К**
- Клавиши**
- быстрого доступа, задание, 448
 - оперативные, задание, 449
- Классы**
- AbstractAction, назначение и методы, 379
 - AbstractCollection
 - методы абстрактные и служебные, 690
 - расширение, 690
 - реализуемые методы, 698
 - Applet, методы, 553; 562; 563
 - Array
 - методы, 261
 - назначение, 260
 - ArrayDeque, конструкторы, 715
 - ArrayList
 - методы, 233
 - назначение, 232
 - применение, 702
 - Arrays, методы, 119; 121; 260; 735
 - BasicButtonUI, представление кнопок, 404
 - BigDecimal, применение, 114
 - BigInteger, применение, 114
 - BorderFactory, конструкторы и методы, 430
 - BorderLayout, константы и методы, 408
 - BoxLayout, назначение, 459
 - ButtonUIListener, контроллер кнопок, 404
 - Class
 - методы, 247; 675; 681
 - назначение, 246
 - обобщение, 675
 - Collections, методы, 731; 734; 741; 743
 - Color
 - константы для выбора цвета, 343
 - конструктор, 343
 - методы, 344
 - Component
 - иерархия наследования, 406
 - методы, 322
 - ConcurrentHashMap
 - конструкторы, 814
 - методы, 813
 - ConcurrentSkipListMap, методы, 813
 - ConcurrentSkipListSet
 - конструкторы, 815
 - методы, 815
 - Console, применение, 89

- Constructor
 - метод, 676
 - обобщение, 675
- Container, назначение и расширение, 406
- Date
 - методы, 138; 141
 - назначение, 137
- DefaultButtonModel, назначение, 403
- Employee
 - анализ реализации, 152
 - методы, 151
 - наследование, 198
 - поля и методы, 155
 - создание, 149
- EnumMap
 - конструктор, 725
 - назначение, 723
- EnumSet
 - методы, 724
 - назначение, 723
- Error, иерархия наследования, 582
- EventHandler
 - методы create(), разновидности, 369
 - создание обработчиков событий, 369
- EventObject, назначение, 392
- EventQueue, методы, 839
- Exception, иерархия наследования, 582
- ExecutorCompletionService
 - конструктор и методы, 828
 - назначение, 827
- Executors
 - исполнители пулов потоков, 821
 - методы, 826
 - фабричные методы, 821
- FileFilter
 - методы, 517
 - фильтры файлов, 509
- FileHandler, конструкторы, 624
- FileView
 - методы, 510; 517
 - расширение, 510
- Font
 - задание параметров шрифта в конструкторе, 347
 - методы, 348
 - объекты для выделения шрифтами, 347
- Formatter, методы, 625
- FutureTask
 - конструкторы, 821
 - назначение, 817
- Graphics
 - методы, 331; 354
 - объекты для рисования графики, 330
- Graphics2D
 - методы, 335; 343
 - объекты для рисования двумерных форм, 334
- GregorianCalendar
 - конструкторы, 141
 - методы, 142
 - назначение, 141
- GridBagConstraints
 - конструктор, 470
 - параметры компоновки, 469
- GridLayout, методы, 413
- GridLayout, конструктор и методы, 478
- GroupLayout.Group, методы, 479
- Handler, методы, 623
- HashMap
 - конструкторы, 720
 - назначение, 716
- HashSet
 - конструкторы, 706
 - методы, 704
 - назначение, 704
- IdentityHashMap
 - конструкторы, 725
 - назначение, 723
- InputMap, назначение, 381
- Integer
 - методы, 241
 - объектной оболочки типа int, 239
- JApplet, расширение, 549
- JButton
 - как оболочка, 404
 - обращение к модели кнопки, 404
- JCheckBox, конструкторы и методы, 423
- JCheckBoxMenuItem, конструкторы, 446
- JColorChooser
 - конструктор и методы, 523
 - назначение, 518
- JComboBox
 - методы, 432
 - назначение, 432
 - обобщение, 432
- JComponent
 - методы, 415
 - переопределение метода paintComponent(), 330
 - привязки ввода, 380
 - расширение, 330
- JDialog, конструктор, 500
- JFileChooser
 - конструктор, 516
 - методы, 507; 516
 - назначение, 507
- JFrame
 - иерархия наследования, 322
 - методы, 322
 - назначение, 319
 - структура, 329
- JLabel
 - конструкторы, 417
 - методы, 417
 - параметры конструктора, 416
- JMenu, конструктор и методы, 443
- JMenuItem, конструкторы и методы, 443; 450
- JOptionPane
 - методы, 486; 494
 - назначение, 486
- JPasswordField
 - конструктор и методы, 418
 - назначение, 418
- JPopupMenu, методы, 447
- JRadioButtonMenuItem, конструкторы, 446
- JRadioButton, конструкторы и методы, 427
- JScrollPane, конструктор, 421
- JSlider, конструкторы и методы, 440
- JTextArea, конструкторы и методы, 421
- JTextComponent
 - методы, 414
 - назначение, 414

