ПОЛНОСТЬЮ ОБНОВЛЕНО С УЧЕТОМ JAVA SE 7

PRENTICE HALL

Java™

БИБЛИОТЕКА ПРОФЕССИОНАЛА

Том 2. Расширенные средства программирования

ДЕВЯТОЕ ИЗДАНИЕ

Кей Хорстманн, Гари Корнелл

Библиотека профессионала

Java™

Том 2. Расширенные средства программирования

Девятое издание

Кей Хорстманн
Гари Корнелл

ВИЛЬЯМС

Москва • Санкт-Петербург • Киев
2014

ББК 32.973.26-018.2.75
Х82
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.

Х82 Java. Библиотека профессионала, том 2. Расширенные средства, 9-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2014. — 1008 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1870-3 (рус., том 2)
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 2. Расширенные средства, 9-е издание

Литературный редактор *И.А. Попова*
Верстка *О.В. Мишутина*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 04.11.2013. Формат 70x100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 63,0. Уч.-изд. л. 51,3
Тираж 1500 экз. Заказ № 3969

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 978-5-8459-1870-3 (рус., том 2)
ISBN 978-5-8459-1886-4 (рус., многотом.)
ISBN 978-0-13-708160-8 (англ.)

© Издательский дом "Вильямс", 2014

© Oracle and/or its affiliates, 2013

Оглавление

Предисловие 13

Глава 1. Потоки ввода-вывода и файлы 19

Глава 2. XML 107

Глава 3. Работа в сети 191

Глава 4. Работа с базами данных 235

Глава 5. Интернационализация 299

Глава 6. Расширенные средства Swing 353

Глава 7. Расширенные средства AWT 525

Глава 8. Компоненты JavaBeans 687

Глава 9. Безопасность 759

Глава 10. Создание сценариев, компиляция и обработка аннотаций 841

Глава 11. Распределенные объекты 897

Глава 12. Собственные методы 929

Предметный указатель 987

Содержание

Предисловие 13
К читателю 13
Краткий обзор книги 13
Условные обозначения 16
Примеры исходного кода 16
Благодарности 17
От издательства 18

Глава 1. Потоки ввода-вывода и файлы 19
Потоки ввода-вывода 20
Чтение и запись байтов 20
Полный комплект потоков ввода-вывода 23
Сочетание потоковых фильтров 27
Ввод-вывод текста 30
Вывод текста 31
Ввод текста 34
Сохранение объектов в текстовом формате 34
Наборы символов 37
Чтение и запись двоичных данных 42
Файлы с произвольным доступом 45
ZIP-архивы 49
Потоки ввода-вывода и сериализация объектов 53
Представление о формате файлов для сериализации объектов 58
Видоизменение исходного механизма сериализации 64
Сериализация одноэлементных множеств и типизированных перечислений 66
Контроль версий 67
Применение сериализации для клонирования 69
Обращение с файлами 72
Пути к файлам 72
Чтение и запись данных в файлы 75
Копирование, перемещение и удаление файлов 76
Создание файлов и каталогов 77
Получение сведений о файлах 78
Перемещение по файлам и каталогам 79
Системы ZIP-файлов 82
Отображаемые в памяти файлы 83
Структура буфера данных 91
Блокирование файлов 93
Регулярные выражения 96

Глава 2. XML 107
Введение в XML 108
Структура XML-документа 110
Синтаксический анализ XML-документов 112
Проверка достоверности XML-документов 125
Определения типов документов 126
Схема XML-документов 133
Практический пример применения XML-документов 135
Поиск информации средствами XPath 149
Использование пространств имен 155
Потоковые синтаксические анализаторы 157
Применение SAX-анализатора 158
Применение StAX-анализатора 163
Формирование XML-документов 167
XML-документы без пространств имен 167
XML-документы с пространствами имен 168
Запись XML-документов 168
Пример формирования файла в формате SVG 169
Запись XML-документов средствами StAX 172
Преобразование XML-документов языковыми средствами XSLT 180

Глава 3. Работа в сети 191
Подключение к серверу 191
Время ожидания для сокетов 196
Межсетевые адреса 197
Реализация серверов 199
Обслуживание многих клиентов 202
Полузакрытие 206
Прерываемые сокеты 207
Получение данных из Интернета 213
URL и URI 213
Извлечение данных средствами класса **URLConnection** 215
Отправка данных формы 224
Отправка электронной почты 231

Глава 4. Работа с базами данных 235
Структура JDBC 236
Типы драйверов JDBC 236
Типичные примеры применения JDBC 237
Язык SQL 239
Конфигурирование JDBC 244
URL базы данных 245
JAR-файлы драйверов 245
Запуск базы данных 245
Регистрация класса драйвера 246
Соединение с базой данных 247
Выполнение команд SQL 249
Управление соединениями, командами и результирующими наборами 253
Анализ исключений SQL 253
Заполнение базы данных 256

Выполнение запросов 259
Подготовленные операторы и запросы 260
Чтение и запись больших объектов 266
Синтаксис переходов в SQL 268
Множественные результаты 269
Извлечение автоматически генерируемых ключей 270
Прокручиваемые и обновляемые результирующие наборы 270
Прокручиваемые результирующие наборы 271
Обновляемые результирующие наборы 273
Наборы строк 278
Построение наборов строк 278
Кешируемые наборы строк 279
Метаданные 282
Транзакции 291
Точки сохранения 292
Групповые обновления 293
Расширенные типы данных SQL 295
Управление соединениями с базами данных в веб-приложениях и производственных приложениях 296

Глава 5. Интернационализация 299

Региональные настройки 300
Числовые форматы 305
Форматирование денежных сумм в разных валютах 311
Форматирование даты и времени 312
Сортировка 320
Избирательность сортировки 321
Разложение на составляющие 322
Форматирование сообщений 327
Форматы выбора 329
Текстовые файлы и наборы символов 331
Кодирование символов в исходных файлах 332
Комплекты ресурсов 333
Обнаружение комплектов ресурсов 333
Файлы свойств 335
Классы комплектов ресурсов 335
Пример интернационализации прикладной программы 337

Глава 6. Расширенные средства Swing 353

Списки 354
Компонент `JList` 354
Модели списков 360
Ввод и удаление значений 365
Воспроизведение значений 366
Таблицы 370
Простая таблица 370
Модели таблиц 375
Обращение со строками и столбцами таблицы 378
Воспроизведение и редактирование ячеек таблицы 394

Деревья 407
Простые деревья 408
Перечисление узлов 423
Воспроизведение узлов 425
Обработка событий в деревьях 428
Специальные модели деревьев 435
Текстовые компоненты 443
Отслеживание изменений в текстовых компонентах 444
Поля ввода форматируемого текста 448
Компонент **JSpinner** 465
Отображение HTML-документов средствами **JEditorPane** 473
Индикаторы состояния 479
Индикаторы выполнения 480
Мониторы текущего состояния 483
Контроль чтения данных из потока ввода 486
Организаторы и декораторы компонентов 492
Разделяемые панели 492
Панели с вкладками 495
Настольные панели и внутренние фреймы 501
Каскадное и мозаичное расположение фреймов 505
Наложение запрета на установку свойств 508

Глава 7. Расширенные средства AWT 525

Конвейер визуализации 526
Фигуры 528
Применение классов фигур 530
Участки 544
Обводка 546
Раскраска 554
Преобразование координат 556
Отсечение 562
Прозрачность и композиция 564
Указания по воспроизведению 573
Чтение и запись изображений 579
Получение средств чтения и записи изображений по типам файлов 580
Чтение и запись файлов с несколькими изображениями 581
Манипулирование изображениями 590
Формирование растровых изображений 590
Фильтрация изображений 597
Вывод изображений на печать 605
Вывод двухмерной графики на печать 605
Многостраничная печать 615
Предварительный просмотр печати 617
Службы печати 625
Потоковые службы печати 630
Буфер обмена 637
Классы и интерфейсы для передачи данных 638
Передача текста 639
Интерфейс **Transferable** и разновидности данных 643
Передача изображений через буфер обмена 645

Передача объектов Java через системный буфер обмена 649
Передача ссылок на объекты через локальный буфер обмена 652
Перетаскивание объектов 653
Поддержка передачи данных в Swing 655
Источники перетаскивания 659
Приемники перетаскивания 662
Начальные экраны 670
Запуск настольных приложений 675
Системная область 680

Глава 8. Компоненты JavaBeans 687

Назначение компонентов JavaBeans 688
Создание компонентов JavaBeans 690
Применение компонентов JavaBeans для разработки приложений 693
Упаковка компонентов JavaBeans в JAR-файлы 693
Составление приложения из компонентов JavaBeans в ИСР 695
Шаблоны именования свойств и событий в компонентах JavaBeans 702
Типы свойств компонентов JavaBeans 705
Простые свойства 705
Индексируемые свойства 706
Связанные свойства 706
Ограниченные свойства 708
Классы типа **`BeanInfo`** 715
Редакторы свойств 719
Создание редакторов свойств 722
Настройщики 730
Создание класса настройщика 732
Сохраняемость компонентов JavaBeans 738
Применение сохраняемости компонентов JavaBeans к произвольным данным 742
Исчерпывающий пример сохраняемости компонентов JavaBeans 748

Глава 9. Безопасность 759

Загрузчики классов 760
Иерархия загрузчиков классов 761
Применение загрузчиков классов в качестве пространств имен 764
Создание собственного загрузчика классов 764
Верификация байт-кода 770
Диспетчеры защиты и полномочия 775
Организация защиты на платформе Java 776
Файлы правил защиты 779
Пользовательские полномочия 785
Реализация класса полномочий 787
Аутентификация пользователей 793
Модули регистрации в службе JAAS 799
Цифровые подписи 807
Свертки сообщений 808
Подписание сообщений 811
Верификация подписи 813
Проблема аутентификации 816

Подписание сертификатов 819
Запросы сертификатов 820
Подписание кода 821
Подписание JAR-файлов 821
Сертификаты разработчиков программного обеспечения 826
Шифрование 828
Симметричные шифры 828
Генерирование ключей шифрования 830
Потоки шифрования 835
Шифрование открытым ключом 836

Глава 10. Создание сценариев, компиляция и обработка аннотаций 841

Написание сценариев для платформы Java 842
Получение механизма сценариев 842
Оценка сценариев и привязки 843
Переадресация ввода-вывода 845
Вызов сценарных функций и методов 846
Компиляция сценариев 848
Пример создания сценария для обработки событий в ГПИ 849
Прикладной интерфейс API для компилятора 854
Простой способ компилирования 854
Выполнение заданий на компиляцию 854
Пример динамического генерирования кода Java 860
Применение аннотаций 865
Пример аннотирования обработчиков событий 866
Синтаксис аннотаций 871
Стандартные аннотации 875
Аннотации для компиляции 876
Аннотации для управления ресурсами 877
Мета-аннотации 877
Обработка аннотаций на уровне исходного кода 880
Конструирование байт-кодов 887
Видоизменение байт-кодов во время загрузки 893

Глава 11. Распределенные объекты 897

Роли клиента и сервера 898
Вызовы удаленных методов 900
Заглушки и упаковка параметров 901
Модель программирования RMI 902
Интерфейсы и реализации 902
Реестр RMI 904
Развертывание программы 908
Протоколирование действий RMI 911
Параметры и возвращаемые значения в удаленных методах 913
Передача удаленных объектов 913
Передача неудаленных объектов 913
Динамическая загрузка классов 916
Удаленные ссылки на несколько интерфейсов 920
Удаленные объекты и методы **`equals()`**, **`hashCode()`** и **`clone()`** 921
Активизация удаленных объектов 922

Глава 12. Собственные методы 929
Вызов функции на C из программы на Java 930
Числовые параметры и возвращаемые значения 936
Форматирование чисел с помощью функции **`printf()`** 937
Строковые параметры 938
Доступ к полям 944
Доступ к полям экземпляра 944
Доступ к статическим полям 948
Кодирование сигнатур 949
Вызов методов на Java 951
Методы экземпляра 951
Статические методы 954
Конструкторы 955
Альтернативные вызовы методов 955
Доступ к элементам массивов 957
Обработка ошибок 961
Применение прикладного интерфейса API для вызовов 965
Пример обращения к реестру Windows 970
Общее представление о реестре Windows 971
Интерфейс на платформе Java для доступа к реестру 972
Реализация функций доступа к реестру в виде собственных методов 973

Предметный указатель 987

Предисловие

К читателю

Книга, которую вы держите в руках, является вторым томом девятого издания, полностью обновленного по версии Java SE 7. В первом томе рассматривались основные языковые средства Java, а в этом томе речь пойдет о расширенных функциональных возможностях, которые могут понадобиться программисту для разработки программного обеспечения на высоком профессиональном уровне. Поэтому этот том, как, впрочем, и первый том настоящего и предыдущих изданий данной книги, нацелен на тех программистов, которые собираются применять технологию Java в работе над реальными проектами.

Следует, однако, иметь в виду, что опытным разработчикам, умеющим свободно обращаться с такими сложными языковыми средствами, как внутренние и обобщенные классы, читать первый том для понимания материала второго тома совсем не обязательно. И хотя в этом томе делаются ссылки (там, где требуется) на главы первого тома, который, мы надеемся, вы все-таки приобретете или уже приобрели, весь необходимый материал по основам Java нетрудно найти в любой начальной литературе по платформе Java.

Краткий обзор книги

В целом главы этого тома составлены независимо друг от друга. Это дает читателю возможность начинать изучение материала с той темы, которая интересует его больше всего, и вообще читать главы второго тома книги в любом удобном ему порядке.

Глава 1 посвящена организации ввода-вывода. В Java весь ввод-вывод осуществляется через так называемые *потоки ввода-вывода*. Такие потоки позволяют единообразно обмениваться данными между различными источниками, включая файлы, сетевые соединения и блоки памяти. В начале этой главы приводится подробное описание классов чтения и записи в потоки ввода-вывода, упрощающие обработку данных в уникоде. Далее в ней рассматривается внутренний механизм сериализации объектов, который делает простым и удобным сохранение и загрузку объектов. И в завершение главы обсуждаются регулярные выражения и библиотека NIO2 в версии Java SE 7, благодаря которой становятся очень удобными такие типичные операции ввода-вывода, как, например, чтение всех строк из файла.

Основной темой **главы 2** является XML. В ней показывается, каким образом осуществляется синтаксический анализ XML-файлов, генерируется разметка в коде XML и выполняются XSL-преобразования. В качестве примера демонстрируется порядок обозначения компоновки Swing-формы в формате XML. В этой главе рассматривается

также прикладной интерфейс API XPath, в значительной степени упрощающий поиск мелких подробностей в больших объемах данных XML.

В **главе 3** рассматривается сетевой прикладной интерфейс API. В Java чрезвычайно просто решаются сложные задачи сетевого программирования. В этой главе показывается, как устанавливаются сетевые соединения с серверами, реализуются собственные серверы и организуется связь по протоколу HTTP.

Глава 4 посвящена программированию баз данных. Основное внимание в ней уделяется интерфейсу JDBC (прикладному интерфейсу API для организации доступа к базам данных из приложений на Java), который позволяет прикладным программам на Java устанавливать связь с реляционными базами данных. В этой главе также показывается, как писать полезные программы для выполнения рутинных операций с настоящими базами данных, применяя только базовое подмножество интерфейса JDBC. (Для рассмотрения всех средств интерфейса JDBC потребовалась бы отдельная книга почти такого же объема, как и эта.) И в завершение главы приводятся краткие сведения об интерфейсе JNDI (Java Naming and Directory Interface — интерфейс именования и каталогов Java).

В **главе 5** обсуждаются вопросы интернационализации, важность которой, на наш взгляд, будет со временем только возрастать. Java относится к тем немногочисленным языкам программирования, где с самого начала предусматривалась возможность обработки данных в уникоде (Unicode), но поддержка интернационализации в Java этим не ограничивается. В частности, интернационализация прикладных программ на Java позволяет сделать их независимыми не только от платформы, но и от страны применения. В качестве примера в этой главе демонстрируется, как написать аплет для расчета времени выхода на пенсию с выбором английского, немецкого или китайского языка в зависимости от региональных настроек в браузере.

В **главе 6** представлен весь материал по библиотеке Swing, который не вошел в первый том данной книги, в том числе описание важных и сложных компонентов деревьев и таблиц. В ней демонстрируются основные способы применения панелей редактора, реализация в Java многодокументного интерфейса, индикаторы выполнения, применяемые в многопоточных программах, а также средства интеграции для рабочего стола вроде заставок и поддержки области уведомлений на панели задач. Опять же основное внимание уделяется только наиболее полезным конструкциям, которые разработчикам чаще всего приходится использовать на практике, поскольку даже краткое рассмотрение всей библиотеки Swing заняло бы не один том и представляло бы интерес только для очень узких специалистов.

В **главе 7** рассматривается прикладной интерфейс Java 2D API, которым можно пользоваться для рисования реалистичных графических изображений и спецэффектов. В ней также рассказывается о некоторых более развитых средствах библиотеки AWT (Abstract Windowing Toolkit — набор инструментальных средств для абстрактных окон), которые оказались слишком специфическими для рассмотрения в первом томе и, тем не менее, должны быть включены в арсенал средств всякого программирующего на Java. К их числу относятся средства вывода на печать и прикладные интерфейсы API, позволяющие выполнять операции вырезания, вставки и перетаскивания объектов.

В **главе 8** рассказывается обо всем, что требуется знать разработчикам о доступном на платформе Java прикладном интерфейсе API для программирования компонентов JavaBeans. В этой главе показывается, как создавать свои собственные компоненты JavaBeans, чтобы другие программисты могли манипулировать ими в интегрированных средах разработки (ИСР). И в завершение главы поясняется, как пользоваться

механизмом сохраняемости компонентов JavaBeans для длительного хранения данных в пригодном для этого формате, в отличие от сериализации объектов.

В **главе 9** представлена модель безопасности Java. Платформа Java с самого начала разрабатывалась с учетом безопасности, и в этой главе объясняется, что именно позволяет ей обеспечивать безопасность. Сначала в ней демонстрируется, как создавать свои собственные загрузчики классов и диспетчеры защиты для специальных приложений. Затем рассматривается прикладной интерфейс API для безопасности, который позволяет оснащать приложения важными средствами вроде механизма цифровых подписей сообщений и кода, а также авторизации, аутентификации и шифрования. И завершается эта глава демонстрацией примеров, в которых применяются такие алгоритмы шифрования, как AES и RSA.

Глава 10 посвящена распределенным объектам. В ней, в частности, подробно рассматривается протокол RMI (Remote Method Invocation — вызов удаленных методов). Этот протокол, по существу, представляет собой прикладной интерфейс API и позволяет обращаться с объектами Java, которые распределены среди многих компьютеров.

В **главе 11** описываются три разные технологии для обработки кода. Так, прикладные интерфейсы API для сценариев и компилятора дают возможность вызывать в программе на Java код, написанный на каком-нибудь языке создания сценариев, например JavaScript или Groovy, и компилировать его в код Java. Аннотации позволяют вводить в программу на Java произвольно выбираемую информацию (иногда еще также называемую *метаданными*). В этой главе показывается, каким образом обработчики аннотаций собирают аннотации на уровне источника и на уровне файлов классов и как с помощью аннотаций оказывается воздействие на поведение классов во время выполнения. Аннотации выгодно использовать только вместе с подходящими инструментальными средствами, и мы надеемся, что материал этой главы поможет читателю научиться выбирать именно те средства обработки аннотаций, которые в наибольшей степени отвечают его потребностям.

Глава 12 посвящена собственным методам, которые позволяют вызывать функции, специально написанные для конкретной платформы, например Microsoft Windows. Очевидно, что данное языковое средство является спорным, ведь применение собственных методов сводит на нет все межплатформенные преимущества Java. Тем не менее всякий, серьезно программирующий на Java приложения для конкретных платформ, должен уметь обращаться с собственными методами. Ведь иногда возникают ситуации, когда требуется обращаться к прикладному интерфейсу API операционной системы целевой платформы для взаимодействия с устройствами или службами, которые не поддерживаются на платформе Java. В этой главе показано, как это сделать, на примере организации доступа из программы на Java к прикладному интерфейсу API системного реестра Windows.

Как обычно, все главы второго тома были полностью обновлены по самой последней версии Java. Весь устаревший материал был изъят, а новые прикладные интерфейсы API, появившиеся в версии Java SE 7, подробно рассматриваются в соответствующих местах.

Условные обозначения

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

НА ЗАМЕТКУ! Этой пиктограммой выделяются примечания.

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 `ScriptTest.java`

Примеры исходного кода

Все примеры исходного кода, приведенные в этом томе в частности и в данной книге вообще, доступны в архивированном виде на посвященном ей веб-сайте по адресу `http://horstmann.com/corejava`.

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 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 Доенч (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 в 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) из компании Oracle, Алека Битона (Alec Beaton) из PointBase, Inc., Клиффа Берга (Cliff Berg), Джошуа Блоха (Joshua Bloch), Дэвида Брауна (David Brown), Корки Картрайта (Corky Cartwright), Френка Коена (Frank Cohen) из PushToTest, Криса Крейна (Chris Crane) из devXsolution, доктора Николаса Дж. Де Лилло (Dr. Nicholas J. De Lillo) из Манхеттенского колледжа, Ракеша Дхупара (Rakesh Dhoopar) из компании Oracle, Дэвида Джири (David Geary) из Sabreware, Джима Гиша (Jim Gish) из Oracle, Брайана Гоетца (Brian Goetz) из Oracle, Анджелу 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 Adobe Systems, Дэна Харки (Dan Harkey) из университета штата Калифорния в Сан-Хосе, Вильяма Хиггинса (William Higgins) из 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), Ильбера Рамадани (Ylber Ramadani) из университета имени Райерсона, Стюарта Реджеса (Stuart Reges) из университета штата Аризона, Рича Розена

(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 Севинка (Paul Sevinc) из Teamup AG, Деванг Ша (Devang Shah) из Oracle, Ричарда Сливчака (Richard Slywczak) из Исследовательского центра имени Гленна, НАСА, Бредли А. Смита (Bradley A. Smith), Стивена Стелтинга (Steven Stelting) из Sun Microsystems, Кристофера Тэйлора (Christopher Taylor), Люка Тэйлора (Luke Taylor) из Valtech, Джорджа Тхируватукала (George Thiruvathukal), Кима Топли (Kim Topley), автора книги *Core JFC, Second Edition*, Джанет Трауб (Janet Traub), Пола Тиму (Paul Tyma), консультанта, Питера Ван Дер Линдена (Peter van der Linden) из Motorola Mobile Devices, Берта Уолша (Burt Walsh), Джо Уанга (Joe Wang) из Oracle и Дана Ксю (Dan Xu) из Oracle.

Кей Хорстманн,
Сан-Франциско, декабрь 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

ГЛАВА 1

Потоки ввода-вывода и файлы

В этой главе...

- Потоки ввода-вывода
- Ввод-вывод текста
- Чтение и запись двоичных данных
- ZIP-архивы
- Потоки ввода-вывода и сериализация объектов
- Обращение с файлами
- Отображаемые в памяти файлы
- Регулярные выражения

В этой главе речь пойдет о прикладных интерфейсах (API), доступных в Java для организации ввода-вывода данных. Из нее вы, в частности, узнаете, как получать доступ к файлам и каталогам, как читать и записывать данные в двоичном и текстовом формате. Кроме того, рассматривается механизм сериализации, позволяющий сохранять объекты так же просто, как и текстовые или числовые данные. Далее в главе представлен ряд усовершенствований в пакете "новых средств ввода-вывода" (new I/O — NIO) с соответствующим названием `java.nio`, внедренном еще в версии Java SE 1.4 и дополненном рядом новых усовершенствований в версии Java 7. В завершение главы обсуждаются регулярные выражения, хотя они и не имеют непосредственного отношения к потокам ввода-вывода и файлам. Тем не менее трудно найти более подходящее место для обсуждения этой темы, как его, очевидно, не удалось найти и разработчикам Java, присоединившим спецификацию прикладного интерфейса API для регулярных выражений к запросу на уточнение новых средств ввода-вывода.

Потоки ввода-вывода

В прикладном интерфейсе Java API объект, из которого можно читать последовательность байтов, называется *потоком ввода*, а объект, в который можно записывать последовательность байтов, — *потоком вывода*. В роли таких источников и приемников последовательностей байтов чаще всего выступают файлы, но могут также быть сетевые соединения и даже блоки памяти. Абстрактные классы `InputStream` и `OutputStream` служат основанием для иерархии классов ввода-вывода.

Байтовые потоки ввода-вывода неудобны для обработки информации, хранящейся в уникоде (Unicode; напомним, что в уникоде на каждый символ приходится несколько байтов). Поэтому для обработки символов в уникоде предусмотрена отдельная иерархия классов, наследующих от абстрактных классов `Reader` и `Writer`. Эти классы позволяют выполнять операции чтения и записи не однобайтовыми символами, а двухбайтовыми кодовыми единицами в уникоде.

Чтение и запись байтов

В классе `InputStream` имеется следующий абстрактный метод:

```
abstract int read()
```

Этот метод читает один байт и возвращает считанный байт или значение **-1**, если обнаруживается конец источника ввода. Разработчик конкретного класса потока ввода может переопределить этот метод таким образом, чтобы он предоставлял какую-нибудь полезную функциональную возможность. Например, в классе `FileInputStream` этот метод выполняет чтение одного байта из файла. А поток ввода `System.in` представляет собой предопределенный объект подкласса `InputStream`, который позволяет читать данные, вводимые с клавиатуры.

У класса `InputStream` имеются также неабстрактные методы для чтения массива байтов или пропуска определенного ряда байтов. В этих методах вызывается абстрактный метод `read()`, благодаря чему в подклассах достаточно переопределить только один метод.

Аналогично в классе `OutputStream` определяется следующий абстрактный метод, записывающий один байт в указанное место для вывода данных:

```
abstract void write(int b)
```

Как методы `read()`, так и методы `write()` *блокируют* доступ до тех пор, пока байты не будут фактически считаны или записаны. Это означает, что если к потоку ввода-вывода не удается получить доступ немедленно (что обычно случается из-за занятости сетевого соединения), происходит блокирование текущего потока исполнения. А это дает другим потокам исполнения возможность выполнять какую-нибудь полезную задачу, в то время как метод ожидает, когда поток ввода-вывода станет снова доступным.

Метод `available()` позволяет проверить количество байтов, доступное для считывания в текущий момент. Это означает, что фрагмент кода, аналогичный приведенному ниже, вряд ли приведет к блокировке.

```
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
  byte[] data = new byte[bytesAvailable];
```

```
  in.read(data);
}
```

По завершении чтения или записи данных в поток ввода-вывода следует непременно закрыть его, вызвав метод `close()`. Такой вызов приводит к освобождению системных ресурсов, доступных в ограниченном количестве. Если же в прикладной программе открывается слишком много потоков ввода-вывода без последующего их закрытия, ресурсы системы могут исчерпаться. Кроме этого, закрытие потока вывода приводит к *очистке* использовавшегося для него буфера: все символы, которые временно размещались в этом буфере с целью их последующей доставки в виде более крупного пакета, рассылаются по местам своего назначения. Так, если не закрыть файл, последний пакет байтов может так никогда и не быть доставлен. Очистить буфер от выводимых данных можно и вручную с помощью метода `flush()`.

Даже если в классе потока ввода-вывода предоставляются конкретные методы для работы с базовыми функциями `read()` и `write()`, разработчики прикладных программ редко пользуются ими. Для них больший интерес представляют данные, содержащие числа, символьные строки и объекты, а не исходные байты. Поэтому в Java предоставляется немало классов потоков ввода-вывода, наследуемых от базовых классов `InputStream` и `OutputStream` и позволяющих обрабатывать данные в нужной форме, а не просто в виде отдельных байтов.

java.io.InputStream 1.0

- `abstract int read()`

  Считывает байт данных и возвращает его. По достижении конца потока возвращает значение `-1`.

- `int read(byte[] b)`

  Считывает данные в байтовый массив и возвращает фактическое количество считанных байтов или значение **-1**, если достигнут конец потока ввода. Этот метод позволяет считать максимум *b.length* байтов.

- `int read(byte[] b, int off, int len)`

  Считывает данные в байтовый массив. Возвращает фактическое количество считанных байтов или значение **-1**, если достигнут конец потока ввода.

  Параметры:

b	Массив, в который должны считываться данные
off	Смещение в массиве `b`, обозначающее позицию, с которой должно начинаться размещение в нем байтов
len	Максимальное количество считываемых байтов

`long skip(long n)`

Пропускает `n` байтов в потоке ввода. Возвращает фактическое количество пропущенных байтов (которое может оказаться меньше `n`, если достигнут конец потока).

- `int available()`

  Возвращает количество байтов, доступных без блокирования. (Напомним, что блокирование означает потерю текущим потоком исполнения своей очереди на выполнение.)

- `void close()`

  Закрывает поток ввода.

- `void mark(int readlimit)`

  Устанавливает маркер на текущей позиции в потоке ввода. (Не все потоки поддерживают такую функциональную возможность.) Если из потока ввода считано байтов больше заданного предела `readlimit`, в потоке ввода можно пренебречь устанавливаемым маркером.

- `void reset()`

  Возвращается к последнему маркеру. Последующие вызовы метода `read()` приводят к повторному считыванию байтов. В отсутствие текущего маркера поток ввода не устанавливается в исходное положение.

- `boolean markSupported()`

  Возвращает логическое значение `true`, если в потоке ввода поддерживается возможность устанавливать маркеры.

java.io.InputStream 1.0

- `abstract void write(int n)`

  Записывает байт данных.

- `void write(byte[] b)`
- `void write(byte[] b, int off, int len)`

  Записывают все байты или определенный ряд байтов из массива `b`.

  Параметры:

`b`	Массив, из которого должны выбираться данные для записи
`off`	Смещение в массиве `b`, обозначающее позицию, с которой должна начинаться выборка байтов для записи
`len`	Общее количество записываемых байтов

- `void close()`

  Очищает и закрывает поток вывода.

- `void flush()`

  Очищает поток вывода, отправляя любые находящиеся в буфере данные по месту их назначения.

Полный комплект потоков ввода-вывода

В отличие от языка C, где для ввода-вывода достаточно и единственного типа `FILE*`, в Java имеется целый комплект из более чем 60 (!) различных типов потоков ввода-вывода (рис. 1.1 и 1.2). Разделим эти типы по областям их применения. Так, для классов, обрабатывающих байты и символы, существуют отдельные иерархии.

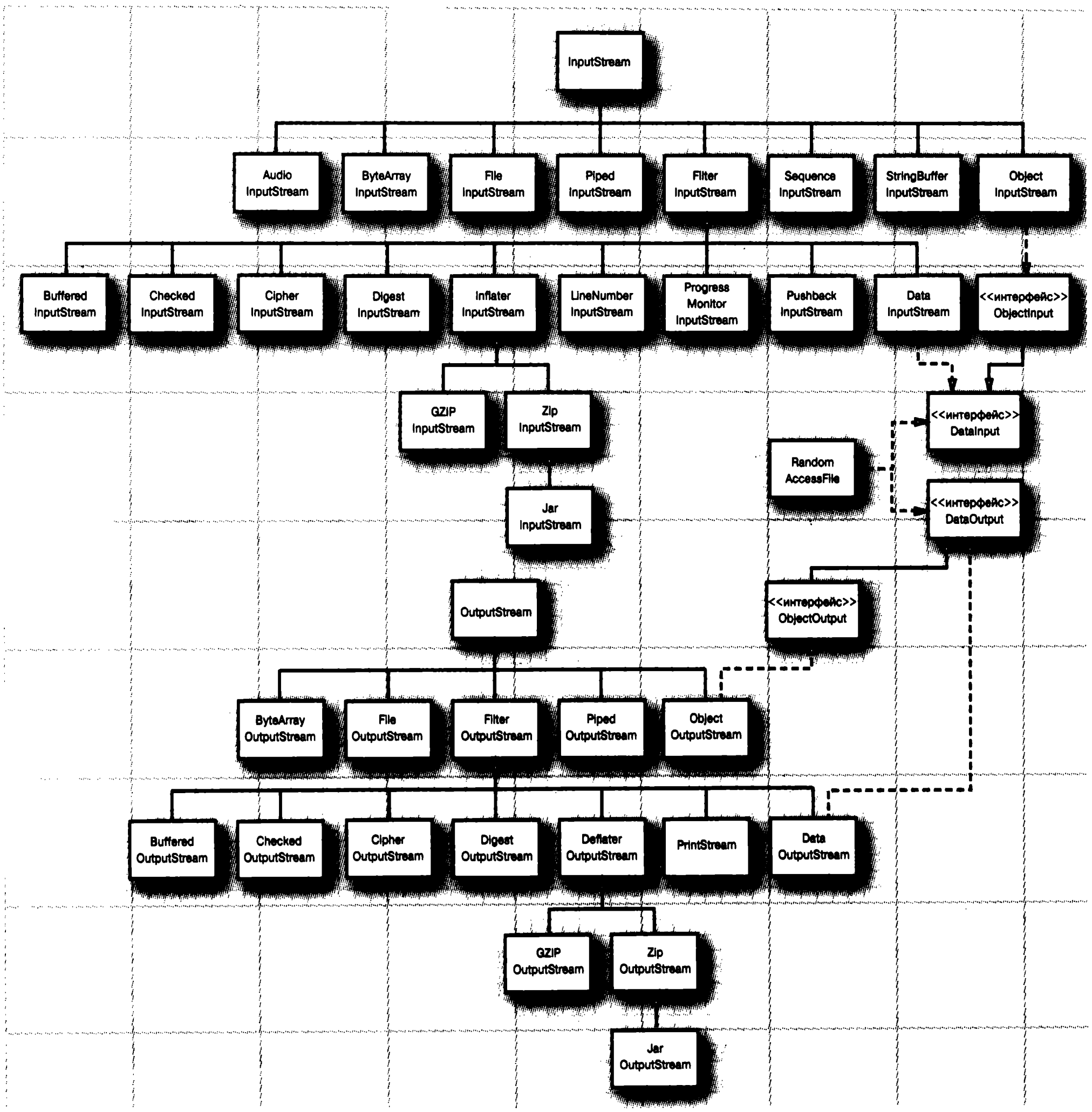


Рис. 1.1. *Иерархия классов для потоков ввода-вывода*

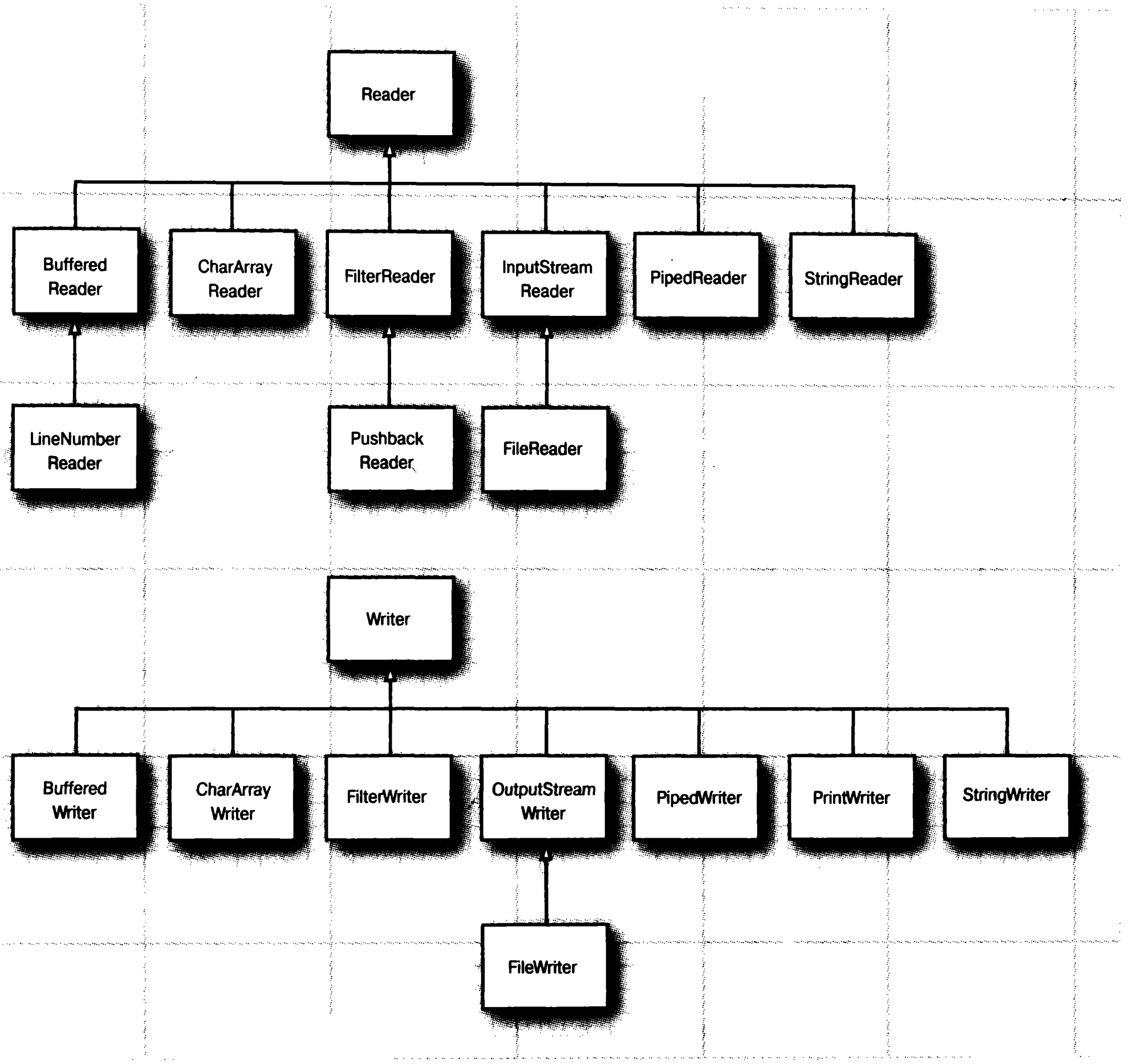


Рис. 1.2. Иерархия классов `Reader` и `Writer`

Как упоминалось выше, классы `InputStream` и `OutputStream` позволяют выполнять чтение и запись отдельных байтов и массивов байтов. Эти классы образуют основу иерархии, приведенной на рис. 1.1. Для чтения и записи символьных строк и чисел требуются подклассы, обладающие немалыми функциональными возможностями. Например, классы `DataInputStream` и `DataOutputStream` позволяют выполнять чтение и запись всех простых типов данных Java в двоичном формате. И, наконец, имеются классы для выполнения отдельных полезных операций ввода-вывода, например классы `ZipInputStream` и `ZipOutputStream`, позволяющие читать и записывать данные в файлы с уплотнением в таком известном формате, как ZIP.

С другой стороны, для ввода-вывода текста в уникоде необходимо обращаться к подклассам таких абстрактных классов, как `Reader` и `Writer` (см. рис. 1.2). Базовые методы из классов `Reader` и `Writer` похожи на базовые методы из классов `InputStream` и `OutputStream`, как показано ниже.

```
abstract int read()
abstract void write(int c)
```

Метод `read()` возвращает кодовую единицу в уникоде (в виде целого числа от **0** до **65535**) или значение **-1**, если достигнут конец файла. А метод `write()` вызывается с заданной кодовой единицей в уникоде. Подробнее о кодовых единицах в частности и уникоде вообще см. в главе 3 первого тома данной книги.

Имеются также четыре дополнительных интерфейса: `Closeable`, `Flushable`, `Readable` и `Appendable` (рис. 1.3). Первые два из них очень просты: в них определяется единственный метод `void close() throws IOException` и `void flush()` соответственно. Классы `InputStream`, `OutputStream`, `Reader` и `Writer` реализуют интерфейс `Closeable`, а классы `OutputStream` и `Writer` — интерфейс `Flushable`.

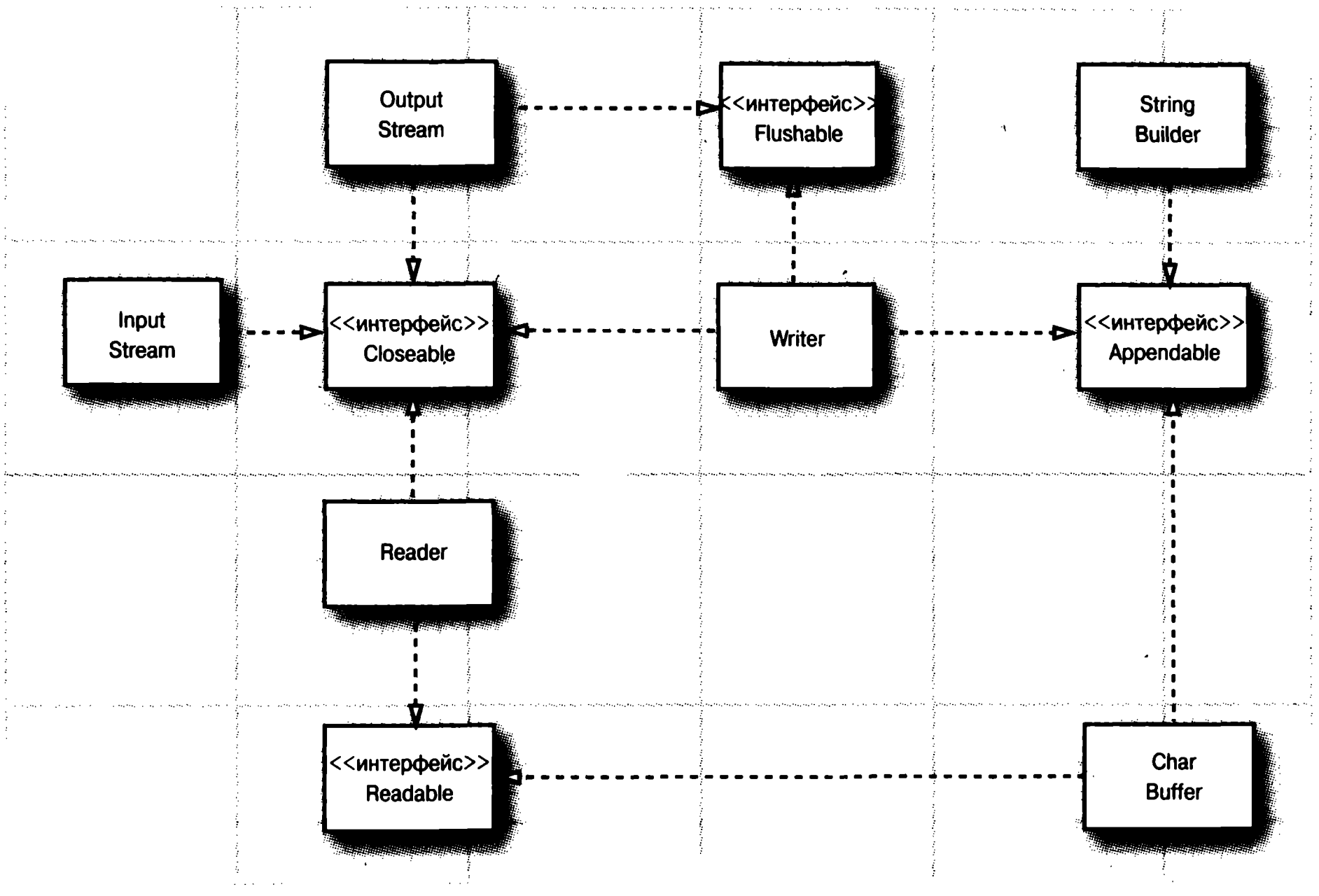


Рис. 1.3. Интерфейсы `Closeable`, `Flushable`, `Readable` и `Appendable`

НА ЗАМЕТКУ! Интерфейс `java.io.Closeable` расширяет интерфейс `java.lang.AutoCloseable`. Следовательно, в любой реализации интерфейса `Closeable` можно употреблять оператор `try` с ресурсами. А зачем вообще нужны два интерфейса? В методе `close()` из интерфейса `Closeable` генерируется только исключение типа `IOException`, тогда как в методе `AutoCloseable.close()` — исключение может быть вообще не сгенерировано.

В интерфейсе `Readable` имеется единственный метод `int read(CharBuffer cb)`, а в классе `CharBuffer` — методы для чтения и записи с последовательным и произвольным доступом. Этот класс представляет буфер в оперативной памяти или отображаемый в памяти файл. (Подробнее об этом речь пойдет далее, в разделе "Структура буфера данных".)

В интерфейсе `Appendable` имеются два приведенных ниже метода, позволяющие присоединять как отдельные символы, так и целые последовательности символов.

```
Appendable append(char c)
Appendable append(CharSequence s)
```

Интерфейс CharSequence описывает основные свойства последовательности значений типа char. Его реализуют такие классы, как String, CharBuffer, StringBuilder и StringBuffer. Из всех классов, представляющих потоки ввода-вывода в рассматриваемом здесь комплекте, только класс Writer реализует интерфейс Appendable.

java.io.Closeable 5.0

- `void close()`

  Закрывает данный поток ввода-вывода типа Closeable. Этот метод может генерировать исключение типа IOException.

java.io.Flushable 5.0

- `void flush()`

  Очищает данный поток ввода-вывода типа Flushable.

java.lang.Readable 5.0

- `int read(CharBuffer cb)`

  Пытается считать столько значений типа char в буфер *cb*, сколько в нем может их уместиться. Возвращает количество считанных значений типа char или значение **-1**, если из данного потока ввода типа Readable больше не доступно никаких значений.

java.lang.Appendable 5.0

- `Appendable append(char c)`
- `Appendable append(CharSequence cs)`

  Присоединяют указанную кодовую единицу или все кодовые единицы из заданной последовательности к данному потоку ввода-вывода типа Appendable. Возвращают ссылку this.

java.lang.CharSequence 1.4

- `char charAt(int index)`

  Возвращает кодовую единицу по заданному индексу.

- `int length()`

  Возвращает сведения об общем количестве кодовых единиц в данной последовательности.

- `CharSequence subSequence(int startIndex, int endIndex)`

  Возвращает последовательность типа `CharSequence`, состоящую только из тех кодовых единиц, которые хранятся в пределах от `startIndex` до *`endIndex`* - 1.

- `String toString()`

  Возвращает символьную строку, состоящую только из тех кодовых единиц, которые входят в данную последовательность.

Сочетание потоковых фильтров

Классы `FileInputStream` и `FileOutputStream` позволяют создавать потоки ввода-вывода и присоединять их к конкретному файлу на диске. Имя требуемого файла и полный путь к нему указываются в конструкторе соответствующего класса. Например, в приведенной ниже строке кода поиск файла под именем `employee.dat` будет производиться в текущем каталоге пользователя.

```
FileInputStream fin = new FileInputStream("employee.dat");
```

СОВЕТ. Во всех классах из пакета `java.io` относительные пути к файлам воспринимаются как команда начинать поиск с рабочего каталога пользователя, поэтому может возникнуть потребность выяснить, как именно выглядит этот каталог. Для этого достаточно вызвать метод `System.getProperty("user.dir")`.

ВНИМАНИЕ! Символ обратной косой черты является экранирующим в символьных строках Java, поэтому непременно указывайте в путях к файлам по два таких символа подряд, как, например, `C:\\Windows\\win.ini`. В Windows допускается указывать в путях к файлам одиночные символы прямой косой черты, как, например, `C:/Windows/win.ini`, поскольку в большинстве вызовов из файловой системы Windows символы прямой косой черты будут интерпретироваться как разделители файлов. Но делать это все же не рекомендуется, поскольку в режиме работы файловой системы Windows возможны изменения. Вместо этого ради переносимости программ в качестве разделителя файлов лучше употреблять именно тот символ, который принят на данной платформе. Такой символ доступен в виде строковой константы `java.io.File.separator`.

Как и в абстрактных классах `InputStream` и `OutputStream`, в классах `FileInputStream` и `FileOutputStream` поддерживаются чтение и запись только на уровне байтов. Так, из объекта `fin` в приведенной ниже строке кода можно только считать отдельные байты и массивы байтов.

```
byte b = (byte) fin.read();
```

Как поясняется в следующем разделе, имея в своем распоряжении только класс `DataInputStream`, можно было бы читать данные числовых типов следующим образом:

```
DataInputStream din = . . .;
double s = din.readDouble();
```

Но как и в классе `FileInputStream` отсутствуют методы для чтения данных числовых типов, так и в классе `DataInputStream` отсутствуют методы для извлечения данных из файла.

В Java применяется искусный механизм для разделения двух видов обязанностей. Одни потоки ввода-вывода (типа `FileInputStream` и поток ввода, возвращаемый методом `openStream()` из класса `URL`) могут извлекать байты из файлов и других более экзотических мест, а другие потоки ввода-вывода (типа `DataInputStream` и `PrintWriter`) — составлять эти байты в более полезные типы данных. Программирующему на Java остается только употреблять их в нужном сочетании. Например, для того чтобы получить возможность читать числа из файла, достаточно создать сначала объект потока ввода типа `FileInputStream`, а затем передать его конструктору класса `DataInputStream`, как показано ниже.

```
FileInputStream fin = new FileInputStream("employee.dat");
DataInputStream din = new DataInputStream(fin);
double s = din.readDouble();
```

Если снова посмотреть на рис. 1.1, то в иерархии классов для потоков ввода-вывода можно обнаружить классы `FilterInputStream` и `FilterOutputStream`. Подклассы этих классов служат для расширения функциональных возможностей базовых байтовых потоков ввода-вывода.

Благодаря вложению фильтров можно расширить функциональные возможности в еще большей степени. Например, по умолчанию потоки не буферизуются. Это означает, что каждый вызов метода `read()` приводит к запрашиванию у операционной системы выдачи очередного байта. Но намного эффективнее запрашивать сразу целые блоки данных и размещать их в буфере. Потребность использовать буферизацию и методы ввода данных в файл диктует применение следующей довольно громоздкой последовательности конструкторов:

```
DataInputStream din = new DataInputStream(
  new BufferedInputStream(
    new FileInputStream("employee.dat")));
```

Обратите внимание на то, что конструктор класса `DataInputStream` указывается последним в цепочке конструкторов. Ведь в данном случае предполагается использовать методы из класса `DataInputStream`, а в них — буферизуемый метод `read()`.