- TextField
 - конструктор, задание параметров, 414
 - методы, 415
- JToolBar, конструкторы и методы, 458
- KeyStroke, назначение, 380
- LineBorder, конструктор, 431
- LinkedHashMap
 - конструкторы и методы, 724
 - назначение, 721
- LinkedHashSet
 - конструкторы, 724
 - назначение, 721
- LinkedList
 - конструкторы, 702
 - методы, 695; 702
 - назначение, 694
- Logger, методы, 621
- LogRecord, методы, 624
- Manager
 - конструкторы, 201
 - определение, 198
 - поля и методы, 198
- Math
 - инкапсуляция только функциональных возможностей, 137
 - константы, 73
 - математические функции, 72
 - методы, 72
- Method
 - методы, 681
 - назначение, 250
- Object
 - методы, 219; 226; 280; 792
 - назначение, 218
- Pair
 - методы, 649
 - назначение, 648
 - объявление переменных типа, 648
- PasswordChooser, назначение, 502
- Point2D, назначение, 337
- Preferences
 - методы, 573; 577
 - реализация центрального хранилища, 572
- PriorityQueue, конструкторы, 716
- Properties
 - конструкторы, 570; 748
 - методы, 567; 570; 748
 - реализация таблиц свойств, 567
- Proxy
 - методы, 308
 - расширение прокси-классами, 311
- Rectangle2D
 - иерархия наследования, 336
 - методы, 337
- RectangularShape
 - иерархия наследования, 337
 - методы, 338
- ReentrantLock
 - защита блока кода, 780
 - конструкторы, 783
- ReentrantReadWriteLock
 - методы, 803
 - назначение, 802
- Robot
 - методы, 634; 638
 - назначение, 634
- RuntimeException, исключения и ошибки, 582
- Scanner, методы, 88
- SoftBevelBorder, конструкторы и методы, 431
- StackTraceElement, методы, 597; 600
- StrictMath, математические функции, 73
- String
 - для символьных строк, 77
 - методы, 81
- StringBuilder
 - методы, 87
 - применение, 86
- SwingWorker
 - методы, 840; 845
 - назначение, 840
- System, методы, 571
- SystemColor, обозначение системных цветов, 344
- Thread
 - конструкторы, 764
 - методы, 764; 767; 769; 771; 773
 - не рекомендованные к применению
 - методы, 803
 - расширение, 761
- ThreadGroup, назначение, 773
- ThreadLocal
 - методы, 800
 - назначение, 799
- ThreadLocalRandom, назначение и метод, 800
- ThreadPoolExecutor
 - методы, 825
 - назначение, 822
- Throwable
 - иерархия наследования, 582
 - конструкторы и методы, 588; 599
- Timer
 - конструктор, применение, 288
 - назначение, 287
- Toolkit, методы, 325; 387
- TreeMap
 - конструкторы, 720
 - назначение, 716
- TreeSet
 - конструкторы, 707; 713
 - назначение, 706
- WeakHashMap
 - конструкторы, 724
 - назначение, 720
- Window, методы, 322
- WindowStateListener, назначение и методы, 373
- абстрактные, 212
- адаптеров
 - для обработки событий, 394
 - назначение, 374
 - расширение, 374
 - создание, 374
- внутренние
 - анонимные, особенности, 301
 - локальные, особенности, 298
 - назначение, 289
 - синтаксис внешних ссылок, 294
 - ссылки на внешние классы, 292
 - статические, особенности, 304
 - упрощение исходного кода, 366
- для рисования двухмерных форм, 334; 338