Иногда возникает потребность отслеживать промежуточные потоки ввода-вывода, когда они соединяются в цепочку. Например, при считывании входных данных нередко требуется считывать следующий байт с упреждением, чтобы выяснить, содержится ли в нем предполагаемое значение. Для этой цели в Java предоставляется класс `PushbackInputStream`, как показано ниже.

```
PushbackInputStream pbin = new PushbackInputStream(
  new BufferedInputStream(
    new FileInputStream("employee.dat")));
```

Теперь можно сначала прочитать следующий байт с упреждением:

```
int b = pbin.read();
```

а затем возвратить его обратно, если он не содержит именно то, что нужно:

```
if (b != '<') pbin.unread(b);
```

Но методы чтения `read()` и непрочтения `unread()` являются *единственными* методами, которые можно применять в потоке ввода типа `PushbackInputStream`. Так, если нужно считывать числовые данные с упреждением, то для этого потребуется ссылка не только на поток ввода типа `PushBackInputStream`, но и на поток ввода типа `DataInputStream`, как выделено ниже полужирным.

```
DataInputStream din = new DataInputStream(
  pbin = new PushbackInputStream(
    new BufferedInputStream(
      new FileInputStream("employee.dat"))));
```

Безусловно, в библиотеках потоков ввода-вывода на других языках программирования такие полезные функции, как буферизация и чтение с упреждением, обеспечиваются автоматически, и поэтому в Java необходимость сочетать для их реализации потоковые фильтры доставляет лишние хлопоты. Но в то же время возможность сочетать и подбирать классы фильтров для создания действительно полезных последовательностей потоков ввода-вывода дает немалую свободу действий. Например, используя приведенную ниже последовательность потоков ввода, можно организовать чтение чисел из архивного файла, уплотненного в формате ZIP (рис. 1.4). (Более подробно о том, как обращаться в Java с файлами, уплотненными в формате ZIP, речь пойдет в разделе "ZIP-архивы".)

```
ZipInputStream zin = new ZipInputStream(new FileInputStream("employee.zip"));
DataInputStream din = new DataInputStream(zin);
```

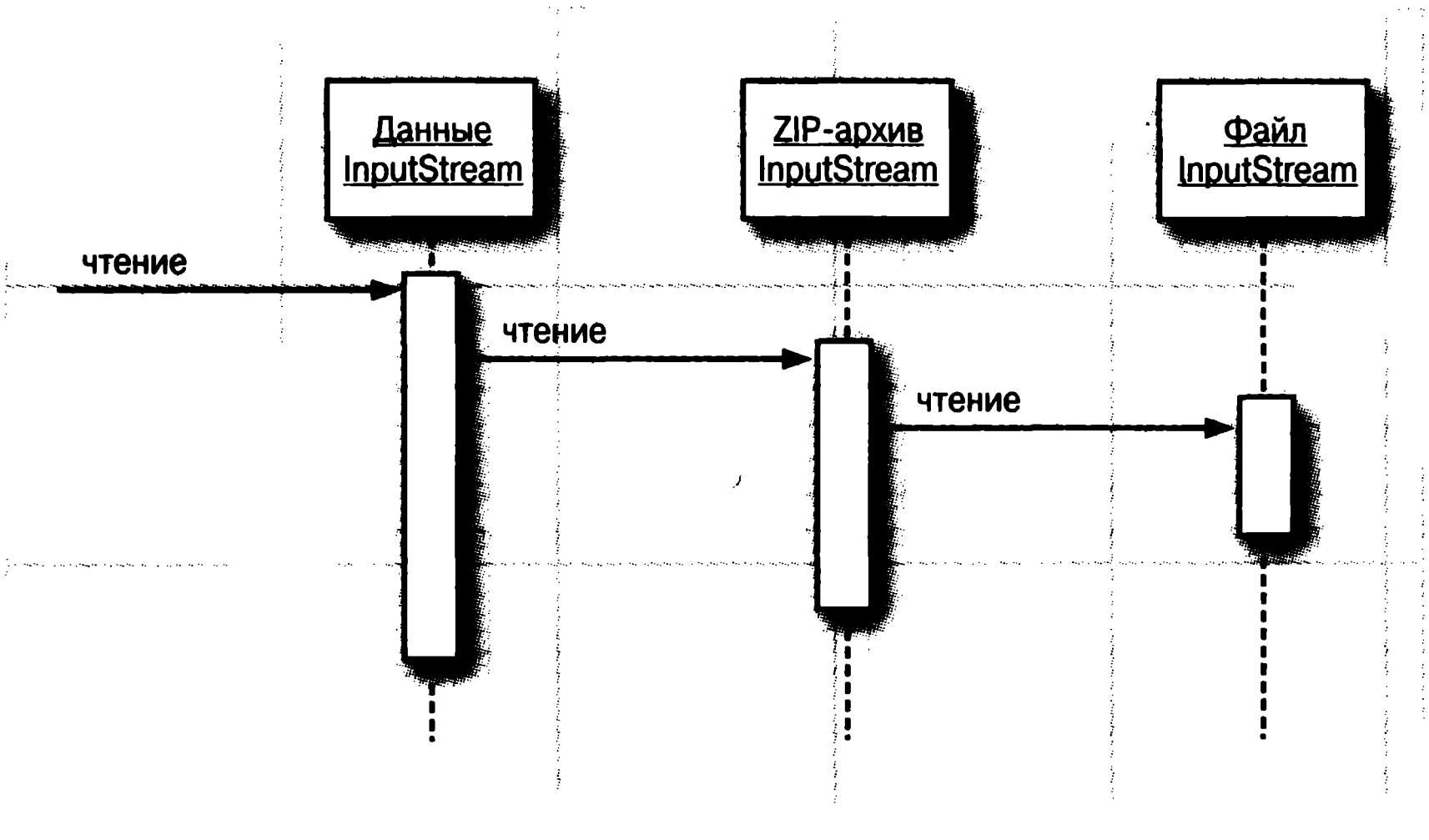


Рис. 1.4. Последовательность фильтруемых потоков

java.io.FileInputStream 1.0

- `FileInputStream(String` *name*`)`
- `FileInputStream(File` *file*`)`

Создают новый поток ввода из файла, путь к которому указывается в символьной строке `name` или в объекте *file*. (О классе `File` более подробно будет рассказываться в конце этой *главы*.) Если указываемый путь не является абсолютным, он определяется относительно рабочего каталога, который был установлен при запуске виртуальной машины.

java.io.FileOutputStream 1.0

- `FileOutputStream(String name)`
- `FileOutputStream(String name, boolean append)`
- `FileOutputStream(File file)`
- `FileOutputStream(File file, boolean append)`

Создают новый поток вывода в файл, который указывается в символьной строке `name` или в объекте *file*. (О классе `File` более подробно будет рассказываться в конце этой главы.) Если параметр `append` принимает логическое значение `true`, данные выводятся в конец файла, а если обнаружится уже существующий файл с таким же именем, он не удаляется. В противном случае удаляется любой уже существующий файл с таким же именем.

java.io.BufferedInputStream 1.0

- `BufferedInputStream(InputStream in)`

Создает буферизированный поток ввода. Такой поток способен накапливать вводимые байты без постоянного обращения к устройству ввода. Когда буфер пуст, в него считывается новый блок данных из потока.

java.io.BufferedOutputStream 1.0

- `BufferedOutputStream(OutputStream out)`

Создает буферизированный поток вывода. Такой поток способен накапливать выводимые байты без постоянного обращения к устройству вывода. Когда буфер заполнен или поток очищен, данные записываются.

java.io.PushbackInputStream 1.0

- `PushbackInputStream(InputStream in)`
- `PushbackInputStream(InputStream in, int size)`

Создают поток ввода или вывода с однобайтовым буфером для чтения с упреждением или буфером указанного размера для возврата данных обратно в поток.

- `void unread(int b)`

Возвращает байт обратно в поток, чтобы он мог быть извлечен снова при последующем вызове для чтения.

Параметры: `b` Повторно читаемый байт

Ввод-вывод текста

При сохранении данных приходится выбирать между двоичным и текстовым форматом. Так, если целое число **1234** сохраняется в двоичном формате, оно

записывается в виде следующей последовательности байтов: **00 00 04 D2** (в шестнадцатеричном представлении), а в текстовом формате — в виде символьной строки `"1234"`. Хотя ввод-вывод данных в двоичном формате осуществляется быстрее и эффективнее, тем не менее, они неудобочитаемы. Поэтому мы обсудим сначала ввод-вывод текстовых, а затем двоичных данных.

При сохранении текстовых строк приходится учитывать конкретную *кодировку символов*. Так, если используется кодировка UTF-16, символьная строка `"1234"` кодируется последовательностью байтов **00 31 00 32 00 33 00 34** (в шестнадцатеричном представлении). Но во многих прикладных программах предполагается другая кодировка содержимого текстовых файлов. Например, в кодировке по стандарту ISO 8859-1, которая чаще всего применятся в США и Западной Европе, та же самая строка будет выглядеть уже следующим образом: **31 32 33 34** (т.е. уже не будет содержать нулевые байты).

Класс `OutputStreamWriter` превращает поток вывода кодовых единиц уникода в поток байтов, применяя выбранную кодировку символов, а класс `InputStreamReader`, наоборот, — превращает поток ввода байтов, представляющих символы в какой-нибудь кодировке, в поток чтения, выдающий символы в виде кодовых единиц уникода.

В качестве примера ниже показано, как создать поток чтения вводимых данных, способный считывать с консоли набираемые на клавиатуре символы и преобразовывать их в уникод.

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

В этом потоке чтения вводимых данных предполагается, что в системе используется стандартная кодировка символов, например, ISO 8859-1 для Западной Европы. Но ничто не мешает выбрать любую другую кодировку, просто указав ее в конструкторе класса `InputStreamReader`, например, так, как показано ниже. Подробнее кодировки символов будут обсуждаться ниже, в разделе "Наборы символов".

```
InputStreamReader in =
    new InputStreamReader(new FileInputStream("kremlin.dat"), "ISO8859_5");
```

Вывод текста

Для вывода текста лучше всего подходит класс `PrintWriter`. В этом классе имеются методы для вывода символьных строк и чисел в текстовом формате. В нем имеется даже служебный конструктор для связывания потоков записи типа `PrintWriter` и `FileWriter`. Так, операторы

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

и

```
PrintWriter out = new PrintWriter(new FileWriter("employee.txt"));
```

полностью эквивалентны.

Для вывода текста в поток записи типа `PrintWriter` применяются те же самые методы `print()`, `println()` и `printf()`, что и для вывода в стандартный поток `System.out`. Эти методы можно использовать для вывода числовых данных (типа `int`, `short`, `long`, `float`, `double`), символов, логических значений типа `boolean`, символьных строк и объектов.

Рассмотрим в качестве пример следующий фрагмент кода:

```
String name = "Harry Hacker";
double salary = 75000;
out.print(name);
out.print(' ');
out.println(salary);
```

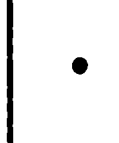
В этом фрагменте кода текстовая строка `"Harry Hacker 75000.0"` выводится в поток `out`. Эта текстовая строка затем преобразуется в байты и в конечном итоге записывается в файл `employee.txt`.

Метод `println()` добавляет к ней символ конца строки, подходящий для целевой платформы (`"\r\n"` — для Windows, `"\n"` — для Unix). А получается этот символ конца строки в результате вызова `System.getProperty("line.separator")`.

Если поток записи выводимых данных устанавливается в режим *автоматической очистки*, то при каждом вызове метода `println()` все символы, хранящиеся в буфере, отправляются по месту их назначения. (Потоки записи выводимых данных всегда снабжаются буфером.) По умолчанию режим автоматической очистки *не* включается. Его можно включать и выключать с помощью конструктора `PrintWriter(Writer out, boolean autoFlush)` следующим образом:

```
PrintWriter out = new PrintWriter(
    new FileWriter("employee.txt"), true); // автоматическая очистка
```

Методы типа `print` не генерируют исключений. Для того чтобы проверить наличие ошибок в потоке вывода, следует вызывать метод `checkError()`.



НА ЗАМЕТКУ! У программирующих на Java со стажем может возникнуть следующий вопрос: что же случилось с классом `PrintStream` и с потоком `System.out`? В версии Java 1.0 класс `PrintStream` просто усекал все символы уникода до символов в коде ASCII, отбрасывая старший байт (в то время в уникоде еще применялась 16-разрядная кодировка). Очевидно, что такой подход не обеспечивал точность и переносимость результатов, из-за чего он был усовершенствован внедрением в версии Java 1.1 потоков чтения и записи данных. Для обеспечения совместимости с существующим кодом `System.in`, `System.out` и `System.err` по-прежнему являются потоками ввода-вывода, но не для чтения и записи данных.

Класс `PrintStream` теперь способен преобразовывать внутренним образом символы уникода в стандартную кодировку хоста точно так же, как и класс `PrintWriter`. Объекты типа `PrintStream` действуют таким же образом, как и объекты типа `PrintWriter`, когда вызываются методы `print()` и `println()`, но, в отличие от объектов типа `PrintWriter`, они позволяют также выводить необработанные байты с помощью методов `write(int)` и `write(byte[])`.

java.io.PrintWriter 1.1

- `PrintWriter(Writer` *`out`*`)`
- `PrintWriter(Writer` *`out`*`, boolean` *`autoFlush`*`)`

  Создают новый экземпляр класса `PrintWriter`.

Параметры: *out* Поток записи выводимых символов

autoFlush При логическом значении `true` этого параметра методы типа `println` будут автоматически очищать буфер вывода данных (по умолчанию имеет логическое значение `false`)

- `PrintWriter(OutputStream out)`
- `PrintWriter(OutputStream out, boolean autoflush)`

Создают новый экземпляр класса `PrintWriter` из существующего экземпляра класса `OutputStream`, получая необходимый промежуточный экземпляр класса `OutputStreamWriter`.

- `PrintWriter(String filename)`
- `PrintWriter(File file)`

Создают новый экземпляр класса `PrintWriter` для записи данных в указанный файл, получая необходимый промежуточный экземпляр класса `FileWriter`.

- `void print(Object obj)`

Выводит объект в виде символьной строки, получаемой из метода `toString()`.

Параметры: *obj* Выводимый объект

- `void print(String s)`

Выводит символьную строку в виде кодовых единиц уникода.

- `void println(String s)`

Выводит символьную строку вместе с символом окончания строки. Очищает поток вывода, если он действует в режиме автоматической очистки.

- `void print(char[] s)`

Выводит все символы из указанного массива в виде кодовых единиц уникода.

- `void print(char c)`

Выводит символ в виде кодовой единицы уникода.

- `void print(int i)`
- `void print(long l)`
- `void print(float f)`
- `void print(double d)`
- `void print(boolean b)`

Выводят заданное значение в текстовом формате.

- `void printf(String format, Object... args)`

Выводит заданные значения так, как указано в строке форматирования. О том, как задается форматирующая строка, см. в главе 3 первого тома данной книги.

- `boolean checkError()`

Возвращает логическое значение `true`, если возникла ошибка при форматировании или выводе данных. При возникновении ошибки поток вывода считается испорченным, а в результате всех вызовов метода `checkError()` возвращается логическое значение `true`.

Ввод текста

Как пояснялось ранее, для записи данных в двоичном формате применяется класс `DataOutputStream`, а для их записи в текстовом формате — класс `PrintWriter`. Поэтому нетрудно догадаться, что у класса `DataInputStream` существует аналог, позволяющий считывать данные в текстовом формате. Наиболее близким его аналогом является класс `Scanner`, довольно часто применявшийся в примерах программ из первого тома данной книги. Но до выхода версии Java SE 5.0 единственным возможным вариантом для обработки вводимых текстовых данных был класс `BufferedReader`. В этом классе имеется метод `readLine()`, позволяющий читать текстовую строку. Но для этого придется объединить буферизованный поток чтения с источником вводимых данных следующим образом:

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(new FileInputStream("employee.txt"), "UTF-8"));
```

Если данных для ввода больше нет, метод `readLine()` возвращает пустое значение `null`. Следовательно, типичный цикл ввода данных выглядит следующим образом:

```
String line;
while ((line = in.readLine()) != null)
{
  сделать что-нибудь
}
```

Но в классе `BufferedReader` отсутствуют методы для чтения числовых данных. Поэтому для ввода текстовых данных рекомендуется применять не этот класс, а класс `Scanner`.

Сохранение объектов в текстовом формате

В этом разделе рассматривается пример программы, сохраняющей массив записей типа `Employee` в текстовом файле. Каждая запись сохраняется в отдельной строке. Поля экземпляра отделяются друг от друга разделителем. В качестве этого разделителя в данном примере используется знак вертикальной черты (|). (Другим распространенным разделителем является знак двоеточия (:). Любопытно, что каждый разработчик обычно пользуется своим разделителем.) Мы пока что не будем касаться того, что может произойти, если символ | встретится непосредственно в одной из сохраняемых символьных строк. Ниже приведен образец сохраняемых записей:

```
Harry Hacker|35500|1989|10|1
Carl Cracker|75000|1987|12|15
Tony Tester|38000|1990|3|15
```

Процесс записи происходит очень просто. Для этой цели применяется класс `PrintWriter`, поскольку запись выполняется в текстовый файл. Все поля просто записываются в файл, завершаясь символом |, а если это последнее поле, то комбинацией символов **\n**. Весь процесс записи совершается в теле приведенного ниже метода `writeData()`, который вводится в класс `Employee`.

```
public void writeData(PrintWriter out) throws IOException
{
  GregorianCalendar calendar = new GregorianCalendar();
  calendar.setTime(hireDay);
  out.println(name + "|"
```

```
      + salary + "|"
      + calendar.get(Calendar.YEAR) + "|"
      + (calendar.get(Calendar.MONTH) + 1) + "|"
      + calendar.get(Calendar.DAY_OF_MONTH));
}
```

Что касается считывания записей, то оно выполняется построчно с разделением полей. Для чтения каждой строки служит поток сканирования (`Scanner`), а затем полученная строка разбивается на лексемы с помощью метода `String.split()`, как показано ниже.

```
public void readData(Scanner in)
{
  String line = in.nextLine();
  String[] tokens = line.split("\\|");
  name = tokens[0];
  salary = Double.parseDouble(tokens[1]);
  int y = Integer.parseInt(tokens[2]);
  int m = Integer.parseInt(tokens[3]);
  int d = Integer.parseInt(tokens[4]);
  GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
  hireDay = calendar.getTime();
}
```

В качестве параметра метода `split()` служит регулярное выражение, описывающее разделитель. Более подробно регулярные выражения рассматриваются в конце этой главы. Оказывается, что знак вертикальной черты (`|`) имеет в регулярных выражениях специальное значение, поэтому он должен обязательно экранироваться знаком `\`, а тот, в свою очередь, еще одним знаком `\`, в результате чего получается следующее регулярное выражение: `"\\|"`.

Весь исходный код данного примера программы представлен в листинге 1.1. В приведенном ниже статическом методе сначала записывается длина массива, а затем — каждая запись.

```
void writeData(Employee[] e, PrintWriter out)
```

А в следующем статическом методе сначала считывается длина массива, а затем каждая запись:

```
Employee[] readData(BufferedReader in)
```

Оказывается, что сделать это не так-то просто, как следует из приведенного ниже фрагмента кода.

```
int n = in.nextInt();
in.nextLine(); // употребить символ новой строки
Employee[] employees = new Employee[n];
for (int i = 0; i < n; i++)
{
  employees[i] = new Employee();
  employees[i].readData(in);
}
```

Вызов метода `nextInt()` приводит к считыванию длины массива, но не завершающего символа новой строки. Этот символ должен обязательно употребляться, чтобы в методе `readData()` можно было перейти к следующей строке вводимых данных при вызове метода `nextLine()`.

Листинг 1.1. Исходный код из файла `textFile/TextFileTest.java`

```
1   package textFile;
2
3   import java.io.*;
4   import java.util.*;
5
6   /**
7    * @version 1.13 2012-05-30
8    * @author Cay Horstmann
9   */
10  public class TextFileTest
11  {
12    public static void main(String[] args) throws IOException
13    {
14      Employee[] staff = new Employee[3];
15
16      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
17      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
18      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
19
20      // сохранить записи обо всех сотрудниках в файле employee.dat
21      try (PrintWriter out = new PrintWriter("employee.dat", "UTF-8"))
22      {
23        writeData(staff, out);
24      }
25
26      // извлечь все записи в новый массив
27      try (Scanner in = new Scanner(
28            new FileInputStream("employee.dat"), "UTF-8"))
29      {
30        Employee[] newStaff = readData(in);
31        // вывести вновь прочитанные записи о сотрудниках
32        for (Employee e : newStaff)
33          System.out.println(e);
34      }
35    }
36
37    /**
38     * Записывает данные обо всех сотрудниках из
39     * массива в поток записи выводимых данных
40     * @param employees Массив записей о сотрудниках
41     * @param out Поток записи выводимых данных
42    */
43    private static void writeData(Employee[] employees, PrintWriter out)
44       throws IOException
45    {
46      // записать число сотрудников
47      out.println(employees.length);
48
49      for (Employee e : employees)
50        writeEmployee(out, e);
51    }
52
53    /**
54     * Читает записи о сотрудниках из потока сканирования в массив
55     * @param in Поток сканирования вводимых данных
56     * @return Массив записей о сотрудниках
```

```
57     */
58     private static Employee[] readData(Scanner in)
59     {
60       // извлечь размер массива
61       int n = in.nextInt();
62       in.nextLine(); // употребить символ новой строки
63
64       Employee[] employees = new Employee[n];
65       for (int i = 0; i < n; i++)
66       {
67         employees[i] = readEmployee(in);
68       }
69       return employees;
70     }
71
72     /**
73      * Записывает данные о сотрудниках в поток записи выводимых данных
74      * @param out Поток записи выводимых данных
75     */
76     public static void writeEmployee(PrintWriter out, Employee e)
77     {
78       GregorianCalendar calendar = new GregorianCalendar();
79       calendar.setTime(e.getHireDay());
80          out.println(e.getName() + "|" + e.getSalary() + "|" +
81          calendar.get(Calendar.YEAR) + "|" +
82          (calendar.get(Calendar.MONTH) + 1) + "|" +
83          calendar.get(Calendar.DAY_OF_MONTH));
84     }
85     /**
86      * Считывает данные о сотрудниках из буферизованного
87      * потока чтения вводимых данных
88      * @param in Поток чтения/сканирования вводимых данных
90     */
91     public static Employee readEmployee(Scanner in)
92     {
93       String line = in.nextLine();
94       String[] tokens = line.split("\\|");
95       String name = tokens[0];
96       double salary = Double.parseDouble(tokens[1]);
97       int year = Integer.parseInt(tokens[2]);
98       int month = Integer.parseInt(tokens[3]);
99       int day = Integer.parseInt(tokens[4]);
100      return new Employee(name, salary, year, month, day);
101    }
102 }
```

Наборы символов

Раньше в библиотеке Java вообще отсутствовало хоть какое-то систематическое обращение с наборами интернациональных символов. С появлением пакета `java.nio` в версии Java SE 1.4 это положение изменилось к лучшему, когда стал доступным класс `Charset`. (Обратите внимание на то, что буква **s** в имени этого класса пишется как строчная.)

В каждом наборе символов ставятся в соответствие последовательности двухбайтовых кодовых единиц уникода и байтовые последовательности, используемые в локальной кодировке символов. К числу наиболее распространенных относится

кодировка ISO-8859-1, которая представляет собой однобайтовую кодировку первых **256** символов уникода. Все большее значение приобретает кодировка ISO-8859-15, которая заменяет некоторые малоупотребительные (в основном в таких языках, как французский и финский) символы акцентированных букв из кодировки ISO-8859-1, а также замещает, что еще важнее, международный знак денежной единицы (¤) знаком евро (€) в кодовой точке **0xA4**. К числу других относятся кодировки символов с переменным количеством байтов, чаще всего применяемые для обозначения иероглифов японского и китайского языков.

В классе `Charset` предполагается использование только тех имен наборов символов, которые были стандартизированы организацией IANA (Internet Assigned Numbers [Names] Authority — Комитет по присвоению номеров и имен в Интернете) и перечислены в ее реестре наборов символов (`http://www.iana.org/assignments/character-sets`). Эти имена немного отличаются от тех, что применялись в предыдущих версиях. Так, набор символов ISO-8859-1 теперь носит официальное имя `ISO-8859-1`, а не `ISO8859_1`, как это было вплоть до версии Java SE 1.3.

Ради совместимости с другими условными обозначениями имен каждый набор символов может иметь ряд псевдонимов. Например, у набора символов ISO-8859-1 имеются следующие псевдонимы:

```
ISO8859-1
ISO_8859_1
ISO8859_1
ISO_8859-1
ISO_8859-1:1987
8859_1
latin1
l1
csISOLatin1
iso-ir-100
cp819
IBM819
IBM-819
819
```

Метод `aliases()` возвращает объект типа `Set` с псевдонимами. А код, требующийся для обхода псевдонимов, выглядит следующим образом:

```
Set<String> aliases = cset.aliases();
for (String alias : aliases)
  System.out.println(alias);
```

В именах наборов символов регистр не учитывается. Для доступа к набору символов типа `Charset` вызывается статический метод `forName()` с официальными именем набора символов или с одним из перечисленных выше его псевдонимов:

```
Charset cset = Charset.forName("ISO-8859-1");
```

НА ЗАМЕТКУ! Отличный источник информации по стандартному набору символов ISO 8859 имеется по адресу `http://czyborra.com/charsets/iso8859.html`.

Для того чтобы выяснить, какие наборы символов доступны в конкретной реализации, нужно вызывать метод `availableCharsets()`. Код, требующийся для выяснения имен всех доступных наборов символов, выглядит следующим образом:

```
Map<String, Charset> charsets = Charset.availableCharsets();
for (String name : charsets.keySet())
  System.out.println(name);
```

В табл. 1.1 перечислены кодировки символов, которые непременно должны присутствовать в каждой реализации Java, в табл. 1.2 — схемы кодировки, устанавливаемые вместе с комплектом JDK (Java Development Kit) по умолчанию, а в табл. 1.3 — наборы символов, устанавливаемые только в тех операционных системах, где используются неевропейские языки.

Таблица 1.1. Обязательные кодировки символов

Стандартное название кодировки	Прежнее название	Описание
US-ASCII	ASCII	Американский стандартный код для обмена информацией
ISO-8859-1	ISO8859_1	ISO 8859-1, Латиница 1
UTF-8	UTF8	8-разрядный формат преобразования символов уникода
UTF-16	UTF-16	16-разрядный формат преобразования символов уникода, где порядок следования байтов задается с помощью необязательной первоначальной отметки
UTF-16BE	UnicodeBigUnmarked	16-разрядный формат преобразования символов уникода с обратным порядком следования байтов от старшего к младшему
UTF-16LE	UnicodeLittleUnmarked	6-разрядный формат преобразования символов уникода с прямым порядком следования байтов от младшего к старшему

Таблица 1.2. Основные кодировки символов

Стандартное название кодировки	Прежнее название	Описание
ISO8859-2	ISO8859_2	ISO 8859-2, Латиница.2
ISO8859-4	ISO8859_4	ISO 8859-4, Латиница 4
ISO8859-5	ISO8859_5	ISO 8859-5, Латиница/Кириллица
ISO8859-7	ISO8859_7	ISO 8859-7, Латиница/Греческая
ISO8859-9	ISO8859_9	ISO 8859-9, Латиница 5
ISO8859-13	ISO8859_13	ISO 8859-13, Латиница 7
ISO8859-15	ISO8859_15	ISO 8859-15, Латиница 9
windows-1250	Cp1250	Восточноевропейская (Windows)
windows-1251	Cp1251	Кириллица (Windows)
windows-1252	Cp1252	Латиница 1 (Windows)
windows-1253	Cp1253	Греческая (Windows)
windows-1254	Cp1254	Турецкая (Windows)
windows-1257	Cp1257	Балтийская (Windows)

Таблица 1.3. Дополнительные кодировки символов

Стандартное название кодировки	Прежнее название	Описание
Big5	Big5	Big5, Китайская традиционная
Big5-HKSCS	Big5_HKSCS	Big5 с гонконгскими расширениями, Китайская традиционная
EUC-JP	EUC_JP	Кодировка JIS X 0201, 0208, 0212, EUC, Японская
EUC-KR	EUC_KR	Кодировка KS C 5601, EUC, Корейская
GB18030	GB18030	Китайская, упрощенная по стандарту PRC
GBK	GBK	GBK, Китайская упрощенная
ISCII91	ISCII91	Кодировка ISCII91 для индийских сценариев
ISO-2022-JP	ISO2022JP	JIS X 0201, 0208 в виде ISO 2022, Японская
ISO-2022-KR	ISO2022KR	ISO 2022 KR, Корейская
ISO8859-3	ISO8859_3	ISO 8859-3, Латиница 3
ISO8859-6	ISO8859_6	ISO 8859-6, Латиница/Арабская
ISO8859-8	ISO8859_8	ISO 8859-8, Латиница/Иврит
Shift_JIS	SJIS	Shift-JIS, Японская
TIS-620	TIS620	TIS620, Тайская
windows-1255	Cp1255	Иврит (Windows)
windows-1256	Cp1256	Арабская (Windows)
windows-1258	Cp1258	Вьетнамская (Windows)
windows-31j	MS932	Японская (Windows)
x-EUC-CN	EUC_CN	GB2312, кодировка EUC, Китайская упрощенная
x-EUC-JP-LINUX	EUC_JP_LINUX	JIS X 0201, 0208, кодировка EUC, Японская
x-EUC-TW	EUC_TW	CNS11643 (Plane 1-3), кодировка EUC, Китайская традиционная
x-MS950-HKSCS	MS950_HKSCS	Китайская традиционная (Windows) с гонконгскими расширениями
x-mswin-936	MS936	Китайская упрощенная (Windows)
x-windows-949	MS949	Корейская (Windows)
x-windows-950	MS950	Китайская традиционная (Windows)

В локальных схемах кодировки нельзя представить все символы уникода. Поэтому если какой-нибудь символ нельзя представить, то он преобразуется в знак вопроса (?). Получив набор символов, можно воспользоваться им для преобразования символьных строк, состоящих из кодовых единиц уникода, в закодированные последовательности байтов. Код, требующийся для кодирования символьной строки в Java, выглядит следующим образом:

```
String str = . . .;
ByteBuffer buffer = cset.encode(str);
byte[] bytes = buffer.array();
```

Для выполнения обратной операции, т.е. для декодирования последовательности байтов, требуется байтовый буфер. Его можно получить из байтового массива

с помощью статического метода wrap() из класса ByteBuffer. В результате выполнения метода decode() получается объект типа CharBuffer. Вызвав для него метод toString(), можно получить символьную строку, как показано ниже.