- идентификация, 134
- импорт, 180
- исключений
 - иерархия наследования, 582
 - создание, 587
- коллекций
 - абстрактные, 686; 728
 - из библиотеки Java, 692
 - конкретные, 728
 - унаследованные, 683; 729; 746; 749
- конечные, 209
- назначение, 58
- наследование, 198
- неизменяемые, 158
- низкоуровневых событий, 394
- обобщенные
 - определение, 648
 - правила наследования типов, 666
 - применение, 647
 - реализация, трудности, 647
- объектных оболочек, разновидности, 239
- определение, 133
- отношения, 135
- перечислений, назначение, 244
- получение экземпляров, 133
- порядок именованья, 58
- потокбезопасные, синхронизирующая оболочка, 815
- пути
 - определение, 187
 - указание, 188
- размещение в отдельных файлах, 151
- расширение, 134
- рекомендации по разработке, 194
- семантических событий, 393
- синхронизаторов, 831
- создание, 148
- суперклассы и подклассы, 198
- форма определения, 149
- Ключевые слова
 - abstract, 213
 - assert, 604
 - class, 58
 - final, 209
 - implements, 273
 - import, 180
 - protected, 218
 - static, 161
 - strictfp, 70
 - super, 201
 - synchronized, 789
 - this, 154; 174; 201
 - volatile, 795
 - языка Java, перечень, 849
- Кнопки
 - анализ по шаблону MVC, 403
 - компоновка на панели, 405
 - модель, 403
- Кнопки-переключатели
 - внешние отличия от флажков, 424
 - группы, 424
 - компоновка, 424
- Коллекции
 - алгоритмы
 - двоичного поиска, 742
 - простые, 743
 - реализация, 738
 - собственные, написание, 744
 - сортировки и перетасовки, 739
- архитектура
 - внутреннее строение, 726
 - назначение и состав, 725
 - удобство конструкции, 691
- выбор реализации, 686
- групповые операции, 736
- и массивы, взаимное преобразование, 737
- интерфейсы, 686
- итераторы
 - особенности, 688
 - применение, 687
 - легковесные оболочки, применение, 730
 - необязательные операции, методика, 734
 - ограниченные, 686
 - отсортированные, 706
 - поиск элементов, 703
 - построение, 686
 - потокбезопасные
 - поддержка читающих и записывающих потоков, 813
 - разновидности, 813
 - слабо совместные итераторы, возврат, 813
 - устаревшие, 815
- представления
 - немодифицируемые, получение, 731
 - поддиапазонов, формирование, 730
 - применение, 729
 - проверяемые, назначение, 733
 - синхронизированные, назначение, 732
- удаление элементов, 688
- упорядоченные, 726
- перебор элементов, 688
- Командная строка
 - выполнение графического приложения, 51
 - компиляция и запуск программ, 44
 - построение и выполнение апплетов, 53
 - режим работы, особенности, 46
- Комментарии
 - гипертекстовые ссылки, 193
 - дескрипторы, 191; 192
 - извлечение, 194
 - к классам, 190
 - к методам, 191
 - к пакетам, 193
 - к полям, 192
 - обзорные, 193
 - общего характера, 192
 - порядок составления, 190
 - способы выделения в коде, 61
- Компиляторы
 - Java, порядок вызова, 44
 - динамические, 27; 210
 - традиционные, 27
- Комплект JDK
 - библиотеки и документация, установка, 41
 - загрузка, 38
 - задание пути к исполняемому файлу, 39
 - оперативно доступная документация, 84
 - установка, 39
- Комплекты ресурсов
 - ввод сопоставлений, 612
 - включение в состав программ, 612
 - назначение, 612