```
byte[] bytes = . . .;
ByteBuffer bbuf = ByteBuffer.wrap(bytes, offset, length);
CharBuffer cbuf = cset.decode(bbuf);
String str = cbuf.toString();
```

java.nio.charset.Charset 1.4

- `static SortedMap availableCharsets()`

  Получает все наборы символов, доступные для данной виртуальной машины. Возвращает таблицу, где ключами являются имена наборов символов, а значениями — сами наборы символов.

- `static Charset forName(String name)`

  Получает набор символов по указанному имени.

- `Set aliases()`

  Возвращает ряд псевдонимов для данного набора символов.

- `ByteBuffer encode(String str)`

  Кодирует указанную символьную строку в последовательность байтов.

- `CharBuffer decode(ByteBuffer buffer)`

  Декодирует указанную последовательность байтов. Нераспознанные при вводе символы преобразуются в символы их замены в уникоде ('\uFFFD').

java.nio.ByteBuffer 1.4

- `byte[] array()`

  Возвращает массив байтов, за управление которым отвечает данный буфер.

- `static ByteBuffer wrap(byte[] bytes)`

- `static ByteBuffer wrap(byte[] bytes, int offset, int length)`

  Возвращает буфер байтов, который управляет всем указанным массивом байтов или только некоторыми из них в заданных пределах.

java.nio.CharBuffer

- `char[] array()`

  Возвращает массив блоков кода, за управление которыми отвечает данный буфер.

- `char charAt(int index)`

  Возвращает блок кода по указанному индексу.

- `String toString()`

  Возвращает символьную строку, состоящую из блоков кода, за управление которыми отвечает данный буфер.

Чтение и запись двоичных данных

В интерфейсе `DataOutput` определяются следующие методы для записи чисел, символов, логических значений типа `boolean` или символьных строк в двоичном формате:

```
writeChars()
writeByte()
writeInt()
writeShort()
writeLong()
writeFloat()
writeDouble()
writeChar()
writeBoolean()
writeUTF()
```

Например, метод `writeInt()` всегда записывает целочисленное значение в виде 4-байтовой двоичной величины независимо от количества цифр, а метод `writeDouble()` — числовое значение с плавающей точкой типа `double` в виде 8-байтовой двоичной величины. Выводимый в итоге результат неудобочитаем, но в то же время объем требующегося пространства будет одинаковым для каждого значения заданного типа, а обратное считывание таких значений будет осуществляться намного быстрее, чем синтаксический анализ текста.

НА ЗАМЕТКУ! Для сохранения целочисленных значений и числовых значений с плавающей точкой имеются два разных способа, зависящих от используемой платформы. Допустим, имеется некоторое 4-байтовое значение типа `int`, скажем, десятичное число **1234**, или **4D2** в шестнадцатеричном представлении (**1234 = 4 × 256 + 13 × 16 + 2**). С одной стороны, оно может быть сохранено таким образом, чтобы первому байту в памяти соответствовал его самый старший байт: **00 00 04 D2**. Такой способ называется сохранением в формате с обратным порядком следования байтов от старшего к младшему. А с другой стороны, оно может быть сохранено таким образом, чтобы первым следовал самый младший байт: **D2 04 00 00**. Такой способ называется сохранением в формате с прямым порядком следования байтов от младшего к старшему. На платформах SPARC, например, применяется формат с обратным порядком следования байтов, а на платформах Pentium — формат с прямым порядком следования байтов.

Это может стать причиной серьезных осложнений. Так, данные сохраняются в файле в программах на C или C++ именно так, как это делает процессор. Вследствие этого перемещение даже простейших файлов данных с одной платформы на другую превращается в совсем не простую задачу. А в Java все значения записываются в формате с обратным порядком следования байтов от старшего к младшему независимо от типа процессора, что, соответственно, делает файлы данных в Java независящими от используемой платформы.

Метод `writeUTF()` записывает строковые данные, используя "модифицированную" версию 8-разрядного формата преобразования уникода. Вместо стандартной кодировки UTF-8 (табл. 1.4) символьные строки сначала представляются в кодировке UTF-16 (табл. 1.5), а полученный результат кодируется по правилам UTF-8. Для символов с кодом больше `0xFFFF` модифицированная кодировка выглядит по-другому. Она применяется для обеспечения обратной совместимости с виртуальными машинами, которые были созданы в те времена, когда уникод еще не мог выходить за рамки 16 битов.

Но такой "модифицированной" версией UTF-8 уже никто не пользуется, и поэтому для записи символьных строк, предназначаемых для виртуальной машины Java, например, при создании программы, генерирующей байт-коды, следует использовать только метод `writeUTF()`, а для всех остальных целей — метод `writeChars()`.

Таблица 1.4. Кодировка UTF-8

Диапазон символов	Кодировка
0...7F	$0a_6a_5a_4a_3a_2a_1a_0$
80...7FF	$110a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$
800...FFFF	$1110a_{15}a_{14}a_{13}a_{12}\ 10a_{11}a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$11110a_{20}a_{19}a_{18}\ 10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}\ 10a_{11}a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$

Таблица 1.5. Кодировка UTF-16

Диапазон символов	Кодировка
0...FFFF	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8\ a_7a_6a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$110110b_{19}b_{18}\ b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}\ 110111a_9a_8\ a_7a_6a_5a_4a_3a_2a_1a_0$, где $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

НА ЗАМЕТКУ! Более подробное описание кодировок UTF-8 и UTF-16 можно найти в документах RFC 2279 (`http://ietf.org/rfc/rfc2279.txt`) и RFC 2781 (`http://ietf.org/rfc/rfc2781.txt`) соответственно.

Для обратного чтения данных можно воспользоваться следующими методами, определенными в интерфейсе `DataInput`:

```
readInt()
readShort()
readLong()
readFloat()
readDouble()
readChar()
readBoolean()
readUTF()
```

Реализуется интерфейс `DataInput()` в классе `DataInputStream`. Читать двоичные данные из файла можно простым сочетанием потока ввода типа `DataInputStream` с нужным источником байтов, например, типа `FileInputStream`, как показано ниже.

```
DataInputStream in = new DataInputStream(new FileInputStream("employee.dat"));
```

Аналогично записывать двоичные данные можно с помощью класса `DataOutputStream`, реализующего интерфейс `DataOutput`, следующим образом:

```
DataOutputStream out = new DataOutputStream(new FileOutputStream("employee.dat"));
```

java.io.DataInput 1.0

- `boolean readBoolean()`
- `byte readByte()`
- `char readChar()`
- `double readDouble()`
- `float readFloat()`
- `int readInt()`
- `long readLong()`
- `short readShort()`

  Считывают значение заданного типа.

- `void readFully(byte[] b)`

  Считывает байты в массив *b*, устанавливая блокировку до тех пор, пока не будут считаны все байты.

  Параметры: *b* Массив, в который должны быть считаны байты

- `void readFully(byte[] b, int off, int len)`

  Считывает байты в массив b, устанавливая блокировку до тех пор, пока не будут считаны все байты.

  Параметры: *b* Массив, в который должны быть считаны байты

  off Начальное смещение данных

  len Максимальное количество считываемых байтов

- `String readUTF()`

  Считывает символьную строку в "модифицированном" формате UTF-8.

- `int skipBytes(int n)`

  Пропускает n байтов, устанавливая блокировку до тех пор, пока не будут пропущены все необходимые байты.

  Параметры: *n* Количество пропускаемых байтов

java.io.DataOutput 1.0

- `void writeBoolean(boolean b)`
- `void writeByte(int b)`
- `void writeChar(int c)`
- `void writeDouble(double d)`
- `void writeFloat(float f)`

- `void writeInt(int i)`
- `void writeLong(long l)`
- `void writeShort(int s)`

  Записывают значение заданного типа.
- `void writeChars(String s)`
- Записывает все символы из строки.
- `void writeUTF(String s)`

  Записывает символьную строку в "модифицированном" формате UTF-8

Файлы с произвольным доступом

Класс `RandomAccessFile` позволяет отыскивать или записывать данные где угодно в файле. Файлы на дисках всегда имеют возможность для произвольного доступа, тогда как потоки ввода-вывода данных из сети такой возможности не имеют. Файл с произвольным доступом может открываться только для чтения или же как для чтения, так и для записи. Требуемый режим доступа задается указанием во втором параметре конструктора символьной строки `"r"` (режим только для чтения) или символьной строки `"rw"` (режим для чтения и записи) соответственно, как показано ниже. Если существующий файл открывается как объект типа `RandomAccessFile`, он не удаляется.

```
RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
RandomAccessFile inOut = new RandomAccessFile("employee.dat", "rw");
```

У любого файла с произвольным доступом имеется так называемый *указатель файла*, обозначающий позицию следующего байта, который будет считываться или записываться. Метод `seek()` устанавливает этот указатель на произвольную байтовую позицию в файле. Этому методу в качестве аргумента может передаваться целочисленное значение типа `long` в пределах от нуля до числового значения, обозначающего длину файла в байтах. А метод `getFilePointer()` возвращает текущую позицию указателя файла.

Класс `RandomAccessFile` реализует как интерфейс `DataInput`, так и интерфейс `DataOutput`. Для чтения данных и записи данных в файл с произвольным доступом применяются методы `readInt()` и `writeInt()`, а также методы `readChar()` и `writeChar()`, обсуждавшиеся в предыдущем разделе.

Рассмотрим в качестве примера программу, сохраняющую записи о сотрудниках в файле с произвольным доступом. Все записи будут иметь одинаковые размеры. Это упрощает процесс чтения произвольной записи. Допустим, указатель файла требуется поместить на третью запись. Для этого достаточно установить указатель файла на соответствующей байтовой позиции и затем приступить к чтению нужной записи:

```
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
Employee e = new Employee();
e.readData(in);
```

Если же требуется сначала внести изменения в запись, а затем сохранить ее в том же самом месте, нужно не забыть снова установить указатель файла на начало записи следующим образом:

```
in.seek((n - 1) * RECORD_SIZE);
e.writeData(out);
```

Для определения общего количества байтов в файле служит метод `length()`. Общее количество записей определяется путем деления длины файла на размер каждой записи:

```
long nbytes = in.length(); // длина файла в байтах
int nrecords = (int) (nbytes / RECORD_SIZE);
```

Целочисленные значения и числовые значения с плавающей точкой имеют фиксированный размер в двоичном формате, тогда как с символьными строками дело обстоит немного сложнее. Для записи и чтения символьных строк фиксированного размера придется ввести два вспомогательных метода. В частности, приведенный ниже метод `writeFixedString()` записывает указанное количество кодовых единиц, отсчитывая от начала символьной строки. (Если кодовых единиц слишком мало, символьная строка дополняется нулевыми значениями.)

```
public static void writeFixedString(String s, int size, DataOutput out)
throws IOException
{
  for (int i = 0; i < size; i++)
  {
    char ch = 0;
    if (i < s.length()) ch = s.charAt(i);
    out.writeChar(ch);
  }
}
```

А приведенный ниже метод `readFixedString()` считывает символы из потока ввода (типа `InputStream`) до тех пор, пока не прочитает указанное в качестве параметра `size` количество кодовых единиц или пока не встретится символ с нулевым значением. В последнем случае все остальные нулевые значения в поле ввода пропускаются. Для повышения эффективности чтения символьных строк в этом методе используется класс `StringBuilder`.

```
public static String readFixedString(int size, DataInput in)
    throws IOException
{
  StringBuilder b = new StringBuilder(size);
  int i = 0;
  boolean more = true;
  while (more && i < size)
  {
    char ch = in.readChar();
    i++;
    if (ch == 0) more = false;
    else b.append(ch);
  }
  in.skipBytes(2 * (size - i));
  return b.toString();
}
```

Методы `writeFixedString()` и `readFixedString()` введены во вспомогательный класс `DataIO`. Для сохранения записи фиксированного размера все поля просто записываются в двоичном формате следующим образом:

```
DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
out.writeDouble(e.getSalary());
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(e.getHireDay());
out.writeInt(calendar.get(Calendar.YEAR));
out.writeInt(calendar.get(Calendar.MONTH) + 1);
out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));
```

Обратное чтение данных выполняется так же просто:

```
String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
double salary = in.readDouble();
int y = in.readInt();
int m = in.readInt();
int d = in.readInt();
```

Подсчитаем размер каждой записи. Для строк с Ф.И.О. выделим по **40** символов. Следовательно, каждая запись будет содержать по **100** байтов:

- по **40** символов, т.е. по **80** байтов, на каждые Ф.И.О.;
- по **1** числовому значению типа `double`, т.е. по **8** байтов, на размер зарплаты;
- по **3** числовых значения типа **`int`**, т.е. по **12** байтов, на дату зачисления на работу.

Программа из листинга 1.2 делает три записи в файл данных, а затем считывает их оттуда в обратном порядке. Для эффективного выполнения данного процесса требуется произвольный доступ к файлу, а в данном случае — доступ сначала к третьей записи.

Листинг 1.2. Исходный код из файла `randomAccess/RandomAccessTest.java`

```
1  package randomAccess;
2
3  import java.io.*;
4  import java.util.*;
5  /**
6   * @version 1.12 2012-05-30
7   * @author Cay Horstmann
8  */
9  public class RandomAccessTest
10 {
11   public static void main(String[] args) throws IOException
12   {
13     Employee[] staff = new Employee[3];
14
15     staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
16     staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
17     staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
18
19     try (DataOutputStream out = new DataOutputStream(
20         new FileOutputStream("employee.dat")))
21     {
22       // сохранить все записи о сотрудниках в файле employee.dat
23       for (Employee e : staff)
24         writeData(out, e);
25     }
26
```

```
27      try (RandomAccessFile in = new RandomAccessFile("employee.dat", "r"))
28      {
29        // извлечь все записи в новый архив
30
31        // определить размер массива
32        int n = (int)(in.length() / Employee.RECORD_SIZE);
33        Employee[] newStaff = new Employee[n];
34
35        // прочитать записи о сотрудниках в обратном порядке
36        for (int i = n - 1; i >= 0; i--)
37        {
38          newStaff[i] = new Employee();
39          in.seek(i * Employee.RECORD_SIZE);
40          newStaff[i] = readData(in);
41        }
42
43        // вывести вновь прочитанные записи от сотрудниках
44        for (Employee e : newStaff)
45          System.out.println(e);
46      }
47    }
48    /**
49     * Записывает сведения о сотрудниках в поток вывода данных
50     * @param out Поток вывода данных
51     * @param e Сотрудник
52    */
53    public static void writeData(DataOutput out, Employee e)
54          throws IOException
55    {
56      DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
57      out.writeDouble(e.getSalary());
58
59      GregorianCalendar calendar = new GregorianCalendar();
60      calendar.setTime(e.getHireDay());
61      out.writeInt(calendar.get(Calendar.YEAR));
62      out.writeInt(calendar.get(Calendar.MONTH) + 1);
63      out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));
64    }
65
66    /**
67     * Читает сведения о сотрудниках из потока ввода данных
68     * @param in Поток ввода данных
69     * @return Возвращает сотрудника
70    */
71    public static Employee readData(DataInput in) throws IOException
72    {
73      String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
74      double salary = in.readDouble();
75      int y = in.readInt();
76      int m = in.readInt();
77      int d = in.readInt();
78      return new Employee(name, salary, y, m - 1, d);
79    }
80  }
```

java.io.RandomAccessFile 1.0

- `RandomAccessFile(String file, String mode)`
- `RandomAccessFile(File file, String mode)`

Параметры:

file	Открываемый файл
mode	`r` — режим только для чтения; `rw` — режим чтения и записи; `rws` — режим чтения и записи с синхронным записыванием на диск данных и метаданных при каждом обновлении; `rwd` — режим чтения и записи с синхронным записыванием на диск только данных

- `long getFilePointer()`

  Возвращает сведения о текущем местоположении файлового указателя.

- `void seek(long pos)`

  Устанавливает файловый указатель на позицию `pos` от начала файла.

- `long length()`

  Возвращает длину файла в байтах.

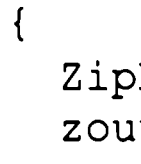
ZIP-архивы

В ZIP-архивах можно хранить один и более файл (как правило) в уплотненном формате. У каждого ZIP-архива имеется заголовок, содержащий сведения вроде имени файла или применявшегося для него алгоритма сжатия. В Java для чтения ZIP-архивов служит класс `ZipInputStream`. В каждом таком архиве всегда требуется просматривать отдельные *записи*. Метод `getNextEntry()` возвращает описывающий запись объект типа `ZipEntry`. А метод `read()` из класса `ZipInputStream` видоизменен таким образом, чтобы он возвращал значение **-1** в конце текущей записи, а не просто в конце ZIP-файла. Далее вызывается метод `closeEntry()` для перехода к чтению следующей записи. Ниже приведен типичный фрагмент кода для чтения содержимого ZIP-файла.

```
ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
  проанализировать запись entry
  прочитать содержимое архива zin
  zin.closeEntry();
}
zin.close();
```

Для чтения содержимого конкретной записи из ZIP-архива эффективнее использовать не стандартный метод `read()`, а методы какого-нибудь более развитого логически потокового фильтра. Например, для чтения текстового файла из ZIP-архива можно организовать следующий цикл:

```
Scanner in = new Scanner(zin);
while (in.hasNextLine())
  сделать что-нибудь с результатом вызова in.nextLine()
```



ВНИМАНИЕ! Не закрывайте поток ввода из ZIP-архива, прочитав единственную запись из ZIP-архива, и не передавайте этот поток методу, который может закрыть его. В противном случае вы не сможете прочитать последующие записи.

Для записи в ZIP-архив служит класс `ZipOutputStream`. В этом случае для каждой записи, которую требуется ввести в ZIP-архив, создается объект типа `ZipEntry`. Требуемое имя файла передается конструктору класса `ZipEntry`, где устанавливаются остальные параметры вроде даты создания файла и алгоритма распаковки. По желанию эти параметры могут быть переопределены. Далее вызывается метод `putNextEntry()` из класса `ZipOutputStream`, чтобы начать процесс записи нового файла. С этой целью данные из самого файла направляются в ZIP-поток, а по завершении вызывается метод `closeEntry()`. Затем описанные здесь действия выполняются повторно для всех остальных файлов, которые требуется сохранить в ZIP-архиве. Ниже приведена общая форма требующегося для этой цели кода.

```
FileOutputStream fout = new FileOutputStream("test.zip");
ZipOutputStream zout = new ZipOutputStream(fout);
для всех файлов
{
  ZipEntry ze = new ZipEntry(имя_файла);
  zout.putNextEntry(ze);
  направить данные в поток вывода zout
  zout.closeEntry();
}
zout.close();
```

НА ЗАМЕТКУ! JAR-файлы (см. главу 10 первого тома данной книги) представляют собой те же самые ZIP-файлы, но только содержат записи несколько иного вида, называемые манифестами. Для чтения и записи манифестов служат классы `JarInputStream` и `JarOutputStream`.

ZIP-потоки служат отличным примером истинного потенциала потоковой абстракции. При чтении данных, хранящихся в уплотненном виде, не нужно особенно беспокоиться о том, будут ли они распаковываться по мере запрашивания. Более того, источником байтов в ZIP-потоках совсем не обязательно должен быть именно файл: ZIP-данные могут поступать и через сетевое соединение. На самом деле при каждом чтении из архивного JAR-файла загрузчик классов аплета будет считывать и распаковывать данные именно из сети.

НА ЗАМЕТКУ! В разделе "Системы ZIP-файлов" далее в этой главе поясняется, как осуществить доступ к ZIP-архиву без специального прикладного интерфейса API, используя класс `FileSystem`, внедренный в версии Java SE 7.

java.util.zip.ZipInputStream 1.1

- `ZipInputStream(InputStream in)`

  Создает объект типа `ZipInputStream`, позволяющий распаковывать данные из указанного объекта типа `InputStream`.

- `ZipEntry getNextEntry()`

  Возвращает объект типа `ZipEntry` для следующей записи или пустое значение `null`, если записей больше нет.

- `void closeEntry()`

  Закрывает текущую открытую запись в ZIP-архиве. Далее может быть прочитана следующая запись с помощью метода `getNextEntry()`.

java.util.zip.ZipOutputStream 1.1

- `ZipOutputStream(OutputStream out)`

  Создает объект типа `ZipOutputStream`, позволяющий записывать уплотненные данные в указанный поток вывода типа `OutputStream`.

- `void putNextEntry(ZipEntry ze)`

  Записывает данные из указанной записи типа `ZipEntry` в поток вывода и устанавливает его в положение для вывода следующей порции данных. После этого данные могут быть выведены в этот поток с помощью метода `write()`.

- `void closeEntry()`

  Закрывает текущую открытую запись в ZIP-архиве. Для перехода к следующей записи используется метод `putNextEntry()`.

- `void setLevel(int level)`

  Устанавливает степень сжатия для последующих записей. По умолчанию устанавливается значение степени сжатия `Deflater.DEFAULT_COMPRESSION`. Генерирует исключение типа `IllegalArgumentException`, если заданная степень сжатия недействительна.

Параметры:	*level*	Степень сжатия от **0** (`NO_COMPRESSION`) до **9** (`BEST_COMPRESSION`)

- `void setMethod(int method)`

  Устанавливает алгоритм сжатия по умолчанию для любых записей, выводимых в поток типа `ZipOutputStream`, без указания конкретного алгоритма сжатия данных.

Параметры:	*method*	Алгоритм сжатия данных: `DEFLATED` или `STORED`

java.util.zip.ZipEntry 1.1

- `ZipEntry(String name)`

  Создает запись в Zip-архиве с заданным именем.

  Параметры: *name* Имя записи

- `long getCrc()`

  Возвращает значение контрольной суммы CRC32 для данной записи типа `ZipEntry`.

- `String getName()`

  Возвращает имя данной записи.

- `long getSize()`

  Возвращает размер данной записи без сжатия или значение **-1**, если размер записи без уплотнения неизвестен.

- `boolean isDirectory()`

  Возвращает логическое значение `true`, если данная запись является каталогом.

- `void setMethod(int method)`

  Задает алгоритм сжатия записей.

  Параметры: *method* Алгоритм сжатия данных:

  `DEFLATED` или `STORED`

- `void setSize(long size)`

  Устанавливает размер данной записи. Требуется только в том случае, если задан алгоритм сжатия данных `STORED`.

  Параметры: *size* Размер данной записи без сжатия

- `void setCrc(long crc)`

  Устанавливает контрольную сумму CRC32 для данной записи. Для вычисления этой суммы должен использоваться класс `CRC32`. Требуется только в том случае, если задан алгоритм сжатия данных `STORED`.

  Параметры: *crc* Контрольная сумма данной записи.

java.util.zip.ZipFile 1.1

- `ZipFile(String name)`
- `ZipFile(File file)`

  Создают объект типа `ZipFile` для чтения из заданной символьной строки или объекта типа `File`.

- `Enumeration entries()`

  Возвращает объект типа `Enumeration`, перечисляющий объекты типа `ZipEntry`, описывающие записи из архива типа `ZipFile`.

- `ZipEntry getEntry(String name)`

  Возвращает запись, соответствующую указанному имени, или пустое значение `null`, если такой записи не существует.

  Параметры: *name* Имя записи

- `InputStream getInputStream(ZipEntry ze)`

  Возвращает поток ввода типа `InputStream` для указанной записи.

  Параметры: *ze* Запись типа `ZipEntry` в ZIP-архиве

- `String getName()`

  Возвращает путь к данному ZIP-архиву.

Потоки ввода-вывода и сериализация объектов

Пользоваться форматом записей фиксированной длины, безусловно, удобно, если сохранять объекты одинакового типа. Но ведь объекты, создаваемые в объектно-ориентированных программах, редко бывают одного и того же типа. Например, может существовать массив `staff`, номинально представляющий собой массив записей типа `Employee`, но фактически содержащий объекты, которые, по существу, являются экземплярами какого-нибудь подкласса вроде `Manager`.

Конечно, можно было бы подобрать такой формат данных, который позволял бы сохранять подобные полиморфные коллекции, но, к счастью, в этом нет никакой необходимости. В Java поддерживается универсальный механизм, называемый *сериализацией объектов* и предоставляющий возможность записать любой объект в поток ввода-вывода, а в дальнейшем считать его снова. (Происхождение термина *сериализация* подробно объясняется далее в этой главе.)

Для сохранения объектных данных необходимо прежде всего открыть поток вывода объектов типа `ObjectOutputStream` следующим образом:

```
ObjectOutputStream out =
    new ObjectOutputStream(new FileOutputStream("employee.dat"));
```

Далее для сохранения объекта остается лишь вызвать метод `writeObject()` из класса `ObjectOutputStream`, как показано в приведенном ниже фрагменте кода.