Компоненты ГПИ

- абсолютное расположение, 480
- автоматическая перерисовка, 331
- блочная компоновка, 459
- граничная компоновка, 408
- групповая компоновка, 460
- для ввода текста, 414
- для выбора разных вариантов, 421
- классы-оболочки, назначение, 401
- компоновка без диспетчера, 480
- меню, 441
- панели
 - инструментов, 455
 - прокрутки, 419
- пометка, 416
- порядок обхода, 484
- поточная компоновка, 405
- пружинная компоновка, 460
- рамки, задание, 427
- расположение в контейнерах, 406
- регулируемые ползунки, 435
- сеточная компоновка, 410
- составляющие и свойства, 399
- диалоговые окна, 485
- для рисования графики, создание, 330
- комбинированные списки, 432
- сеточно-контейнерная компоновка
 - весовые поля, 462
 - внешнее и внутреннее заполнение, параметры, 463
 - заполнение и привязка компонентов, параметры, 463
 - наложение ограничений, вспомогательный класс, 465
 - описание, 461
 - особенности, 459
 - принцип действия, 460
 - расположение компонентов, параметры, 462
 - рекомендации, 463

Константы

- класса, 69
- обозначение, 68
- статические, 160

Конструкторы

- без аргументов, 172
- вызов
 - одних из других, 174
 - порядок действий, 175
- именование параметров, 173
- локальные переменные, 154
- назначение, 137
- особенности, 153
- перегрузка, 171

Курсоры

- для Windows, виды, 386
- определение нового вида, 387

М**Манифест**

- главный класс, указание, 528
- назначение, 527
- разделы, 527
- редактирование, 527
- точка входа, указание, 528
- файл, 527

Массивы

- анонимные, 118
- доступ по индексу, 116
- инициализация, 118
- копирование, 119
- многомерные, 124
- неровные, 128
- объявление, 116
- определение, 116
- размер
 - выделение памяти, 234
 - задание, 232
- свойство length, 117
- сортировка, 121
- списочные
 - добавление элементов, 233
 - доступ к элементам, 235
 - емкость, 234
 - обобщенные, 232
 - применение, 702
 - создание, 232

Меню

- всплывающие
 - построение, 446
 - триггер, 447
- подменю и пункты, 441
- построение, 442
- пункты
 - разрешение и запрет доступа, 450
 - с кнопками-переключателями, 445
 - с пиктограммами, 444
 - с флажками, 445
- строка, 441

Метки

- назначение, 416
- размещение в контейнере, 417
- с пиктограммой и текстом надписи, задание, 417
- с текстом надписи, задание, 417

Методы

- clone()
 - применение, 283
 - реализация, 283
- equals()
 - назначение, 219
 - рекомендации по реализации, 222
 - характеристики, 221
- finalize(), применение, 179
- hashCode()
 - назначение, 224
 - усовершенствование, 225
- main()
 - назначение, 59
 - объявление, 60
 - особенности, 162
 - параметры командной строки, 120
- printf()
 - назначение, 91
 - переменное число параметров, 243
- println(), назначение, 60
- setVisible(), назначение и вызов, 501
- toString()
 - назначение, 226
 - обобщенные, реализация, 257
 - основания для реализации, 227
- абстрактные, 213