```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
out.writeObject(harry);
out.writeObject(boss);
```

А для того чтобы считать объект обратно, нужно сначала получить объект типа `ObjectInputStream`, т.е. поток ввода объектов, следующим образом:

```
ObjectInputStream in =
    new ObjectInputStream(new FileInputStream("employee.dat"));
```

И затем извлечь объекты в том порядке, в котором они записывались, вызвав метод `readObject()`, как показано ниже.

```
Employee e1 = (Employee) in.readObject();
Employee e2 = (Employee) in.readObject();
```

Имеется, однако, одно изменение, которое нужно внести в любой класс, объекты которого требуется сохранить и восстановить в потоке ввода-вывода объектов, а именно: каждый такой класс должен обязательно реализовать интерфейс `Serializable` следующим образом:

```
class Employee implements Serializable { . . . }
```

У интерфейса `Serializable` отсутствуют методы, поэтому изменять каким-то образом свои собственные классы не нужно. В этом отношении интерфейс `Serializable` подобен интерфейсу `Cloneable`, рассматривавшемуся в главе 6 первого тома данной книги. Но для того чтобы сделать класс пригодным для клонирования, все равно требовалось переопределить метод `clone()` из класса `Object`. А для того, чтобы сделать класс пригодным для сериализации, ничего больше делать не нужно.

НА ЗАМЕТКУ! Записывать и считывать только объекты можно и с помощью методов `writeObject()` и `readObject()`. Что же касается значений простых типов, то для их ввода-вывода следует применять такие методы, как `writeInt()` и `readInt()` или `writeDouble()` и `readDouble()`. (Классы потоков ввода-вывода объектов реализуют интерфейсы `DataInput` и `DataOutput`.)

Класс `ObjectOutputStream` просматривает подспудно все поля объектов и сохраняет их содержимое. Так, при записи объекта типа `Employee` в поток вывода записывается содержимое полей Ф.И.О., даты зачисления на работу и зарплаты сотрудника.

Необходимо, однако, рассмотреть очень важный вопрос: что произойдет, если один объект совместно используется рядом других объектов? Чтобы проиллюстрировать важность данного вопроса, внесем одно небольшое изменение в класс `Manager`. В частности, допустим, что у каждого руководителя имеется свой секретарь, как показано в приведенном ниже фрагменте кода.

```
class Manager extends Employee
{
  private Employee secretary;
  . . .
}
```

Теперь каждый объект типа `Manager` будет содержать ссылку на объект типа `Employee`, описывающий секретаря. Безусловно, два руководителя вполне могут пользоваться услугами одного и того же секретаря, как показано на рис. 1.5 и в следующем фрагменте кода:

```
harry = new Employee("Harry Hacker", . . .);
Manager carl = new Manager("Carl Cracker", . . .);
carl.setSecretary(harry);
Manager tony = new Manager("Tony Tester", . . .);
tony.setSecretary(harry);
```

Сохранение такой разветвленной сети объектов оказывается непростой задачей. Разумеется, сохранять и восстанавливать адреса ячеек памяти для объектов секретарей нельзя. При повторной загрузке каждый такой объект, скорее всего, будет занимать уже совершенно другую ячейку памяти по сравнению с той, которую он занимал первоначально.

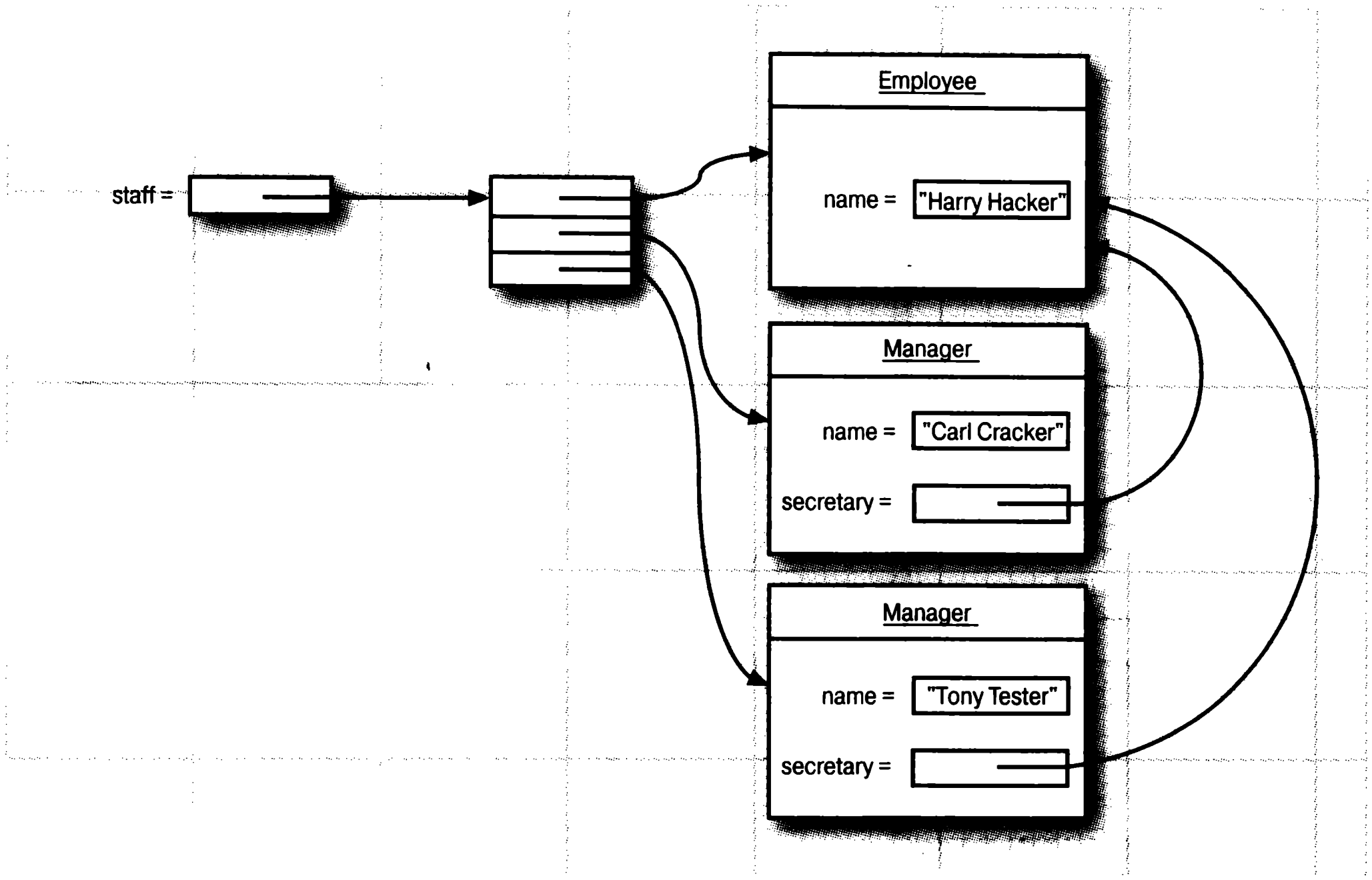


Рис. 1.5. Два руководителя могут совместно пользоваться услугами одного и того же сотрудника в качестве секретаря

Поэтому каждый такой объект сохраняется под *серийным номером*, откуда, собственно говоря, и происходит название механизма *сериализации объектов*. Ниже описывается порядок действий при сериализации объектов.

1. Серийный (т.е. порядковый) номер связывается с каждой встречающейся ссылкой на объект, как показано на рис. 1.6.
2. Если ссылка на объект встречается впервые, данные из этого объекта сохраняются в потоке ввода-вывода объектов.
3. Если же данные были ранее сохранены, просто добавляется метка "`same as previously saved object with serial number` ***x***" (совпадает с объектом, сохраненным ранее под серийным номером ***x***).

При чтении объектов обратно из потока ввода-вывода объектов порядок действий меняется на обратный.

1. Если объект впервые указывается в потоке ввода-вывода объектов, он создается и инициализируется данными из потока, а связь серийного номера с ссылкой на объект запоминается.
2. Если же встречается метка "`same as previously saved object with serial number` ***x***", то извлекается ссылка на объект по данному серийному номеру.

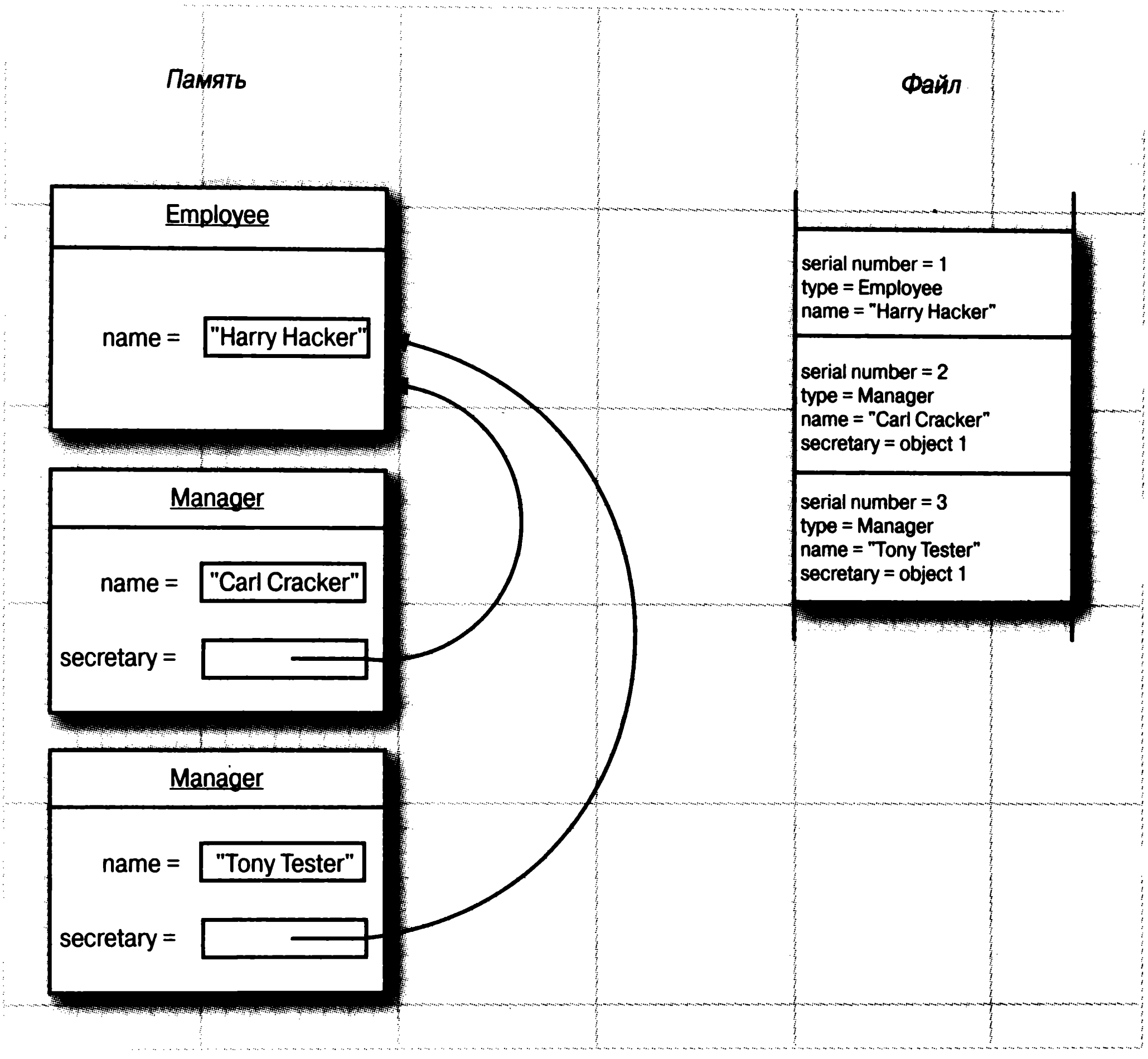


Рис. 1.6. Пример сериализации объектов

НА ЗАМЕТКУ! В этой главе демонстрируется, каким образом механизм сериализации можно применять для сохранения объектов в файле на диске и извлечения их в точном соответствии с тем, как они сохранялись. Другой очень важной областью применения данного механизма является передача коллекции объектов по сетевому соединению на другой компьютер. Исходные адреса ячеек памяти не имеют никакого значения при взаимодействии с другим процессором, как и при обращении к файлу. Заменяя адреса ячеек памяти серийными номерами, механизм сериализации делает вполне возможной перенос коллекций объектов с одной машины на другую. Более подробно такое применение механизма сериализации будет обсуждаться в главе 5 при рассмотрении способов вызова удаленных методов.

В листинге 1.3 приведен исходный код примера программы, способной сохранять и перезагружать сеть объектов типа `Employee` и `Manager` (некоторые из руководителей пользуются услугами общего сотрудника в качестве секретаря). Обратите внимание на то, что объект секретаря остается однозначным после повторной перезагрузки,

т.е. когда сотрудник, представленный объектом, хранящимся в элементе массива `newStaff[1]`, получает повышение, это отражается в полях `secretary` объектов типа `Manager`.

Листинг 1.3. Исходный код из файла `objectStream/ObjectStreamTest.java`

```
1  package objectStream;
2
3  import java.io.*;
4
5  /**
6   * @version 1.10 17 Aug 1998
7   * @author Cay Horstmann
8  */
9  class ObjectStreamTest
10 {
11   public static void main(String[] args)
12         throws IOException, ClassNotFoundException
13   {
14     Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
15     Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
16     carl.setSecretary(harry);
17     Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
18     tony.setSecretary(harry);
19
20     Employee[] staff = new Employee[3];
21     staff[0] = carl;
22     staff[1] = harry;
23     staff[2] = tony;
24
25     // сохранить записи обо всех сотрудниках в файле employee.dat
26     try (ObjectOutputStream out = new ObjectOutputStream(
27           new FileOutputStream("employee.dat")))
28     {
29       out.writeObject(staff);
30     }
31
32     try (ObjectInputStream in = new ObjectInputStream(
33           new FileInputStream("employee.dat")))
34     {
35       // извлечь все записи в новый массив
36
37       Employee[] newStaff = (Employee[]) in.readObject();
38
39       // поднять зарплату секретарю
40       newStaff[1].raiseSalary(10);
41
42       // вывести вновь прочитанные записи о сотрудниках
43       for (Employee e : newStaff)
44       System.out.println(e);
45     }
46   }
47 }
```

java.io.ObjectOutputStream 1.1

- `ObjectOutputStream(OutputStream out)`

  Создает поток вывода объектов типа `ObjectOutputStream`, чтобы объекты можно было записывать в указанный поток вывода типа `OutputStream`.

- `void writeObject(Object obj)`

  Записывает указанный объект в поток вывода объектов типа `ObjectOutputStream`. Сохраняет класс объекта, его сигнатуру и значения из любого нестатического и непереходного поля данного класса и его суперклассов.

java.io.ObjectInputStream 1.1

- `ObjectInputStream(InputStream in)`

  Создает поток ввода объектов типа `ObjectInputStream` для обратного чтения данных об объектах из указанного потока ввода типа `InputStream`.

- `Object readObject()`

  Читает объект из потока ввода объектов типа `ObjectInputStream`. В частности, читает обратно класс объекта, его сигнатуры, а также значения всех непереходных и нестатических полей этого класса и всех его суперклассов. Осуществляет десериализацию для восстановления многих ссылок на объекты.

Представление о формате файлов для сериализации объектов

При сериализации объектов их данные сохраняются в файле определенного формата. Безусловно, методами `writeObject()` и `readObject()` можно было бы воспользоваться, даже не зная, каким образом выглядит последовательность байтов, представляющая объекты в файле. Тем не менее изучение формата данных очень полезно для получения ясного представления о процессе потоковой обработки объектов. Рассматриваемый здесь материал носит до некоторой степени технический характер, поэтому если вас не интересуют подробности реализации, то можете пропустить этот раздел.

Каждый файл начинается с состоящего из двух байтов "магического числа" **AC ED**, сопровождаемого номером версии формата сериализации объектов, который в настоящее время имеет вид **00 05**. (Здесь и далее в этом разделе байты представлены шестнадцатеричными числами.) Далее в файле находится последовательность объектов, которые следуют друг за другом именно в том порядке, в каком они сохранялись. Строковые объекты сохраняются в следующем виде:

74	Двухбайтовое число, обозначающее длину файла	Символы

Например, символьная строка `"Harry"` сохраняется в файле, как показано ниже. Символы уникода из строковых объектов сохраняются в "модифицированном" формате UTF-8.

```
74 00 05 Harry
```

Вместе с объектом должен непременно сохраняться и его класс. Описание класса включает в себя следующее:

- Имя класса.
- *Однозначный идентификатор порядкового номера версии,* представляющий собой отпечаток типов полей данных и сигнатур методов.
- Набор признаков, описывающих метод сериализации.
- Описание полей данных.

Для получения отпечатка сначала каноническим способом упорядочиваются описания класса, суперкласса, интерфейсов, типов полей и сигнатур методов, а затем к этим данным применяется алгоритм так называемого безопасного хеширования (Secure Hash Algorithm — SHA).

Алгоритм SHA позволяет быстро получить "отпечаток" с большого блока данных. Этот "отпечаток" всегда представляет собой 20-байтовый пакет данных, каким бы ни был размер исходных данных. Он создается в результате выполнения над данными некоторой искусно составленной последовательности поразрядных операций, что дает практически полную уверенность, что при любом изменении данных непременно изменится и сам "отпечаток". (Подробнее с алгоритмом SHA можно ознакомиться в книге *Cryptography and Network Security: Principles and Practice, Fifth Edition* Уильяма Столингса (William Stallings; издательство Prentice Hall, 2011 г.) Но в механизме сериализации для "отпечатка" класса используются только первые **8** байтов кода SHA. И тем не менее это гарантирует, что при изменении полей данных или методов изменится и "отпечаток" класса.

При чтении объекта его "отпечаток" сравнивается с текущим "отпечатком" класса. Если они не совпадают, это означает, что после записи объекта определение класса изменилось, а следовательно, генерируется исключение. На практике классы постепенно усовершенствуются, и поэтому от прикладной программы, возможно, потребуется способность считывать прежние версии объектов. Более подробно данный вопрос обсуждается в разделе "Контроль версий" далее в этой главе.

Идентификатор класса сохраняется в следующей последовательности:

- **72**
- Двухбайтовое число, обозначающее длину имени класса
- Имя класса
- 8-байтовый отпечаток
- Однобайтовый признак
- Двухбайтовое число, обозначающее количество дескрипторов полей данных
- Дескрипторы полей данных
- **78** (конечный маркер)
- Тип суперкласса (если таковой отсутствует, то **70**)

Байт признака состоит из трех битовых масок, определенных в классе `java.io.ObjectStreamConstants` следующим образом:

```
static final byte SC_WRITE_METHOD = 1;
// в данном классе имеется метод writeObject(),
```

```
// записывающий дополнительные данные
static final byte SC_SERIALIZABLE = 2;
// в данном классе классе реализуется интерфейс Serializable
static final byte SC_EXTERNALIZABLE = 4;
// в данном классе реализуется интерфейс Externalizable
```

Более подробно интерфейс `Externalizable` обсуждается далее в этой главе, а до тех пор достаточно сказать, что в классах, реализующих интерфейс `Externalizable`, предоставляются специальные методы чтения и записи, которые принимают данные, выводимые из полей экземпляров этих классов. Рассматриваемые здесь классы реализуют интерфейс `Serializable` со значением признака **02**. Класс `java.util.Date` также реализует интерфейс `Serializable`, но помимо этого он определяет свои методы `readObject()` и `writeObject()` и имеет значение признака **03**.

Каждый дескриптор поля данных имеет следующий формат:

- Однобайтовый код типа
- Двухбайтовое число, обозначающее длину имени поля
- Имя поля
- Имя класса (если поле является объектом)

Код типа может принимать одно из следующих значений:

B	`byte`
C	`char`
D	`double`
F	`float`
I	`int`
J	`long`
L	объект
S	`short`
Z	`boolean`
[	массив

Если код типа принимает значение **L**, после имени поля следует тип поля. Символьные строки имен классов и полей не начинаются со строкового кода **74**, тогда как символьные строки типов полей начинаются с него. Для имен типов полей используется несколько иная кодировка, а именно формат собственных методов. Например, поле зарплаты из класса `Employee` кодируется следующим образом:

```
D 00 06 salary
```

В качестве примера ниже полностью показан дескриптор класса `Employee`.

```
72  00 08 Employee
    E6 D2 86 7D AE AC 18 1B 02      Отпечаток и признаки
    00 03                           Количество полей экземпляра
    D 00 06 salary                  Тип и имя поля экземпляра
    L 00 07 hireDay                 Тип и имя поля экземпляра
    74 00 10 Ljava/util/Date;       Имя класса для поля экземпляра: Date
    L 00 04 name                    Тип и имя поля экземпляра
    74 00 12 Ljava/lang/String;     Имя класса для поля экземпляра: String
```

78	Конечный маркер
70	Суперкласс отсутствует

Эти дескрипторы довольно длинные. Поэтому если дескриптор одного и того же класса снова потребуется в файле, то используется следующая сокращенная форма:

71	4-байтовый порядковый номер

Порядковый (иначе серийный) номер обозначает упоминавшийся выше явный дескриптор класса. А порядок нумерации рассматривается далее. Объект сохраняется следующим образом:

73	Дескриптор класса	Данные объекта

В качестве примера ниже показано, каким образом объект типа `Employee` сохраняется в файле.

40	E8 6A 00 00 00 00 00	Значение поля `salary` — `double`
73		Значение поля `hireDay` — новый объект
	71 00 7E 00 08	Существующий класс `java.util.Date`
	77 08 00 00 00 91 1B 4E B1 80 78	Внешнее хранилище — рассматривается ниже
74	00 0C Harry Hacker	Значение поля `name` — `String`

Как видите, в файле данных сохраняется достаточно сведений для восстановления объекта типа `Employee`. А массивы сохраняются в следующем формате:

75	Дескриптор класса	4-байтовое число, обозначающее общее количество записей	Записи

Имя класса массива в дескрипторе класса указывается в том же формате, что и в собственных методах (т.е. немного иначе, чем в формате, используемом для имен классов в других дескрипторах классов). В этом формате имена классов начинаются с буквы **L**, а завершаются точкой с запятой. Например, массив из трех объектов типа `Employee` будет начинаться следующим образом:

75		Массив
72	00 0B [LEmployee;	Новый класс, длина строки, имя класса `Employee[]`
	00 00	Количество полей экземпляра
	78	Конечный маркер
	70	Суперкласс отсутствует
	00 00 00 03	Количество записей в массиве

Обратите внимание на то, что отпечаток массива объектов типа `Employee` отличается от отпечатка самого класса `Employee`. При сохранении в выходном файле всем объектам (массивам и символьным строкам включительно), а также всем дескрипторам классов присваиваются порядковые номера. Эти номера начинаются с **00 7E 00 00**.

Как упоминалось ранее, дескриптор любого конкретного класса указывается полностью в файле только один раз, а все последующие дескрипторы просто ссылаются на него. Так, в предыдущем примере повторяющаяся ссылка на класс `Date` кодировалась следующим образом: **71 00 7E 00 08**.

Тот же самый механизм применяется и для объектов. Так, если записывается ссылка на сохраненный ранее объект, она сохраняется точно так же, т.е. в виде маркера **71**, после которого следует порядковый номер. Указывает ли ссылка на объект или же на дескриптор класса, всегда можно выяснить из контекста. И, наконец, пустая ссылка сохраняется следующим образом: **70**.

Ниже приведен снабженный комментариями результат вывода из программы `ObjectStreamTest`, представленной в листинге 1.3. По желанию вы можете запустить эту программу на выполнение, посмотреть результат вывода данных в шестнадцатеричном виде из памяти в файл `employee.dat` и сравнить его с приведенным ниже результатом. Наиболее важные строки находятся ближе к концу и демонстрируют ссылку на сохраненный ранее объект.