ВЫЗОВ
 обозначение, 60
 по значению, 164
 по имени, 164
 по ссылке, 164
доступа
 к полям, 155
 определение, 142
защищенные, 218
идентификация, 135
именование параметров, 174
ковариантные возвращаемые типы, 207
конечные, 209
модифицирующие, 142
мостовые, назначение, 656
обобщенные
 выводимость типов, 650
 вызов, 650
 определение, 650
 служебные, 690
 сопоставление типов, 676
определение, 60
открытые и закрытые, 158
параметры и аргументы, 61
перегрузка, 171
переопределение, 199
с переменным числом параметров, 243
сигнатура, 171; 207
синхронизированные, 790
статические, 160
тело, обозначение, 60
типы параметров, 166
фабричные, 161
явные и неявные параметры, 154
Множества
битовые, реализация, 749
древовидные
 ввод элементов, 707
 как отсортированные коллекции, 706
 сравнение объектов, 707
 структура красно-черного дерева, 707
на основе хеш-таблиц, 704
определение, 704
перечислимые, реализация, 723
Модификаторы доступа
final, 158; 209; 796
private, 158; 218
protected, 218
public, 158; 218
static, 159
volatile, 796
 назначение, 58
 отсутствие, 218
Мониторы
 принцип, 794
 свойства, 794

Н
Наследование
 иерархия и цепочки, 204
 как принцип ООП, 197
 обозначение, 198
 предотвращение, 209
 приведение типов, правила, 211
 признак, 198

применение, 198
 принцип подстановки, 205
 рекомендации по применению, 267

0

Обобщения
 захват подстановок, 673
 и виртуальная машина, 653
 накладываемые ограничения, 658–665
 неограниченные подстановки, 672
 ограничения супертипа на подстановки, 670
 параметры типа, 646
 подстановочные типы, 647; 668
 преобразование, особенности, 657
 стирание типов, 653
 экземпляры обобщенного типа, создание, 648
Обратный вызов, мезанизм, 286
Объектные оболочки
 классы, 239
 применение, 241
 примитивных типов, 239
Объекты
 блокировки, 781
 действий, 379
 исключений, 590
 клонирование, особенности, 282
 ключевые свойства, 134
 конструирование, 170
 копирование
 неполное, 281
 полное, 281
 назначение, 132
 присваивание переменным, 138
 событий, 360; 392
 создание экземпляров, 137
 уничтожение, 179
 условий, 783
 функциональные, 709
Окрашивание цветом
 выбор цвета, 343
 геометрических форм, 343
 установка цвета фона и переднего плана, 344
Операторы
break с меткой, применение, 111
continue
 применение, 113
 с меткой, 113
switch
 метки ветвей **case**, 109
 принцип действия, 108
throw, назначение, 586
throws, назначение, 584
try с ресурсами
 назначение, 596
 формы, 596
ромбовидные, 233
Операции
instanceof, 211
new, 139
 арифметические, 69
 инкрементирования и
 декрементирования, 70
 логические, 71
 отношения, 71
 поразрядные, 72

- приоритетность, 75
- сравнения, 71
- тернарные, 71
- Отладка программ
 - в среде JUnit, 627
 - рекомендации, 626
 - с ГПИ, рекомендации, 631
- Отладчики
 - Swing, графические, 631
 - автономные, применение, 639
 - в ИСР Eclipse, 640
 - точка прерывания, установка, 639
- Отображения
 - ввод элементов, 717
 - назначение, 716
 - перечислимые, реализация, 723
 - представления
 - получение, 717
 - разновидности, 717
 - применение, 718
 - разновидности реализации, 716
 - связные хеш-отображения, 722
 - слабые хеш-отображения, 721
 - удаление элементов, 717
 - хеш-отображения идентичности, построение, 723
- Очереди
 - блокирующие
 - классы, 807
 - методы, 805
 - назначение, 805
 - интерфейс, 684
 - односторонние и двусторонние, 713
 - по приоритету
 - организация в виде кучи, 715
 - применение, 715
 - применение, 684
 - принцип действия, 684
 - реализация, 685
 - синхронные, 833
- Ошибки
 - возврат кода или признака, 581
 - инкапсуляция в объектах, 581
 - порядок обработки, 580
 - причины появления, 580
 - разновидности, 581