```
AC ED 00 05                                     Заголовок файла
75                                              Массив staff (порядковый номер #1)
   72 00 0B [LEmployee;                         Новый класс, длина строки, имя класса
                                                Employee[] (порядковый номер #0)
      FC BF 36 11 C5 91 11 C7 02                Отпечаток и признаки
      00 00                                     Количество полей экземпляра
      78                                        Конечный маркер
      70                                        Суперкласс отсутствует
      00 00 00 03                               Количество записей в массиве
73                                              staff[0] — новый объект (порядковый
                                                номер #7)
   72 00 07 Manager                             Новый класс, длина строки, имя класса
                                                (порядковый номер #2)
      36 06 AE 13 63 8F 59 B7 02                Отпечаток и признаки
      00 01                                     Количество полей данных
      L  00 09 secretary                        Тип и имя поля экземпляра
      74 00 0A LEmployee;                       Имя класса для поля экземпляра — String
                                                (порядковый номер #3)
      78                                        Конечный маркер
   72 00 08 Employee                            Суперкласс — новый класс, длина строки,
                                                имя класса (порядковый номер #4)
      E6 D2 86 7D AE AC 18 1B 02                Отпечаток и признаки
      00 03                                     Количество полей экземпляра
      D 00 06 salary                            Тип и имя поля экземпляра
      L 00 07 hireDay                           Тип и имя поля экземпляра
      74 00 10 Ljava/util/Date;                 Имя класса для поля экземпляра — String
                                                (порядковый номер #5)
      L 00 04 name                              Тип и имя поля экземпляра
      74 00 12 Ljava/lang/String;               Имя класса для поля экземпляра — String
                                                (порядковый номер #6)
      78                                        Конечный маркер
      70                                        Суперкласс отсутствует
40 F3 88 00 00 00 00 00                         Значение поля salary — double
```

73	Значение поля `secretary` — новый объект (порядковый номер #9)
72 00 0E java.util.Date	Новый класс, длина строки, имя класса (порядковый номер #8)
68 6A 81 01 4B 59 74 19 03	Отпечаток и признаки
00 00	Переменные экземпляра отсутствуют
78	Конечный маркер
77 08	Внешнее хранилище, количество байтов
00 00 00 83 E9 39 E0 00	Дата
78	Конечный маркер
74 00 0C Carl Cracker	Значение поля `name` — `String` (порядковый номер #10)
73	Значение поля `secretary` — новый объект (порядковый номер #11)
71 00 7E 00 04	Существующий класс (использовать порядковый номер #4)
40 E8 6A 00 00 00 00 00	Значение поля `salary` — `double`
73	Значение поля `hireDay` — новый объект (порядковый номер #15)
71 00 7E 00 08	Существующий класс (использовать порядковый номер #8)
77 08	Внешнее хранилище, количество байтов
00 00 00 91 1B 4E B1 80	Дата
78	Конечный маркер
74 00 0C Harry Hacker	Значение поля `name` — `String` (порядковый номер #13)
71 00 7E 00 0B	`staff[1]` — существующий объект (использовать порядковый номер #11)
73	`staff[2]` — новый объект (порядковый номер #14)
71 00 7E 00 02	Существующий класс (использовать порядковый номер #2)
40 E3 88 00 00 00 00 00	Значение поля `salary` — `double`
73	Значение поля `hireDay` — новый объект (порядковый номер #15)
71 00 7E 00 08	Существующий класс (использовать порядковый номер #8)
77 08	Внешнее хранилище, количество байтов
00 00 00 94 6D 3E EC 00 00	Дата
78	Конечный маркер
74 00 0B Tony Tester	Значение поля `name` — `String` (порядковый номер #16)
71 00 7E 00 0B	Значение поля `secretary` — существующий объект (использовать порядковый номер #11)

Разумеется, изучение этих кодов вряд ли увлекательнее чтения телефонного справочника. И хотя знать точный формат файла совсем не обязательно (если только изменение данных не преследует какую-нибудь злонамеренную цель), вам не помешает ясно представлять себе, что поток ввода-вывода содержит подробное описание всех направляемых в него объектов наряду с достаточными сведениями для восстановления как отдельных объектов, так и массивов объектов.

Самое главное — запомнить следующее.

- Поток ввода-вывода объектов содержит сведения о типах и полях данных всех входящих в него объектов.
- Для каждого объекта назначается порядковый (т.е. серийный) номер.
- Повторные вхождения того же самого объекта сохраняются в виде ссылок на его порядковый номер.

Видоизменение исходного механизма сериализации

Некоторые поля данных не должны вообще подвергаться сериализации, как, например, поля с целочисленными значениями, в которых хранятся дескрипторы файлов или дескрипторы окон, имеющие значение только для собственных методов. Такие сведения, без сомнения, становятся бесполезными при последующей повторной загрузке объекта или при его переносе на другую машину. На самом деле неверные значения в таких полях могут даже привести к аварийному завершению собственных методов. Поэтому в Java предусмотрен простой механизм, позволяющий предотвращать сериализацию подобных полей. Все, что для этого требуется, — объявить их как переходные с ключевым словом `transient`. Объявлять их подобным образом требуется и в том случае, если они не относятся к сериализуемым классам. А при сериализации объектов переходные поля всегда пропускаются.

Механизм сериализации предусматривает для отдельных классов возможность дополнять стандартный режим чтения и записи процедурами проверки правильности данных или любыми другими требующимися действиями. В сериализуемом классе могут быть определены методы с приведенными ниже сигнатурами. В таком случае поля данных больше не станут автоматически подвергаться сериализации, а вместо нее будут вызываться эти методы.

```
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
    throws IOException;
```

Рассмотрим типичный пример. В пакете `java.awt.geom` некоторые классы вроде `Point2D.Double` не являются сериализуемыми. Допустим, требуется сериализовать класс `LabeledPoint`, содержащий поля `String` и `Point2D.Double`. Для этого поле `Point2D.Double`, прежде всего, объявляется как переходное (`transient`), чтобы не возникло исключение типа `NotSerializableException`, как показано ниже.

```
public class LabeledPoint implements Serializable
{
  private String label;
  private transient Point2D.Double point;
  . . .
}
```

Далее в методе `writeObject()` сначала записывается дескриптор объекта и поле `label` типа `String`. Для этого служит специальный метод `defaultWriteObject()` из класса `ObjectOutputStream`, который может вызываться только из метода `writeObject()` сериализуемого класса. Затем координаты точки записываются в поток вывода данных с помощью стандартных методов, вызываемых из класса `DataOutput` следующим образом:

```
private void writeObject(ObjectOutputStream out)
throws IOException
{
  out.defaultWriteObject();
  out.writeDouble(point.getX());
  out.writeDouble(point.getY());
}
```

В методе `readObject()` выполняется обратный процесс:

```
private void readObject(ObjectInputStream in)
    throws IOException
{
  in.defaultReadObject();
  double x = in.readDouble();
  double y = in.readDouble();
  point = new Point2D.Double(x, y);
}
```

Еще одним примером может служить класс `java.util.Date`, предоставляющий свои собственные методы `readObject()` и `writeObject()`. Эти методы записывают дату в виде количества миллисекунд от начального момента отсчета времени (т.е. от полуночи по Гринвичу 1 января 1970 г.). Класс `Date` имеет сложное внутреннее представление, в котором хранится как объект типа `Calendar`, так и счетчик миллисекунд для оптимизации операций поиска. Состояние объекта типа `Calendar` избыточно, и поэтому не требует обязательного сохранения. Методы `readObject()` и `writeObject()` должны сохранять и загружать поля данных только своего класса, не обращая особого внимания на данные из суперкласса или каких-нибудь других классов.

Вместо того чтобы сохранять и восстанавливать данные в объектах с помощью стандартного механизма сериализации, в классе можно определить свой механизм. Для этого класс должен реализовать интерфейс `Externalizable`. Это, в свою очередь, требует, чтобы в нем были также определены два следующих метода:

```
public void readExternal(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutputStream out)
    throws IOException;
```

В отличие от методов `readObject()` и `writeObject()`, которые обсуждались в предыдущем разделе, эти методы сами полностью отвечают за сохранение и восстановление всего объекта *вместе* с данными суперкласса. Механизм сериализации просто фиксирует класс объекта в потоке ввода-вывода. При чтении объекта типа `Externalizable` поток ввода создает этот объект с помощью конструктора без аргументов и затем вызывает метод `readExternal()`. Ниже показано, как эти методы можно реализовать в классе `Employee`.

```
public void readExternal(ObjectInput s)
    throws IOException
{
  name = s.readUTF();
```

```
  salary = s.readDouble();
  hireDay = new Date(s.readLong());
}

public void writeExternal(ObjectOutput s)
    throws IOException
{
  s.writeUTF(name);
  s.writeDouble(salary);
  s.writeLong(hireDay.getTime());
}
```

ВНИМАНИЕ! В отличие от методов `readObject()` и `writeObject()`, которые являются закрытыми и могут вызываться только механизмом сериализации, методы `readExternal()` и `writeExternal()` являются открытыми. В частности, метод `readExternal()` допускает возможное изменение состояния существующего объекта.

Сериализация одноэлементных множеств и типизированных перечислений

Особого внимания требует сериализация и десериализация объектов, которые считаются единственными в своем роде. Обычно такое внимание требуется при реализации одноэлементных множеств (так называемых одиночек) и типизированных перечислений.

Так, если при написании программ на Java используется языковая конструкция `enum`, особенно беспокоиться по поводу сериализации не стоит, поскольку она будет произведена должным образом. Но что, если имеется некоторый унаследованный код, содержащий перечислимый тип вроде приведенного ниже.

```
public class Orientation
{
  public static final Orientation HORIZONTAL = new Orientation(1);
  public static final Orientation VERTICAL = new Orientation(2);
  private int value;
  private Orientation(int v) { value = v; }
}
```

Подобный подход очень широко применялся до того, как в Java были внедрены перечисления. Обратите внимание на закрытый характер конструктора. Это означает, что создать какие-нибудь другие объекты, помимо `Orientation.HORIZONTAL` и `Orientation.VERTICAL`, нельзя. А для выполнения проверки на равенство объектов можно использовать операцию `==`, как показано ниже.

```
if (orientation == Orientation.HORIZONTAL) . . .
```

Но имеется одна важная особенность, о которой не следует забывать, когда типизированное перечисление реализует интерфейс `Serializable`. Применяемый по умолчанию механизм сериализации для этого не подходит. Попробуем, например, записать значение типа `Orientation` и затем прочитать его обратно следующим образом:

```
Orientation original = Orientation.HORIZONTAL;
ObjectOutputStream out = . . .;
out.write(original);
out.close();
ObjectInputStream in = . . .;
Orientation saved = (Orientation) in.read();
```

Если затем произвести приведенную ниже проверку, она не пройдет. На самом деле переменная `saved` содержит совершенно новый объект типа `Orientation`, не равный ни одной из предопределенных констант. И несмотря на то что конструктор класса `Orientation` является закрытым, механизм сериализации все равно позволяет создавать новые объекты!

```
if (saved == Orientation.HORIZONTAL) . . .
```

В качестве выхода из этого затруднительного положения придется определить еще один специальный метод сериализации под названием `readResolve()`. В этом случае метод `readResolve()` вызывается после десериализации объекта. Он должен возвращать объект, превращаемый далее в значение, возвращаемое из метода `readObject()`. В рассматриваемом здесь примере метод `readResolve()` обследует поле `value` и возвратит соответствующую перечислимую константу, как показано ниже. Не следует, однако, забывать, что метод `readResolve()` нужно ввести во все типизированные перечисления в унаследованном коде, а также во все классы, где применяется шаблон одиночки.

```
protected Object readResolve() throws ObjectStreamException
{
  if (value == 1) return Orientation.HORIZONTAL;
  if (value == 2) return Orientation.VERTICAL;
  return null; // этого не должно произойти!
}
```

Контроль версий

Если вы применяете механизм сериализации для сохранения объектов, вам придется заранее продумать все, что может произойти при последующем усовершенствовании вашей прикладной программы. В частности, сможет ли версия 1.1 вашей программы читать старые файлы и смогут ли пользователи, по-прежнему работающие с версией 1.0, читать файлы, которые производит новая версия? Конечно, было бы совсем неплохо, если бы файлы объектов могли успешно справляться с неизбежной эволюцией классов.

На первый взгляд, подобное кажется невозможным. Ведь если определение класса изменяется хоть каким-то образом, сразу же изменяется и его "отпечаток" SHA, а, как вам должно быть уже известно, потоки ввода объектов откажутся читать объекты с отличающимися "отпечатками". Но в то же время класс может уведомить, что он совместим со своей более ранней версией. А для этого следует прежде всего получить "отпечаток" более ранней версии класса. Это можно сделать с помощью входящей в состав JDK автономной утилиты `serialver`. Например, выполнение команды

```
serialver Employee
```

даст такой результат:

```
Employee: static final long serialVersionUID = -1814239825517340645L;
```

Если же запустить утилиту `serialver` с параметром **`-show`**, на экране появится диалоговое окно с элементами ГПИ, приведенное на рис. 1.7.

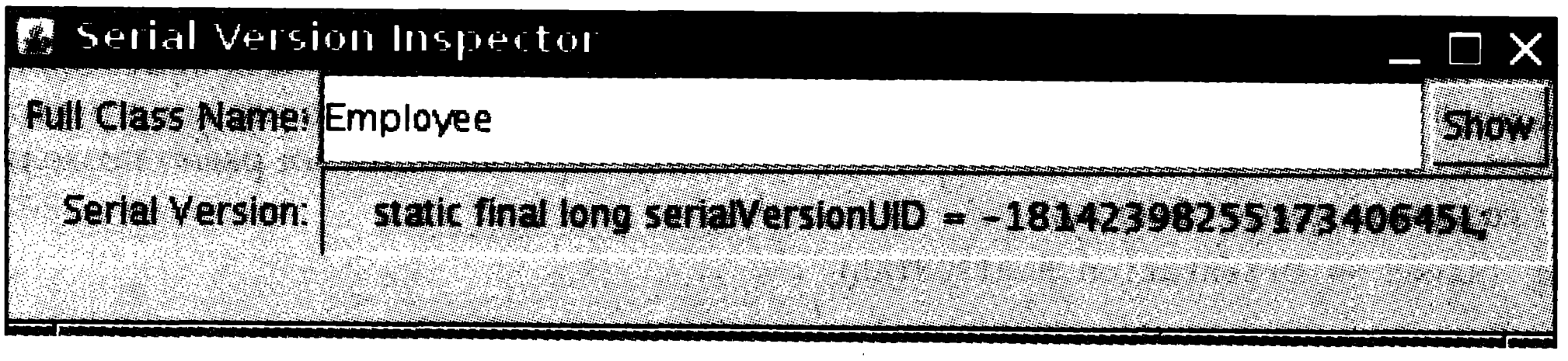


Рис. 1.7. Графическая версия утилиты `serialver`

Во всех *последующих* версиях класса константа `serialVersionUID` должна определяться точно с таким же "отпечатком", как и в исходной версии:

```
class Employee implements Serializable // версия 1.1
{
  . . .
  public static final long serialVersionUID = -1814239825517340645L;
}
```

При наличии в классе статического члена данных `serialVersionUID` он не станет вычислять "отпечаток" вручную, а просто воспользуется значением, содержащимся в этом члене. А после размещения такого статического члена в классе система сериализации будет сразу же готова к чтению разных версий объектов данного класса.

При изменении только методов класса никаких осложнений при чтении данных о новых объектах не возникнет. А если изменения произойдут в полях данных, некоторые осложнения все же могут возникнуть. Например, старый файловый объект может содержать больше или меньше полей данных, чем тот, что применяется в программе в данный момент, да и типы полей данных могут отличаться. В таком случае поток ввода объектов попытается привести потоковый объект к текущей версии класса.

Поток ввода объектов сравнит поля данных из текущей версии класса с полями данных из той версии класса, которая указана в потоке. Конечно, принимать во внимание он будет только непереходные и нестатические поля данных. При совпадении имен, но несовпадении типов полей, поток ввода объектов, естественно, не будет и пытаться преобразовывать один тип данных в другой, а следовательно, такие объекты будут считаться просто несовместимыми. При наличии у объекта в потоке таких полей данных, которых нет в текущей версии, поток ввода объектов будет просто игнорировать их. А при наличии в текущей версии таких полей, которых нет в потоковом объекте, для всех дополнительных полей будет устанавливаться соответствующее им значение по умолчанию (для объектов это пустое значение `null`, для чисел — 0, для логических значений — `false`).

Обратимся к конкретному примеру. Допустим, что ряд записей о сотрудниках был сохранен на диске с помощью исходной версии класса `Employee` (версии 1.0), а затем она была заменена версией 2.0 данного класса, в которой введено дополнительное поле данных `department`. На рис. 1.8 показано, что произойдет при чтении объекта версии 1.0 в программе, где используются объекты версии 2.0. А произойдет следующее: в поле `department` будет установлено пустое значение `null`. А на рис. 1.9 показана обратная ситуация, возникающая при чтении объекта версии 2.0 в программе, где используются объекты версии 1.0. В этом случае дополнительное поле `department` будет просто проигнорировано.

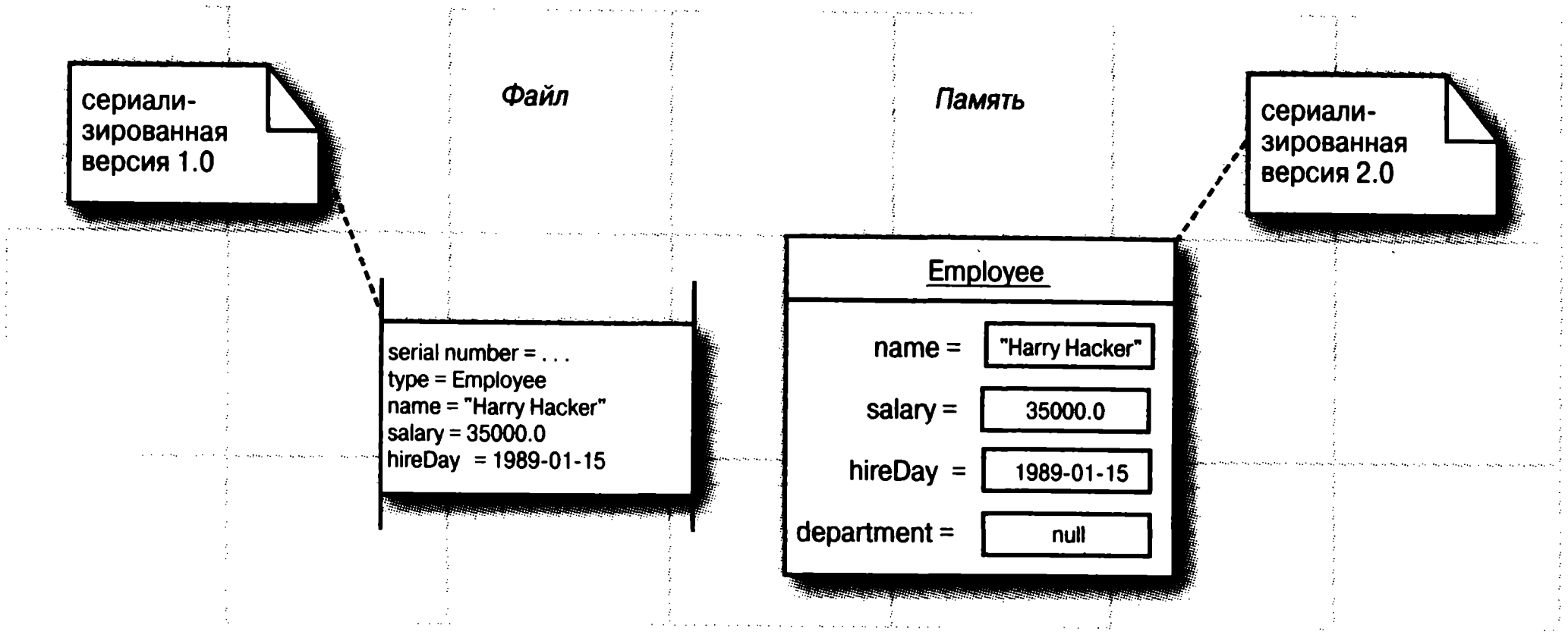


Рис. 1.8. Чтение объекта с меньшим количеством полей данных

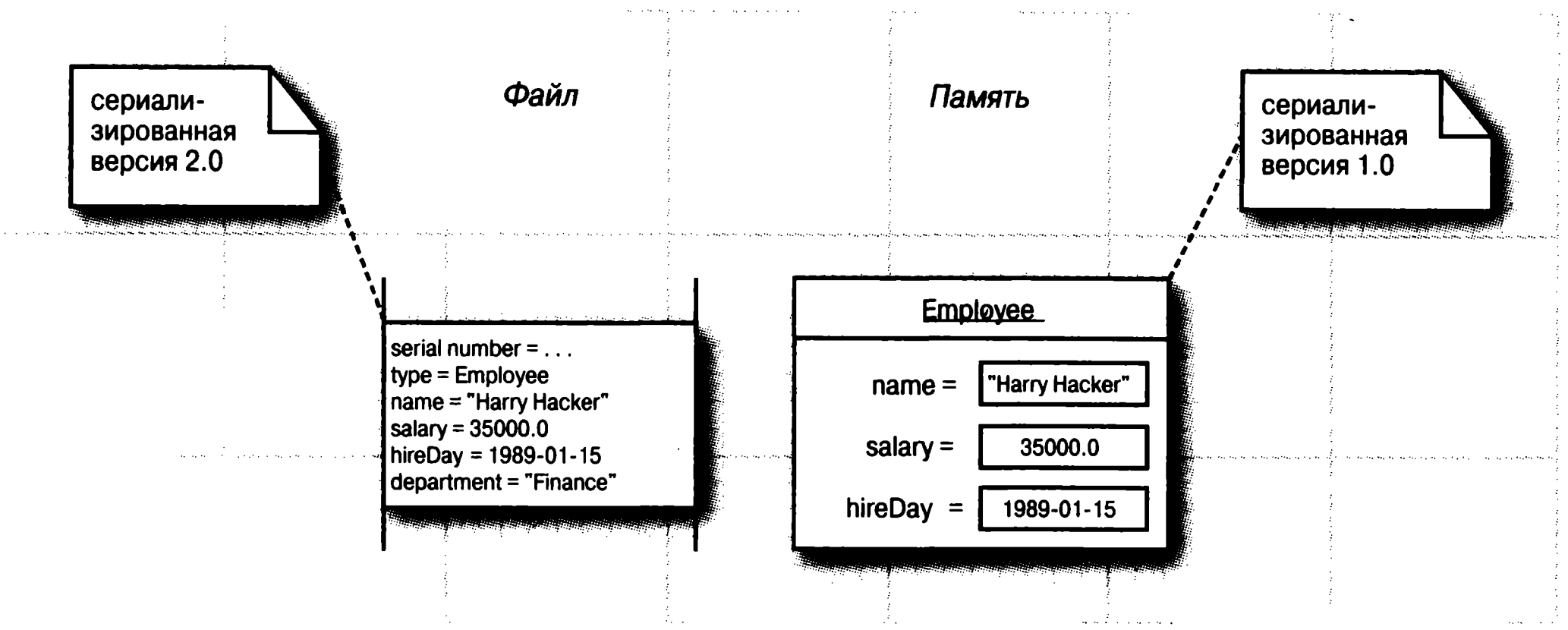


Рис. 1.9. Чтение объекта с большим количеством полей данных

Насколько безопасным окажется данный процесс? Все зависит от обстоятельств. Игнорирование поля данных кажется безвредным. Ведь у получателя все равно остаются те же самые данные, с которым он знает, как обращаться. В то же время установка в поле данных пустого значения `null` может и не быть безопасной.

Во многих классах делается немало для инициализации всех полей данных во всех конструкторах значениями, отличающимися от `null`, чтобы не предусматривать заранее в методах обработку пустых данных. Следовательно, разработчик классов должен сам решить, что лучше: реализовать дополнительный код в методе `readObject()` для устранения препятствий на пути к совместимости версий или же обеспечить достаточную надежность методов для успешной обработки пустых данных.

Применение сериализации для клонирования

У механизма сериализации имеется еще один интересный способ применения: он позволяет легко клонировать объект, при условии, что класс последнего является сериализуемым. Для этого достаточно сериализовать объект в поток вывода, а затем прочитать его обратно. В результате получится новый объект, представляющий

собой точную (полную) копию того, что уже существует. Записывать этот объект в файл совсем не обязательно. Вместо этого можно воспользоваться потоком вывода типа `ByteArrayOutputStream`, сохранив данные в байтовом массиве.

Как показывает пример программы из листинга 1.4, чтобы получить клон объекта, достаточно расширить класс `SerialCloneable`. Следует, однако, иметь в виду, что несмотря на всю изощренность такого способа, он, как правило, оказывается менее быстродействующим, чем способ клонирования, явным образом создающий новый объект и копирующий или клонирующий поля данных.

Листинг 1.4. Исходный код из файла `serialClone/SerialCloneTest.java`

```
1   package serialClone;
2
3   /**
4    * @version 1.20 17 Aug 1998
5    * @author Cay Horstmann
6   */
7
8   import java.io.*;
9   import java.util.*;
10
11  public class SerialCloneTest
12  {
13    public static void main(String[] args)
14    {
15      Employee harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
16      // клонировать объект harry
17      Employee harry2 = (Employee) harry.clone();
18
19      // видоизменить объект harry .
20      harry.raiseSalary(10);
21
22      // теперь оригинал и клон объекта harry отличаются
23      System.out.println(harry);
24      System.out.println(harry2);
25    }
26  }
27
28  /**
29   * Класс, в методе клонирования которого применяется сериализация
30  */
31  class SerialCloneable implements Cloneable, Serializable
32  {
33    public Object clone()
34    {
35      try
36      {
37        // сохранить объект в байтовом массиве
38        ByteArrayOutputStream bout = new ByteArrayOutputStream();
39        ObjectOutputStream out = new ObjectOutputStream(bout);
40        out.writeObject(this);
41        out.close();
42
43        // ввести клон объекта из байтового массива
44        ByteArrayInputStream bin =
```

```
45            new ByteArrayInputStream(bout.toByteArray());
46          ObjectInputStream in = new ObjectInputStream(bin);
47          Object ret = in.readObject();
48          in.close();
49
50          return ret;
51        }
52        catch (Exception e)
53        {
54          return null;
55        }
56      }
57  }
58
59  /**
60   * Класс Employee, переопределяемый для расширения
61   * класса SerialCloneable
62  */
63  class Employee extends SerialCloneable
64  {
65    private String name;
66    private double salary;
67    private Date hireDay;
68
69    public Employee(String n, double s, int year, int month, int day)
70    {
71      name = n;
72      salary = s;
73      GregorianCalendar calendar =
74        new GregorianCalendar(year, month - 1, day);
75        hireDay = calendar.getTime();
76    }
77
78    public String getName()
79    {
80      return name;
81    }
82
83    public double getSalary()
84    {
85      return salary;
86    }
87
88    public Date getHireDay()
89    {
90      return hireDay;
91    }
92    public void raiseSalary(double byPercent)
93    {
94      double raise = salary * byPercent / 100;
95      salary += raise;
96    }
97
98    public String toString()
99    {
100     return getClass().getName()
101       + "[name=" + name
```

```
102         + ",salary=" + salary
103         + ",hireDay=" + hireDay
104         + "]";
105    }
106 }
```

Обращение с файлами

Ранее в этой главе уже пояснялось, как читать и записывать данные в файл. Но управление файлами не ограничивается только чтением и записью. Классы `Path` и `Files` инкапсулируют все функциональные возможности, которые могут потребоваться для работы с файловой системой на машине пользователя. Так, с помощью класса `Files` можно выяснить время последнего изменения, удалить или переименовать файлы. Иными словами, классы потоков ввода-вывода служат для обращения с содержимым файлов, а классы, рассматриваемые в этом разделе, — для хранения файлов на диске.

Классы `Path` и `Files` были внедрены в версии Java SE 7. Они намного удобнее класса `File`, внедренного еще в версии JDK 1.0. Есть все основания полагать, что они найдут широкое признание у программирующих на Java. Именно поэтому они и рассматриваются в этом разделе.

Пути к файлам

Путь представляет собой последовательность имен каталогов, после которой следует (хотя и не обязательно) имя файла. Первой составляющей пути может быть корневой каталог, например / или `C:\`. В зависимости от конкретной операционной системы в пути допускаются разные составляющие. Если путь начинается с корневого каталога, он считается *абсолютным*, а иначе — *относительным*. В качестве примера в приведенном ниже фрагменте кода составляется абсолютный и относительный путь. При составлении абсолютного пути предполагается, что компьютер работает под управлением UNIX-подобной операционной системы.