П

Пакеты

- java.awt.event, 393
- java.swing.event, 394
- java.util, 392
- java.util.concurrent, 790; 807
- java.util.concurrent.atomic, 796
- java.util.concurrent.locks, 802
- javax.swing, 320
- javax.swing.filechooser, 510
- ввод классов, 182
- вложенные, 180
- герметизация, 186; 532
- назначение, 179
- область действия, 185
- по умолчанию, 182
- размещение, 182
- стандартные, 179

- Панели
 - инструментов
 - назначение, 454
 - обособленные, 455
 - перетаскивание, 455
 - построение, 456
 - прокрутки
 - ввод компонентов ГПИ, 419
 - назначение, 419
- Пароли
 - поля для ввода, 418
 - ввод с консоли, 89
- Перегрузка
 - назначение, 171
 - разрешение, 171; 206
- Переменные
 - инициализация, 68
 - локальные, 171
 - объектные
 - инициализация, 139
 - особенности, 138
 - ссылки на объекты, 139
 - объявление, 66
 - разделяемые, 796
 - типа
 - именование, 648
 - объявление, 648
 - ограничения, 651
 - стирание, 653
- Перечисления, реализация, 746
- Полиморфизм
 - как принцип ООП, 205
 - определение, 202
- Поля
 - для ввода пароля, компоновка, 418
 - защищенные, 218
 - инициализация
 - по умолчанию, 171
 - явная, 172
 - конечные, 209
 - неизменяемые, 158
 - открытые и закрытые, 155
 - разделяемые, 795
 - экземпляра, 133
 - статические, 159
 - текстовые
 - ввод и расположение, 414
 - задание размеров, 415
 - правка содержимого, 415
- Потоки
 - блокировка
 - встроенная, 789
 - клиентская, 793
 - по условию, 785
 - равноправная, особенности, 783
 - реентрабельная, 782
 - счетчик захватов, 782
 - установка и снятие, 782
 - чтения или записи, 802
 - блокированные, состояние, 768
 - временно ожидающие, состояние, 769
 - время ожидания для снятия блокировки, 801
 - группы, назначение, 773
 - диспетчеризации событий, назначение, 759
 - для выполнения отдельных задач, организация, 760