```
Path absolute = Paths.get("/home", "cay");
Path relative = Paths.get("myprog", "conf", "user.properties");
```

Статический метод `Paths.get()` получает в качестве своих параметров одну или больше символьных строк, соединяя их через разделитель пути, используемый по умолчанию в данной файловой системе (/ — в UNIX-подобной файловой системе или / — в Windows). Затем в этом методе осуществляется синтаксический анализ результата, и если путь оказывается недостоверным для данной файловой системы, то генерируется исключение типа `InvalidPathException`, а иначе — получается объект типа `Path`.

В качестве параметра методу `get()` можно передать единственную символьную строку, содержащую несколько составляющих пути. Например, путь можно прочитать из конфигурационного файла следующим образом:

```
String baseDir = props.getProperty("base.dir")
  // может быть следующей символьной строкой:
  // opt/myprog или c:\Program Files\myprog
Path basePath = Paths.get(baseDir); // в переменной baseDir допускаются
                                    // разделители пути
```

НА ЗАМЕТКУ! Путь совсем не обязательно должен приводить к уже существующему файлу. Ведь он представляет собой не более, чем абстрактную последовательность имен. Как поясняется в следующем разделе, при создании файла сначала составляется путь к нему, а затем вызывается метод для создания соответствующего файла.

Зачастую пути объединяются, или *разрешаются*. Так, в результате вызова `p.resolve(q)` возвращается путь по следующим правилам.

- Если `q` — абсолютный путь, то результат равен `q`.
- В противном случае результат равен "`p` затем `q`" по правилам, принятым в данной файловой системе.

Допустим, в прикладной программе требуется найти рабочий каталог относительно заданного базового каталога, читаемого из конфигурационного файла, как было показано в предыдущем примере кода. Для этой цели служит приведенный ниже фрагмент кода.

```
Path workRelative = Paths.get("work");
Path workPath = basePath.resolve(workRelative);
```

Имеется сокращенный вариант метода `resolve()`, принимающий в качестве своего параметра символьную строку вместо пути, как показано ниже.

```
Path workPath = basePath.resolve("work");
```

Кроме того, имеется служебный метод `resolveSibling()`, разрешающий путь относительно его родителя, порождая родственный путь. Так, если `/opt/myapp/work` — путь к рабочему каталогу, то в результате следующего вызова образуется путь `/opt/myapp/temp`:

```
Path tempPath = workPath.resolveSibling("temp")
```

Противоположную методу `resolve()` функцию осуществляет метод `relativize()`. В результате вызова `p.relativize(r)` порождается путь `q`, который, в свою очередь, порождает путь `r` при своем разрешении. Например, в результате релятивизации пути `"/home/cay"` относительно пути `"/home/fred/myprog"` порождается относительный путь `"../fred/myapp"`. В данном случае две точки (`..`) обозначают родительский каталог в файловой системе.

Метод `normalize()` удаляет любые избыточные знаки `.` и `..` или иные составляющие пути, которые считаются избыточными в файловой системе. Например, в результате нормализации пути `/home/cay/../fred/./myprog` получается путь `/home/fred/myprog`. А метод `toAbsolutePath()` порождает абсолютный путь из заданного пути, начиная с коренной составляющей.

В классе `Path` имеется немало полезных методов для разделения их на составляющие и объединения с другими путями. В приведенном ниже фрагменте кода демонстрируется применение наиболее полезных методов из этого класса.

```
Path p = Paths.get("/home", "cay", "myprog.properties");
Path parent = p.getParent(); // путь /home/cay
Path file = p.getFileName(); // путь myprog.properties
Path root = p.getRoot(); // путь /
```

НА ЗАМЕТКУ! Время от времени вам, возможно, придется иметь дело с унаследованными прикладными интерфейсами API, использующими класс `File` вместо класса `Path`. Так, в классе `Path` имеется метод `toFile()`, тогда как в классе `File` — метод `toPath()`.

java.nio.file.Paths 7

- `static Path get(String first, String... more)`

  Создает путь соединением заданных символьных строк.

java.nio.file.Path 7

- `Path resolve(Path other)`
- `Path resolve(String other)`

  Если параметр *other* содержит абсолютный путь, то возвращается путь *other*, а иначе — путь, получаемый в результате объединения путей `this` и *other*.

- `Path resolveSibling(Path other)`
- `Path resolveSibling(String other)`

  Если параметр *other* содержит абсолютный путь, то возвращается путь *other*, а иначе — путь, получаемый в результате объединения родительского пути `this` и пути *other*.

- `Path relativize(Path other)`

  Возвращает относительный путь, порождающий путь *other* при разрешении пути `this`.

- `Path normalize()`

  Удаляет из пути избыточные составляющие, например, знаки `.` и `..`

- `Path toAbsolutePath()`

  Возвращает абсолютный путь, равнозначный данному пути.

- `Path getParent()`

  Возвращает родительский путь или пустое значение `null`, если у данного пути отсутствует родительский путь.

- `Path getFileName()`

  Возвращает последнюю составляющую данного пути или пустое значение `null`, если у данного пути отсутствуют составляющие.

- `Path getRoot()`

  Возвращает корневую составляющую данного пути или пустое значение `null`, если у данного пути отсутствует корневая составляющая.

- `toFile()`

  Составляет объект типа `File` из данного пути.

java.io.File 1.0

- `Path toPath() 7`

  Составляет объект типа `Path` из данного пути.

Чтение и запись данных в файлы

Класс `Files` упрощает и ускоряет выполнение типичных операций над файлами. Например, содержимое всего файла нетрудно прочитать следующим образом:

```
byte[] bytes = Files.readAllBytes(path);
```

Если же требуется прочитать содержимое файла в виде символьной строки, достаточно вызвать метод `readAllBytes()`, как показано в приведенной ниже строке кода.

```
String content = new String(bytes, charset);
```

Но если требуется получить содержимое файла в виде последовательности строк, достаточно сделать следующий вызов:

```
List<String> lines = Files.readAllLines(path, charset);
```

С другой стороны, если требуется записать в файл символьную строку, достаточно сделать приведенный ниже вызов.

```
Files.write(path, content.getBytes(charset));
```

Кроме того, в файл можно записать целый ряд строк следующим образом:

```
Files.write(path, lines);
```

Приведенные выше простые методы предназначены для обращения с текстовыми файлами умеренной длины. Если же файл крупный или двоичный, то для обращения с ним можно воспользоваться упоминавшимися ранее потоками ввода-вывода или чтения и записи данных, как показано ниже. Их служебные методы избавляют от необходимости обращаться непосредственно к классам `FileInputStream`, `FileOutputStream`, `BufferedReader` или `BufferedWriter`.

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader in = Files.newBufferedReader(path, charset);
Writer out = Files.newBufferedWriter(path, charset);
```

java.nio.file.Files 7

- `static byte[] readAllBytes(Path path)`
- `static List<String> readAllLines(Path path, Charset charset)`
- Читают содержимое файла.
- `static Path write(Path path, byte[] contents, OpenOption... options)`
- `static Path write(Path path, Iterable<? extends CharSequence> contents, OpenOption options)`

  Записывают заданное содержимое в файл и возвращают `path` как путь к нему.

- `static InputStream newInputStream(Path path, OpenOption... options)`
- `static OutputStream newOutputStream(Path path, OpenOption... options)`
- `static BufferedReader newBufferedReader(Path path, Charset charset)`
- `static BufferedWriter newBufferedWriter(Path path, Charset charset, OpenOption... options)`

  Открывают файл для чтения или записи.

Копирование, перемещение и удаление файлов

Для того чтобы скопировать файл из одного места в другое, достаточно сделать следующий вызов:

```
Files.copy(fromPath, toPath);
```

А для того чтобы переместить файл, т.е. сделать его копию и удалить оригинал, следует сделать приведенный ниже вызов.

```
Files.move(fromPath, toPath);
```

Исход операции копирования или перемещения файлов окажется неудачным, если целевой файл существует. Если же требуется перезаписать целевой файл, при вызове соответствующего метода следует указать дополнительный параметр `REPLACE_EXISTING`, а если требуется скопировать все атрибуты файла, — дополнительный параметр `COPY_ATTRIBUTES`. Кроме того, можно указать оба дополнительных параметра следующим образом:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.COPY_ATTRIBUTES);
```

Имеется также возможность сделать операцию перемещения атомарной. Этим гарантируется, что операция перемещения завершится успешно или что источник данных продолжает существовать. В приведенной ниже строке кода показано, каким образом для этой цели используется дополнительный параметр `ATOMIC_MOVE`.

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

И наконец, для удаления файла достаточно вызвать следующий метод:

```
Files.delete(path);
```

Этот метод генерирует исключение, если файл не существует. Поэтому вместо него, возможно, придется вызвать приведенный ниже метод. Оба метода можно также использовать для удаления пустого каталога.

```
boolean deleted = Files.deleteIfExists(path);
```

java.nio.file.Files 7

- `static Path copy(Path from, Path to, CopyOption... options)`
- `staticPath move(Path from, Path to, CopyOption... options)`

  Копируют или перемещают файл из исходного места *from* в заданное целевое место *to*, возвращая последнее.

- `static void delete(Path path)`
- `static boolean deleteIfExists(Path path)`

  Удаляют заданный файл или пустой каталог. Первый метод генерирует исключение, если заданный файл или каталог не существует. А второй метод возвращает в этом случае логическое значение `false`.

Создание файлов и каталогов

Для того чтобы создать новый каталог, достаточно сделать следующий вызов:

```
Files.createDirectory(path);
```

А для того чтобы создать пустой файл, следует сделать приведенный ниже вызов.

```
Files.createFile(path);
```

Если файл уже существует, то в результате данного вызова генерируется исключение. Поэтому, выполняя операцию создания файла, следует проверять ее атомарность и факт существования файла. Если файл не существует, он создается, прежде чем у кого-нибудь другого появится возможность сделать то же самое.

Для создания временного файла или каталога в указанном месте файловой системы имеются служебные методы. Примеры их применения демонстрируются в приведенном ниже фрагменте кода, где `dir` — это объект типа `Path`, а `prefix`/`suffix` — символьные строки, которые могут быть пустыми (`null`). Например, в результате вызова `Files.createTempFile(null, ".txt")` может быть возвращен путь `/tmp/12344055223648371 94.txt`.

```
Path newPath = Files.createTempFile(dir, prefix, suffix);
Path newPath = Files.createTempFile(prefix, suffix);
Path newPath = Files.createTempDirectory(dir, prefix);
Path newPath = Files.createTempDirectory(prefix);
```

При создании файла или каталога можно также указать его атрибуты, в том числе владельцев или права доступа. Но конкретные подробности зависят от применяемой файловой системы, и поэтому они здесь не рассматриваются.

java.nio.file.Files 7

- `static Path createFile(Path path, FileAttribute<?>... attrs)`
- `static Path createDirectory(Path path, FileAttribute<?>... attrs)`
- `static Path createDirectories(Path path, FileAttribute<?>... attrs)`

  Создают файл или каталог. В частности, метод `createDirectories()` создает также любые промежуточные каталоги.

- `static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)`
- `static Path createTempFile(Path parentDir, String prefix, String suffix, FileAttribute<?>...attrs)`

- `static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)`
- `static Path createTempDirectory(Path parentDir, String prefix, FileAttribute<?>... attrs)`

  Создают временный файл или каталог в месте, пригодном для хранения временных файлов, или же в заданном родительском каталоге. Возвращают путь к созданному файлу или каталогу.

Получение сведений о файлах

Ниже перечислены методы, возвращающие логическое значение для проверки свойства пути.

- `exists()`
- `isHidden()`
- `isReadable()`, `isWritable()`, `isExecutable()`
- `isRegularFile()`, `isDirectory()`, `isSymbolicLink()`

Метод `size()` возвращает количество байт в файле, как показано в приведенной ниже строке кода. А метод `getOwner()` возвращает владельца файла в виде экземпляра класса `java.nio.file.attribute.UserPrincipal`.

```
long fileSize = Files.size(path);
```

Все файловые системы уведомляют об основных атрибутах, инкапсулированных в интерфейсе `BasicFileAttributes`. Они частично перекрывают прочие сведения о файлах. Ниже перечислены основные атрибуты файлов.

- Моменты времени, когда файл был создан, последний раз открывался и видоизменялся, в виде экземпляров класса `java.nio.file.attribute.FileTime`.
- Является ли файл обычным, каталогом, символической ссылкой или ни одним из перечисленного.
- Размер файла.
- Файловый ключ — объект некоторого класса, характерный для применяемой файловой системы и способный (или не способный) однозначно определять файл.

Для получения перечисленных выше атрибутов файлов достаточно сделать следующий вызов:

```
BasicFileAttributes attributes =
    files.readAttributes(path, BasicFileAttributes.class);
```

Если заранее известно, что пользовательская файловая система соответствует стандарту POSIX, в таком случае можно получить экземпляр класса `PosixFileAttributes` следующим образом:

```
PosixFileAttributes attributes =
    files.readAttributes(path, PosixFileAttributes.class);
```

А далее можно определить группового или индивидуального владельца файла, а также права на групповой или глобальный доступ к нему. Мы не будем здесь

вдаваться в подробности этого процесса, поскольку бо́льшая часть сведений о файлах не переносится из одной операционной системы в другую.

java.nio.file.Files 7

- `static boolean exists(Path path)`
- `static boolean isHidden(Path path)`
- `static boolean isReadable(Path path)`
- `static boolean isWritable(Path path)`
- `static boolean isExecutable(Path path)`
- `static boolean isRegularFile(Path path)`
- `static boolean isDirectory(Path path)`
- `static boolean isSymbolicLink(Path path)`

  Проверяют заданное свойство файла по указанному пути.
- `static long size(Path path)`

  Получает размер файла в байтах.
- `A readAttributes(Path path, Class<A> type, LinkOption... options)`

  Читает файловые атрибуты типа `A`.

java.nio.file.attribute.BasicFileAttributes 7

- `FileTime creationTime()`
- `FileTime lastAccessTime()`
- `FileTime lastModifiedTime()`
- `boolean isRegularFile()`
- `boolean isDirectory()`
- `boolean isSymbolicLink()`
- `long size()`
- `Object fileKey()`

  Получают запрашиваемый атрибут.

Перемещение по файлам и каталогам

В прежнем классе `File` имелся метод для получения массива всех файлов в каталоге, но это приводило к снижению производительности при перемещении по каталогам, содержащим огромное количество файлов. Именно поэтому в классе `Files` появился метод, предоставляющий объект типа `Iterable` вместо массива. В приведенном ниже примере кода показано, как пользоваться этим методом.

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir))
{
  for (Path entry : entries)
```

```
    обработать entries
}
```

Блок оператора try с ресурсами обеспечивает надлежащее закрытие потока ввода из каталога. Определенного порядка обращения к элементам каталога не существует. Файлы можно отфильтровать по глобальному шаблону, как показано ниже. Все глобальные шаблоны перечислены в табл. 1.6.

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir, "*.java"))
```

Таблица 1.6. Глобальные шаблоны

Шаблон	Описание	Пример применения
*	Совпадает со всеми нулями и дополнительными символами в составляющей пути	Шаблон `*.java` совпадает со всеми файлами исходного кода на Java в текущем каталоге
**	Совпадает со всеми нулями и дополнительными символами, пересекая границы каталогов	Шаблон `**.java` совпадает со всеми файлами исходного кода на Java в любом подкаталоге
?	Совпадает с одним символом	Шаблон `????.java` совпадает со всеми файлами исходного кода на Java, имена которых состоят из четырех символов, не считая расширения
[...]	Совпадает с рядом символов, указываемых через дефис или со знаком отрицания: `[0-9]` или `[!0-9]` соответственно	Шаблон `Test[0-9A-F].java` совпадает с файлом `Testx.java`, где *x* — одна шестнадцатеричная цифра
{...}	Совпадает с альтернативными символами, разделенными запятыми	Шаблон `*.{java,class}` совпадает со всеми файлами исходного кода и классов на Java
\	Экранирует любые перечисленные выше шаблоны	Шаблон `*\**` совпадает со всеми файлами, в именах которых содержится знак *

ВНИМАНИЕ! Используя синтаксис глобальных шаблонов в Windows, экранируйте символы обратной косой черты дважды: один раз — в синтаксисе глобального шаблона, а другой раз — в синтаксисе символьной строки: `Files.newDirectoryStream(dir, "C:\\\\")`.

Если требуется обратиться к производным от каталога, следует вызвать метод `walkFileTree()` с объектом типа `FileVisitor` в качестве параметра. Этот объект уведомляется в следующих случаях.

- Когда встречается файл или каталог: `FileVisitResult visitFile(T path, BasicFileAttributes attrs)`.
- До обработки каталога: `FileVisitResult preVisitDirectory(T dir, IOException ex)`.
- После обработки каталога: `FileVisitResult postVisitDirectory(T dir, IOException ex)`.

- Когда возникает ошибка в связи с попыткой обратиться к файлу или каталогу, например, открыть каталог без надлежащих прав доступа: `FileVisitResult visitFileFailed(T path, IOException ex)`.

В каждом случае можно указать следующее.

- Продолжить обращение к следующему файлу: `FileVisitResult.CONTINUE`.
- Продолжить обход, но не обращаясь к элементам каталога: `FileVisitResult.SKIP_SUBTREE`.
- Продолжить обход, но не обращаясь к элементам каталога, родственным данному файлу: `FileVisitResult.SKIP_SIBLINGS`.
- Завершить обход: `FileVisitResult.TERMINATE`.

Если любой из методов генерирует исключение, обход каталогов завершается и данное исключение генерируется далее в методе `walkFileTree()`.

НА ЗАМЕТКУ! Несмотря на то что интерфейс `FileVisitor` относится к обобщенному типу, вам вряд придется пользоваться какой-нибудь другой его разновидностью, кроме `FileVisitor<Path>`. Метод `walkFileTree()` охотно принимает в качестве своего параметра объект обобщенного типа `FileVisitor<? super Path>`, но у типа `Path` не так уж и много супертипов.

Служебный класс `SimpleFileVisitor` реализует интерфейс `FileVisitor`. Все его методы, кроме `visitFileFailed()`, ничего особенного не делают, лишь продолжая выполнение. А метод `visitFileFailed()` генерирует исключение, возникающее в результате сбоя, а следовательно, прекращающее обращение к файлу. В качестве примера в приведенном ниже фрагменте кода демонстрируется, каким образом выводятся все подкаталоги из заданного каталога.

```
Files.walkFileTree(dir, new SimpleFileVisitor<Path>()
  {
    public FileVisitResult visitFile(Path path, BasicFileAttributes attrs)
          throws IOException
    {
      if (attrs.isDirectory())
      System.out.println(path);
      return FileVisitResult.CONTINUE;
    }
    public FileVisitResult visitFileFailed(Path path, IOException exc)
          throws IOException
    {
      return FileVisitResult.CONTINUE;
    }
  });
```

Следует, однако, иметь в виду, что метод `visitFileFailed()` придется переопределить, иначе обращение к файлам сразу же завершится аварийно, как только встретится каталог, который не разрешается открывать. Следует также иметь в виду, атрибуты пути передаются в качестве параметра. Методу `walkFileTree()` уже пришлось обратиться к операционной системе для получения атрибутов, поскольку ему нужно каким-то образом отличать файлы от каталогов. Благодаря этому исключается необходимость делать еще один вызов.

Остальные методы из интерфейса `FileVisitor` оказываются полезными в том случае, если требуется выполнить служебные операции для входа и выхода из каталога. Так, если создается копия дерева каталогов, необходимо скопировать текущий каталог, прежде чем вводить в него копируемые файлы. А при удалении дерева каталогов, нужно удалить текущий каталог после того, как из него будут удалены все файлы.

java.nio.file.Files 7

- `DirectoryStream<Path> newDirectoryStream(Path path)`
- `DirectoryStream<Path> newDirectoryStream(Path path, String glob)`

  Получают итератор для обхода всех файлов и каталогов в данном каталоге. Второй метод принимает только те элементы файловой системы, которые совпадают с заданным глобальным шаблоном.
- `Path walkFileTree(Path start, FileVisitor<? super Path> visitor)`

  Обходит все порожденные составляющие заданного пути, применяя к ним указанный порядок обращения.

java.nio.file.SimpleFileVisitor<T> 7

- `FileVisitResult visitFile(T path, BasicFileAttributes attrs)`

  Вызывается при обращении к файлу или каталогу. Возвращает одно из значений констант `CONTINUE`, `SKIP_SUBTREE`, `SKIP_SIBLINGS` или `TERMINATE`. В исходной реализации по умолчанию ничего особенного не делает и только продолжает выполнение.
- `FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)`
- `FileVisitResult postVisitDirectory(T dir, BasicFileAttributes attrs)`

  Вызываются до и после обращения к каталогу. В исходной реализации по умолчанию ничего особенного не делает и только продолжает выполнение.
- `FileVisitResult visitFileFailed(T path, IOException exc)`

  Вызывается, если генерируется исключение в связи с попыткой получить сведения о заданном файле. В исходной реализации по умолчанию повторно генерирует исключение, прекращающее обращение к файлу с данным исключением. Этот метод следует переопределить, чтобы продолжить выполнение.

Системы ZIP-файлов

В классе `Paths` осуществляется поиск по путям в исходной файловой системе, т.е. там, где файлы хранятся на локальном диске пользовательского компьютера. Но ведь могут быть и другие файловые системы. К числу наиболее употребительных относится *система ZIP-файлов*. Если `zipname` — это имя ZIP-файла, то в результате следующего вызова устанавливается файловая система, содержащая все файлы в ZIP-архиве:

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
```

Если же известно имя файла, его нетрудно скопировать из такого архива следующим образом:

```
Files.copy(fs.getPath(sourceName), targetPath);
```

Здесь метод `fs.getPath()` выполняет функции, аналогичные методу `Paths.get()`, для произвольной файловой системы. Для того чтобы перечислить все файлы в ZIP-архиве, следует обойти дерево файлов, как показано в приведенном ниже фрагменте кода. Это намного удобнее, чем пользоваться прикладным интерфейсом API, описанным в разделе "ZIP-архивы" ранее в этой главе, где требовался новый ряд классов только для обращения с ZIP-архивами.

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
Files.walkFileTree(fs.getPath("/"), new SimpleFileVisitor<Path>()
  {
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
          throws IOException
    {
      System.out.println(file);
      return FileVisitResult.CONTINUE;
    }
});
```

java.nio.file.FileSystems 7

- `static FileSystem newFileSystem(Path` *`path`*`, ClassLoader` *`loader`*`)`
  Обходит все установленные поставщики файловых систем, а также файловые системы, которые способен загрузить указанный загрузчик классов, если не указано пустое значение `null` параметра *`loader`*. По умолчанию предоставляется поставщик для систем ZIP-файлов, принимающий файлы с расширением `.zip` или `.jar`.

java.nio.file.FileSystem 7

- `static Path getPath(String` *`first`*`, String... ` *`more`*`)`
  Составляет путь, объединяя заданные символьные строки.

Отображаемые в памяти файлы

В большинстве операционных систем можно выгодно пользоваться реализациями виртуальной памяти для отображения файла или только определенной его части в оперативной памяти. В этом случае доступ к файлу можно получить так, как будто он хранится в виде массива в оперативной памяти, что намного быстрее, чем при выполнении традиционных операций над файлами.

В конце этого раздела приведен пример программы, вычисляющей контрольную сумму CRC32 для файла с использованием традиционной операции ввода из файла и отображаемого в памяти файла. В табл. 1.7 перечислены временные характеристики, полученные на одном компьютере при вычислении с помощью этой программы контрольной суммы для файла `rt.jar` объемом 37 Мбайт, входящего в состав JDK и расположенного в каталоге `jre/lib`.

Таблица 1.7. Временные характеристики некоторых операций над файлами

Средство выполнения операции	Время (в секундах)
Простой поток ввода	110
Буферизованный поток ввода	9,9
Файл с произвольным доступом	162
Отображаемый в памяти файл	7,2

Как следует из табл. 1.7, на отдельно взятой машине при отображении файла в памяти потребовалось немного меньше времени, чем при последовательном вводе из файла с буферизацией, и значительно меньше времени, чем при произвольном доступе к файлу средствами класса `RandomAccessFile`. Разумеется, на других машинах эти показатели будут выглядеть несколько иначе, но не вызывает никаких сомнений, что выигрыш в производительности может оказаться значительным при отображении файла в памяти по сравнению с произвольным доступом к нему. В то же время для последовательного ввода из файлов среднего размера прибегать к отображению в памяти нецелесообразно.

Пакет `java.nio` делает отображение файлов в памяти довольно простым процессом. С этой целью для файла сначала получается *канал*, как показано ниже. Под каналом подразумевается предназначенная для дисковых файлов абстракция, которая позволяет получать доступ к таким функциональным возможностям операционной системы, как отображение в памяти, блокировка файлов и быстрая передача данных между файлами.