- завершенные, состояние, 769
 - исполняемые, состояние, 768
 - набор ожидания, 785
 - новые, состояние, 768
 - ожидающие, состояние, 769
 - определение, 753
 - отличие от процессов, 754
 - плановое выполнение, 826
 - превращение в потоковые демоны, 772
 - прерывание, 765
 - приоритеты, 771
 - пулы
 - назначение, 821
 - организация, 822
 - построение, 821
 - синхронизаторы
 - барьеры, 832
 - защелки с обратным отсчетом, 832
 - классы, 831
 - обменники, 833
 - семафоры, 831
 - синхронные очереди, 833
 - синхронизация
 - по принципу монитора, 794
 - путем блокировок и условий, 781
 - состояния, разновидности, 767
 - установка и проверка состояния прерывания, 765
 - Предусловие
 - наложение, 606
 - нарушение, 606
 - определение, 606
 - Приведение типов
 - определение, 75
 - при наследовании, 211
 - обозначение, 75
 - Привязки
 - ввода
 - назначение, 380
 - получение, 381
 - порядок проверки условий, 381
 - условия, 380
 - действий к компонентам ГПИ, 381
 - Примеры программ
 - загрузка, 17
 - установка, 42
 - Программирование
 - обобщенное
 - назначение, 646
 - реализация, способы, 646
 - уровни квалификации, 647
 - объектно-ориентированное
 - инкапсуляция, 133
 - наследование, 134; 197
 - основные понятия, 133
 - особенности, 132
 - полиморфизм, 205
 - структурное, особенности, 132
 - Прокси-классы
 - методы, 307
 - назначение, 307
 - прокси-объекты, создание и применение, 308
 - протоколирующие прокси-объекты, назначение, 627
 - свойства, 311
 - Протоколирование
 - вывод и хранение протокольных записей, 613
 - иерархия регистраторов, 609
 - интернационализация протокольных сообщений, 612
 - обработчики протоколов
 - в файлах, параметры настройки, 614
 - собственные, определение, 615
 - уровни, 613
 - установка, 613
 - прикладной интерфейс API, преимущества, 608
 - рекомендации по выполнению операций, 617
 - служебные методы, 610
 - уровни, 609
 - усовершенствованное, организация, 608
 - файлы протоколов
 - именование по шаблону, 614
 - ротация, 614
 - фильтрация протокольных записей, 616
 - форматирование протокольных записей, 616
 - элементарное, организация, 608
- Р**
- Развертывание приложений
 - JAR-файлы
 - запуск приложений, особенности, 528
 - назначение, 526
 - создание, 526
 - по технологии Java Web Start, 533
 - ресурсы
 - загрузка, 530
 - применение, 531
 - рановидности, 529
 - традиционный способ, 525
 - упаковка, 526
 - Рамки
 - компоновка, 428
 - назначение, 427
 - стили оформления, 428
 - Региональные настройки
 - назначение, 144
 - текущие и выбираемые, 144
 - Регулируемые ползунки
 - дополнение отметками, 436
 - компоновка, 435
 - назначение, 435
 - Ресурсы
 - порядок освобождения, 179; 596
 - система сборки мусора, 179
 - Рефлексия
 - анализ
 - объектов во время выполнения, 256
 - структуры классов, 250
 - вызов произвольных методов, 264
 - классы и методы, 250
 - манипулирование массивами, 261
 - механизм, 246
 - обобщенных типов, 677
 - применение, 246
- С**
- Связывание
 - динамическое, 202; 207
 - статическое, 207

- Символьные строки
 - в Java, 77
 - выделение подстрок, 77
 - построение, 86
 - принцип постоянства, 78
 - проверка на равенство, 79
 - пустые и нулевые, 80
 - сцепление, 77
 - Системные сбои, механизмы обработки, 605
 - Смешанное написание, 58
 - События
 - в окне, обработка, 373
 - действий, 361
 - источники и приемники, 360
 - модель делегирования, 360
 - низкоуровневые, 392
 - от мыши
 - обработка, 385–388
 - разновидности, 388
 - от щелчков на кнопке, обработка, 361
 - порядок обработки в Java, 360
 - простые приемники, создание, 368
 - семантические, 392
 - Состояние гонок
 - как явление, 774
 - причины возникновения, 778
 - Списки
 - комбинированные
 - компоновка, 432
 - назначение, 432
 - раскрывающиеся
 - назначение, 432
 - редактируемые, 432
 - связные
 - ввод элементов, 696
 - двунаправленные, 693
 - итераторы, особенности, 697
 - как упорядоченные коллекции, 695
 - организация, 693
 - применение, 699
 - реализация, 694
 - структурные видоизменения,
 - отслеживание, 697
 - удаление элементов, 697
 - Стеки, реализация, 748
- Т**
- Таблицы
 - методов, создание, 207
 - свойств
 - значения по умолчанию, 568
 - назначение, 567
 - особенности, 747
 - реализация, 567
 - характеристики, 567
 - Таймеры
 - запуск, 288
 - передача объекта для обратного вызова, 287
 - установка, 287
 - Текстовые области
 - автоматический перенос строк, 419
 - компоновка, 418
 - на панели прокрутки, 419
 - назначение, 418
 - Технология Java Web Start
 - порядок доставки приложений, 533
 - характеристики доставляемых
 - приложений, 533
 - цифровая подпись приложений, 538
 - прикладной интерфейс JNLP API
 - возможности, 540
 - службы, 541
 - песочница
 - назначение, 537
 - ограничения на исполняемые
 - программы, 537
 - подготовка приложений к доставке, 533
 - Типы данных
 - boolean, 66
 - char, 64
 - динамическая идентификация, 246
 - контейнерные, 242
 - обобщенные и базовые, соответствие, 653
 - перечислимые, 76; 244
 - простые, 62
 - целочисленные, 62
 - числовые
 - преобразование, 74
 - с плавающей точкой, 63
 - Трассировка стека
 - вывод, 598
 - определение, 597
 - получение, 597
- У**
- Уникод
 - кодировка
 - UTF-16, 66
 - назначение, 65
 - кодовые
 - единицы, 66
 - плоскости, 66
 - точки, 65
 - область подстановки, 66
 - Управляющие последовательности
 - специальных символов, 65
 - Условные операторы if
 - общая форма, 99
 - повторяющиеся, 100
 - принцип действия, 97
 - Утверждения
 - документирование предположений, 607
 - назначение, 603
 - проверка параметров, 605
 - разрешение и запрет, 604
 - условия для применения, 605
 - Утилиты
 - appletviewer, применение, 53; 551
 - jar
 - параметры, 526
 - применение, 186; 526
 - javac, применение, 45
 - javadoc, применение, 189
 - javap, применение, 295
 - java, применение, 45
 - jconsole, применение, 612; 629
 - jar, применение, 630

Ф

- Фигурные скобки
 - назначение, 59
 - стиль расстановки, 60
- Флажки
 - компоновка, 422
 - установка и сброс, 422
- Фокус ввода
 - отображение, 380
 - перемещение, 380
 - с клавиатуры, 380
- Форматирование
 - выводимых данных, 91
 - даты и времени, 92
 - символы преобразования, 91
 - спецификатор формата, 91
 - флаги, 91
- Фреймы
 - вывод данных в компоненте, 329
 - задание размеров, 324
 - определение, 319
 - отображение, 321
 - расположение, 322
 - свойства, 324
 - создание, 319
 - рекомендации по обращению, 326

Х

- Хеш-коды
 - вычисление, алгоритм, 224; 703
 - определение, 224; 703
 - порождение, 703
- Хеш-множества
 - ввод и вывод элементов, 705
 - итераторы, применение, 704
- Хеш-таблицы
 - группы, 703
 - коэффициент загрузки, 704
 - назначение, 703
 - повторное хеширование, 704
 - реализация, 703
 - связные, 721
 - хеш-конфликты, явление, 703

Ц

- Циклы
 - do-while, принцип действия, 101
 - for, принцип действия, 105
 - while, принцип действия, 100
 - в стиле for each
 - организация, 687
 - применение, 118; 687
 - принцип действия, 118

Ш

- Шаблоны проектирования
 - MVC
 - взаимодействие составляющих, 401
 - описание, 399
 - преимущества, 401
 - составляющие, назначение и реализация, 400
 - в программировании, другие, 399
 - в строительстве, 398
 - определение, 398
 - Шрифтовое оформление
 - выравнивание текста
 - по высоте, 350
 - по центру, 348
 - рамка, ограничивающая текст, 349
 - типографские характеристики, 349
 - Шрифты
 - ввод из файлов, 348
 - гарнитура, 346
 - доступные в системе, 346
 - логические названия, 347
 - начертание, 346
 - размеры в пунктах, 347

Я

- Язык Java
 - версии, 31
 - виртуальная машина, 30
 - компилятор, 45
 - краткая история развития, 30
 - особенности, 22–28
 - программная платформа, 22
 - программные средства, обозначение, 38
 - распространенные заблуждения, 32
 - система безопасности, 25
 - строго типизированный, 62
 - учет регистра символов, 58