```
FileChannel channel = FileChannel.open(path, options);
```

Затем из канала получается объект типа `ByteBuffer` в результате вызова метода `map()` из класса `FileChannel`. При этом указывается отображаемая в памяти часть файла и *режим отображения*. В целом поддерживаются три следующих режима отображения.

- `FileChannel.MapMode.READ_ONLY`. Получаемый в итоге буфер служит только для чтения. Любые попытки записи данных в этот буфер приведут к исключению типа `ReadOnlyBufferException`.
- `FileChannel.MapMode.READ_WRITE`. Получаемый в итоге буфер служит как для чтения, так и для записи, благодаря чему все вносимые изменения будут в определенный момент времени записываться обратно в файл. Следует, однако, иметь в виду, что другие программы, отобразившие в памяти тот же самый файл, возможно, и не сразу обнаружат эти изменения. Конкретное поведение при одновременном отображении файла в памяти многими программами зависит от используемой операционной системы.
- `FileChannel.MapMode.PRIVATE`. Получаемый в итоге буфер служит как для чтения, так и для записи, но любые вносимые изменения относятся только к этому буферу, а следовательно, они не будут распространяться на файл.

Получив требуемый буфер, можно переходить непосредственно к чтению и записи данных с помощью методов из класса `ByteBuffer` и его суперкласса `Buffer`. Буферы поддерживают как последовательный, так и произвольный доступ к данным. В любом буфере имеется *позиция*, продвигаемая методами `get()` и `put()`. В качестве

примера ниже приведен код, требующийся для последовательного обхода всех байтов в буфере.

```
while (buffer.hasRemaining())
{
  byte b = buffer.get();
  . . .
}
```

А код, требующийся для произвольного доступа, выглядит следующим образом:

```
for (int i = 0; i < buffer.limit(); i++)
{
  byte b = buffer.get(i);
  . . .
}
```

Кроме того, читать и записывать массивы байтов можно с помощью следующих методов:

```
get(byte[] bytes)
get(byte[], int offset, int length)
```

И, наконец, с помощью перечисленных ниже методов можно читать значения примитивных типов, хранящиеся в файле в *двоичном* формате.

```
getInt()
getLong()
getShort()
getChar()
getFloat()
getDouble()
```

Как упоминалось ранее, в Java для хранения данных в двоичном формате применяется обратный порядок следования байтов от старшего к младшему. Но если требуется обработать файл, содержащий данные в двоичном формате с прямым порядком следования байтов от младшего к старшему, достаточно сделать следующий вызов:

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

А для выяснения текущего порядка следования байтов достаточно сделать приведенный ниже вызов.

```
ByteOrder b = buffer.order()
```

ВНИМАНИЕ! В двух последних методах условные обозначения имен `get/set` не соблюдаются.

Для записи числовых данных в буфер можно воспользоваться одним из перечисленных ниже методов. В какой-то момент и, безусловно, когда канал закрывается, внесенные изменения записываются обратно в файл.

```
PutInt()
putLong()
putShort()
putChar()
putFloat()
putDouble()
```

В листинге 1.5 приведен пример программы, вычисляющей для файла контрольную сумму в 32-разрядном циклическом коде с избыточностью (CRC32). Такая контрольная сума часто применяется для того, чтобы выяснить, не был ли файл поврежден. Повреждение файла практически неизбежно ведет к изменению этой контрольной суммы. В состав пакета `java.util.zip` входит класс `CRC32`, позволяющий вычислять такую контрольную сумму для последовательности байтов в следующем цикле:

```
CRC32 crc = new CRC32();
while (есть ли дополнительные байты?)
  crc.update(следующий байт)
long checksum = crc.getValue();
```

НА ЗАМЕТКУ! Подробное описание алгоритма CRC имеется по адресу `http://www.relisoft.com/Science/CrcMath.html`.

Подробности вычисления контрольной суммы CRC32 не так важны. Но здесь оно демонстрируется лишь в качестве наглядного примера полезной операции над файлами. Запустить рассматриваемую здесь программу можно, введя следующую команду:

```
java memoryMap.MemoryMapTest имя_файла
```

Листинг 1.5. Исходный код из файла `memoryMap/MemoryMapTest.java`

```
1   package memoryMap;
2
3   import java.io.*;
4   import java.nio.*;
5   import java.nio.channels.*;
6   import java.nio.file.*;
7   import java.util.zip.*;
8   /**
9    * В этой программе вычисляется четырьмя способами контрольная
10   * сумма CRC32 для файла
11   * Использование: java memoryMap.MemoryMapTest имя_файла
12   * @version 1.01 2012-05-30
13   * @author Cay Horstmann
14  */
15  public class MemoryMapTest
16  {
17    public static long checksumInputStream(Path filename) throws IOException
18    {
19      try (InputStream in = Files.newInputStream(filename))
20      {
21        CRC32 crc = new CRC32();
22
23        int c;
24        while ((c = in.read()) != -1)
25          crc.update(c);
26        return crc.getValue();
27      }
28    }
29
```

```
30    public static long checksumBufferedInputStream(Path filename)
31          throws IOException
32    {
33      try (InputStream in =
34           new BufferedInputStream(Files.newInputStream(filename)))
35      {
36         CRC32 crc = new CRC32();
37
38         int c;
39         while ((c = in.read()) != -1)
40           crc.update(c);
41         return crc.getValue();
42      }
43    }
44
45    public static long checksumRandomAccessFile(Path filename)
46          throws IOException
47    {
48      try (RandomAccessFile file =
49           new RandomAccessFile(filename.toFile(), "r"))
50      {
51        long length = file.length();
52        CRC32 crc = new CRC32();
53
54        for (long p = 0; p < length; p++)
55        {
56          file.seek(p);
57          int c = file.readByte();
58          crc.update(c);
59        }
60        return crc.getValue();
61      }
62    }
63
64    public static long checksumMappedFile(Path filename) throws IOException
65    {
66      try (FileChannel channel = FileChannel.open(filename))
67      {
68        CRC32 crc = new CRC32();
69        int length = (int) channel.size();
70        MappedByteBuffer buffer =
71          channel.map(FileChannel.MapMode.READ_ONLY, 0, length);
72
73          for (int p = 0; p < length; p++)
74          {
75            int c = buffer.get(p);
76            crc.update(c);
77          }
78          return crc.getValue();
79      }
80    }
81
82    public static void main(String[] args) throws IOException
83    {
84      System.out.println("Input Stream:");
85      long start = System.currentTimeMillis();
86      Path filename = Paths.get(args[0]);
```

```
87      long crcValue = checksumInputStream(filename);
88      long end = System.currentTimeMillis();
89      System.out.println(Long.toHexString(crcValue));
90      System.out.println((end - start) + " milliseconds");
91
92      System.out.println("Buffered Input Stream:");
93      start = System.currentTimeMillis();
94      crcValue = checksumBufferedInputStream(filename);
95      end = System.currentTimeMillis();
96      System.out.println(Long.toHexString(crcValue));
97      System.out.println((end - start) + " milliseconds");
98
99      System.out.println("Random Access File:");
100     start = System.currentTimeMillis();
101     crcValue = checksumRandomAccessFile(filename);
102     end = System.currentTimeMillis();
103     System.out.println(Long.toHexString(crcValue));
104     System.out.println((end - start) + " milliseconds");
105
106     System.out.println("Mapped File:");
107     start = System.currentTimeMillis();
108     crcValue = checksumMappedFile(filename);
109     end = System.currentTimeMillis();
110     System.out.println(Long.toHexString(crcValue));
111     System.out.println((end - start) + " milliseconds");
112   }
113 }
```

java.io.FileInputStream 1.0

- `FileChannel getChannel()` **1.4**

  Возвращает канал для получения доступа к данному потоку ввода.

java.io.FileOutputStream 1.0

- `FileChannel getChannel()` **1.4**

  Возвращает канал для получения доступа к данному потоку вывода.

java.io.RandomAccessFile 1.0

- `FileChannel getChannel()` **1.4**

  Возвращает канал для получения доступа к данному файлу.

java.nio.channels.FileChannel 1.4

- `static FileChannel open(Path` *path*`, OpenOption...` *options*`)` **7**

  Открывает канал доступа к файлу по заданному пути. По умолчанию канал открывается для чтения.

Параметры: *path* Путь к файлу, для доступа к которому открывается канал

options Принимает значения следующих констант из перечисления типа StandardOpenOption: WRITE, APPEND, TRUNCATE_EXISTING, CREATE

- `MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`
  Отображает часть файла в памяти.

  Параметры: *mode* Принимает значения следующих констант из класса FileChannel.MapMode: READ_ONLY, READ_WRITE или PRIVATE

  position Начало отображаемой части файла

  size Размер отображаемой части файла

java.nio.Buffer 1.4

- `boolean hasRemaining()`
  Возвращает логическое значение `true`, если текущее положение в буфере еще не достигло предельной позиции.
- `int limit()`
  Возвращает предельную позицию в буфере, т.е. первую позицию, где больше нет никаких значений.

java.nio.ByteBuffer 1.4

- `byte get()`
  Получает байт из текущей позиции и продвигает текущую позицию к следующему байту.
- `byte get(int index)`
  Получает байт по указанному индексу.
- `ByteBuffer put(byte b)`
  Размещает байт на текущей позиции и продвигает ее к следующему байту. Возвращает ссылку на данный буфер.
- `ByteBuffer put(int index, byte b)`
  Размещает байт по указанному индексу. Возвращает ссылку на данный буфер.

- `ByteBuffer get(byte[] destination)`
- `ByteBuffer get(byte[] destination, int offset, int length)`

  Заполняют байтовый массив или только какую-то его часть байтами из буфера и продвигают текущую позицию на количество считанных байтов. Если в буфере недостаточно байтов, то ничего не считывают и генерируют исключение типа `BufferUnderflowException`. Возвращают ссылку на данный буфер.

  Параметры:

destination	Заполняемый байтовый массив
offset	Смещение заполняемой части массива
length	Длина заполняемой части массива

- `ByteBuffer put(byte[] source)`
- `ByteBuffer put(byte[] source, int offset, int length)`

  Размещают в буфере все байты из байтового массива или только какую-то их часть и продвигают текущую позицию на количество записанных байтов. Если в буфере слишком много байтов, то ничего не записывают и генерируют исключение типа `BufferOverflowException`. Возвращают ссылку на данный буфер.

  Параметры:

source	Записываемый байтовый массив
offset	Смещение записываемой части массива
length	Длина записываемой части массива

`Xxx getXxx()`

- `Xxx getXxx(int index)`
- `ByteBuffer putXxx(Xxx value)`
- `ByteBuffer putXxx(int index, Xxx value)`

  Получают или размещают в буфере двоичное число. Вместо *Xxx* может быть указан один из следующих примитивных типов данных: `Int`, `Long`, `Short`, `Char`, `Float` или `Double`.

- `ByteBuffer order(ByteOrder order)`
- `ByteOrder order()`

  Устанавливают или получают порядок следования байтов. Параметр *order* может принимать значение одной из следующих констант из класса `ByteOrder`: `BIG_ENDIAN` или `LITTLE_ENDIAN`.

- `static ByteBuffer allocate(int capacity)`

  Конструирует буфер заданной емкости.

- `static ByteBuffer wrap(byte[] values)`

  Конструирует буфер, опирающийся на заданный массив.

- `CharBuffer asCharBuffer()`

  Конструирует символьный буфер, опирающийся на данный буфер. Изменения в символьном буфере отражаются в данном буфере, но у символьного буфера имеется своя позиция, предел и отметка.

java.nio.CharBuffer 1.4

- `char get()`
- `CharBuffer get(char[] destination)`
- `CharBuffer get(char[] destination, int offset, int length)`

  Получают значение типа `char` или ряд подобных значений, начиная с указанной позиции в буфере и продвигая ее на количество считанных символов. Два последних метода возвращают ссылку `this` на данный буфер.

- `CharBuffer put(char c)`
- `CharBuffer put(char[] source)`
- `CharBuffer put(char[] source, int offset, int length)`
- `CharBuffer put(String source)`
- `CharBuffer put(CharBuffer source)`

  Размещают в буфере одно значение типа `char` или ряд подобных значений, начиная с указанной позиции в буфере и продвигая ее на количество записанных символов. При чтении из исходного буфера типа `CharBuffer` считываются все оставшиеся в нем символы. Возвращают ссылку `this` на данный буфер.

Структура буфера данных

При использовании механизма отображения файлов в памяти создается единственный буфер, охватывающий файл полностью или только интересующую его часть. Но буферы могут применяться для чтения или записи и более скромных фрагментов данных.

В этом разделе дается краткое описание основных операций, которые могут выполняться над объектами типа `Buffer`. *Буфером* называется массив значений одинакового типа. Класс `Buffer` является абстрактным с такими производными от него конкретными подклассами, как `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer` и `ShortBuffer`.

НА ЗАМЕТКУ! Класс `StringBuffer` не имеет никакого отношения к этим подклассам, реализующим буферы данных.

На практике чаще всего применяются классы `ByteBuffer` и `CharBuffer`. Как показано на рис. 1.10, каждый буфер обладает следующим свойствами:

- *Емкость*, которая вообще не изменяется.
- *Позиция*, начиная с которой считывается или записывается следующее значение.
- *Предел*, вне которого чтение или запись не имеет смысла.
- Необязательная *отметка* для повторения операции чтения или записи.

Все эти свойства удовлетворяют следующему условию:

0 ≤ *отметка* ≤ *позиция* ≤ *предел* ≤ *емкость*

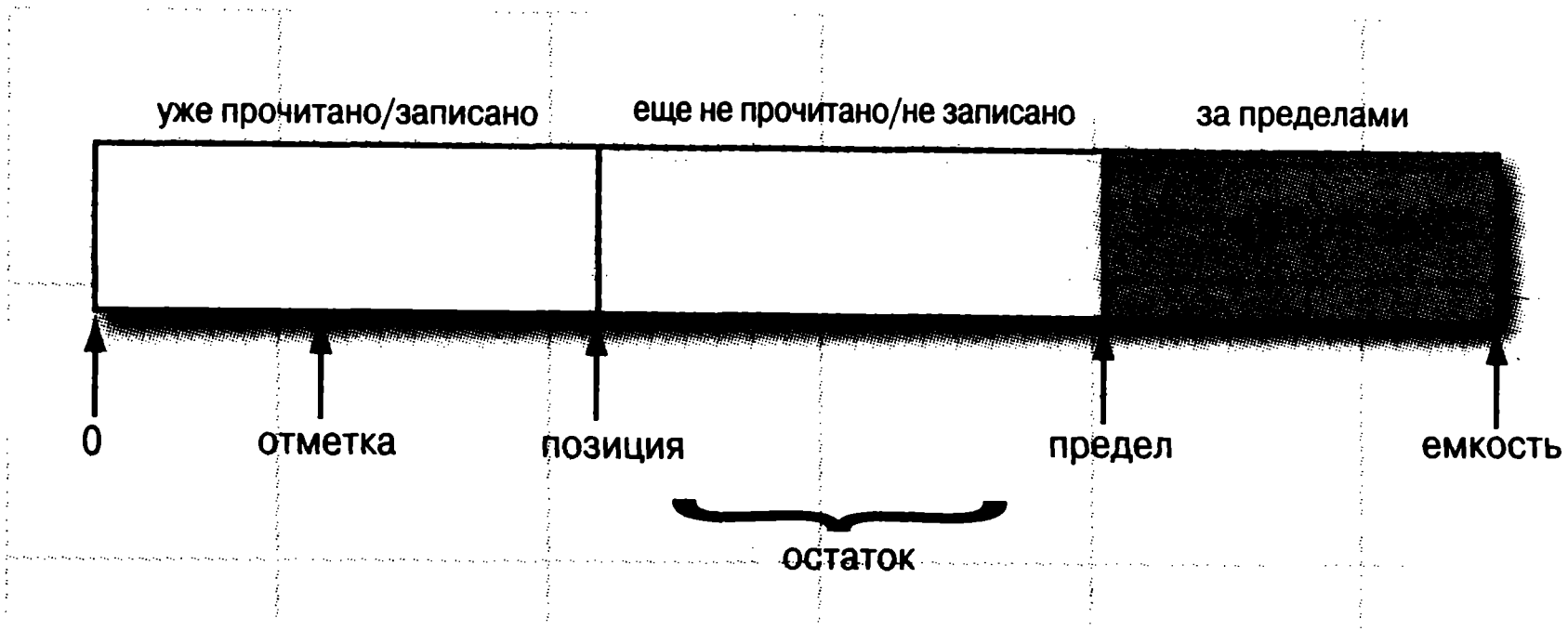


Рис. 1.10. Буфер

Буфер работает, главным образом, по циклу "сначала запись, затем чтение". Исходная позиция в буфере соответствует нулю (**0**), а предел — его емкости. Для ввода значений в буфер следует вызвать метод `put()`. Исчерпав вводимые в буфер данные или заполнив всю его емкость, можно переходить к чтению данных из буфера.

Для того чтобы установить предел на текущей позиции, а позицию — на нуле, следует вызвать метод `flip()`. Далее можно вызывать метод `get()` до тех пор, пока метод `remaining()` будет возвращать положительные значения разности *предела* и *позиции*. Считав все значения из буфера, нужно вызвать метод `clear()`, чтобы подготовить буфер к следующему циклу записи. Метод `clear()` устанавливает позицию в исходное нулевое положение, а предел — равным емкости буфера.

Ели требуется выполнить повторное чтение данных из буфера, то для этого следует воспользоваться такими методами, как `rewind()` или `mark()` и `reset()`. Более подробно эти и другие методы обращения с буфером данных поясняются в приведенном далее описании прикладного интерфейса API.

Для получения самого буфера нужно вызвать статический метод `ByteBuffer.allocate()` или `ByteBuffer.wrap()`. После этого можно заполнить буфер из открытого канала или вывести его содержимое в канал. В приведенном ниже примере кода показано, как это делается. Такой способ может оказаться полезной альтернативой произвольному доступу к файлу.

```
ByteBuffer buffer = ByteBuffer.allocate(RECORD_SIZE);
channel.read(buffer);
channel.position(newpos);
buffer.flip();
channel.write(buffer);
```

java.nio.Buffer 1.4

- `Buffer clear()`

  Подготавливает данный буфер к записи данных, устанавливая позицию в нулевое положение, а предела — равным емкости буфера. Возвращает ссылку `this` на данный буфер.

- `Buffer flip()`

  Подготавливает данный буфер к чтению после записи, устанавливая предел на текущей позиции, а саму позицию — в нулевое положение. Возвращает ссылку `this` на данный буфер.

- `Buffer rewind()`

  Подготавливает данный буфер к повторному чтению тех же самых значений, устанавливая позицию в нулевое положение и оставляя предел неизменным. Возвращает ссылку `this` на данный буфер.

- `Buffer mark()`

  Устанавливает отметку данного буфера на текущей позиции. Возвращает ссылку `this` на данный буфер.

- `Buffer reset()`

  Устанавливает текущую позицию данного буфера на отметке, позволяя тем самым снова читать и записывать данные в буфер с отмеченной позиции. Возвращает ссылку `this` на данный буфер.

- `int remaining()`

  Возвращает оставшееся количество значений, доступных для чтения или записи в буфере, т.е. разность предела и позиции.

- `int position()`

- `void position(int newValue)`

  Получают или устанавливают текущую позицию в данном буфере

- `int capacity()`

  Возвращает емкость данного буфера.

Блокирование файлов

Когда нескольким одновременно выполняющимся программам требуется видоизменить один и тот же файл, они должны каким-то образом взаимодействовать друг с другом, иначе они могут легко испортить файл. В качестве выхода из этого затруднительного положения могут послужить блокировки файлов. В частности, блокировка файла управляет доступом ко всему файлу в целом или же к определенному ряду байтов в нем.

Допустим, пользовательские глобальные параметры настройки прикладной программы сохраняются в конфигурационном файле. Если пользователь вызывает два экземпляра этой программы, то вполне возможно, что в обоих ее экземплярах потребуется выполнить запись в конфигурационный файл в один и тот же момент времени. В таком случае файл должен быть заблокирован в первом экземпляре прикладной программы. Когда же во втором ее экземпляре обнаружится, что файл заблокирован, то в нем может быть принято решение подождать до тех пор, пока файл не разблокируется, или вообще отказаться от записи.

Для блокирования файла можно вызывать метод `lock()` или `tryLock()` из класса `FileChannel` следующим образом:

```
FileChannel = FileChannel.open(path);
FileLock lock = channel.lock();
```

или

```
FileLock lock = channel.tryLock();
```

Когда вызывается метод `lock()`, он блокируется до тех пор, пока блокировка не будет доступна. А когда вызывается метод `tryLock()`, то сразу же возвращается разрешение на блокировку или пустое значение `null`, если блокировка недоступна для установки. Файл остается заблокированным до тех пор, пока не закроется канал или не будет вызван снимающий блокировку метод `release()`.

Для блокирования только какой-нибудь определенной части файла можно сделать один из следующих вызовов:

```
FileLock lock(long start, long size, boolean exclusive)
```

или

```
FileLock tryLock(long start, long size, boolean exclusive)
```

Если указать логическое значение `false` признака `shared`, файл будет заблокирован как для записи, так и для чтения. Если же указать логическое значение `true` этого признака, то станет доступной *разделяемая блокировка*, позволяющая нескольким процессам читать из файла, но не допускающая ни для одного из них приобретение права на исключительную блокировку. Разделяемые блокировки поддерживаются не во всех операционных системах. Поэтому вполне возможно получить право на исключительную блокировку, запрашивая только разделяемую блокировку. Для того чтобы выяснить, какой из этих видов блокировки доступен, нужно вызывать метод `isShared()` из класса `FileLock`.

НА ЗАМЕТКУ! Если сначала блокируется концевая часть файла, а затем содержимое файла разрастается за пределы заблокированной части, то дополнительная часть файла не блокируется. Для блокирования всех байтов в файле нужно указать значение `Long.MAX_VALUE` его размера.

По завершении операций над файлом непременно снимите с него блокировку. Как всегда, это лучше всего сделать с помощью оператора `try` с ресурсами, как показано ниже.

```
try (FileLock lock = channel.lock())
{
  получить доступ к заблокированному файлу или его части
}
```

Не следует, однако, забывать, что блокирование файлов зависит от используемой системы. Ниже перечислены некоторые особенности блокирования файлов, на которые следует обращать внимание.

- В некоторых системах блокирование файлов является лишь *желательным*, но не обязательным. Если прикладной программе не удастся получить разрешение на блокировку, она все равно может начать запись данных файл, несмотря на то, что доступ к этому файлу в данный момент заблокирован другой прикладной программой.
- В некоторых системах нельзя одновременно блокировать файл и отображать его в памяти.
- Блокировки файлов удерживаются всей виртуальной машиной Java. Если с помощью одной и той же виртуальной машины запускаются сразу две программы (например, аплет или программа запуска приложений), эти программы не смогут по отдельности получать разрешение на блокировку одного и того же файла. А если виртуальная машина уже удерживает другую перекрывающую

блокировку для того же самого файла, то методы `lock()` и `tryLock()` будут просто генерировать исключение типа `OverlappingFileLockException`.

- В некоторых системах закрытие канала приводит к снятию с базового файла всех блокировок, которые удерживаются виртуальной машиной Java. Поэтому лучше не открывать много каналов для одного и того же заблокированного файла.
- Блокирование файлов в сетевой файловой системе очень сильно зависит от используемой системы, и поэтому в таких системах лучше всего избегать этого механизма.

java.nio.channels.FileChannel 1.4

- `FileLock lock()`

  Получает разрешение на исключительную блокировку всего файла. Блокируется до тех пор, пока не будет получено разрешение на блокировку.

- `FileLock tryLock()`

  Получает разрешение на исключительную блокировку всего файла или возвращает пустое значение `null`, если блокировка не может быть получена.

- `FileLock lock(long` *`position`*`, long` *`size`*`, boolean` *`shared`*`)`
- `FileLock tryLock(long` *`position`*`, long` *`size`*`, boolean` *`shared`*`)`

  Получают разрешение на блокировку доступа к определенной части файла. Первый метод блокируется до тех пор, пока такое разрешение не будет получено, а второй — возвращает пустое значение `null`, если получить разрешение на блокировку не удается.

  Параметры:

`position`	Начало блокируемой части файла
`size`	Размер блокируемой части файла
`shared`	Принимает логическое значение `true`, если требуется разделяемая блокировка, или логическое значение `false`, если требуется исключительная блокировка

java.nio.channels.FileLock 1.4

- `void close()` **1.7**

  Снимает блокировку.

Регулярные выражения

Регулярные выражения применяются для указания шаблонов строк. С их помощью можно отыскать символьные строки, совпадающие с конкретным шаблоном. Например, в одном из рассматриваемых далее примеров программ осуществляется поиск по шаблону `<a href="...">` для обнаружения всех гиперссылок в HTML-файле.

Безусловно, для определения шаблона обозначение `...` является не достаточно точным. Необходимо как можно конкретнее указывать, какая именно последовательность символов допускается для совпадения. Поэтому для описания каждого шаблона требуется специальный синтаксис регулярных выражений. В качестве примера рассмотрим простое регулярное выражение `[Jj]ava.+`, обеспечивающее совпадение с любой символьной строкой, отвечающей следующим критериям поиска:

- начинается с буквы **J** или **j**;
- содержит **ava** на месте трех последующих букв;
- а в остальной части содержит один или более произвольный символ.

Например, строка `"javanese"` совпадает с данным регулярным выражением, а строка `"Core Java"` — не совпадает.

Как видите, для того чтобы понять смысл регулярного выражения, нужно хотя бы немного разбираться в его синтаксисе. Правда, для большинства целей вполне хватает небольшого набора довольно простых синтаксических конструкций, рассматриваемых ниже.

- *Класс символов* — это набор альтернативных символов, заключенных в квадратные скобки, например: `[Jj]`, `[0-9]`, `[A-Za-z]` или `[^0-9]`. Здесь знаком **-** обозначается диапазон символов (т.е. все символы, значения которых в уникоде находятся в указанных пределах), а знаком **^** — дополнение (т.е. все символы, кроме указанных).
- Для того чтобы включить знак **-** в класс символов, его нужно сделать первым или последним элементом данного класса. А для того чтобы включить знак **[**, его следует сделать первым элементом. И для того чтобы включить знак **^**, его достаточно разместить где угодно, только не в начале класса символов. Экранировать необходимо только знаки **[** и ****.
- Имеется немало предопределенных классов символов вроде **\d** (для цифр) или `\p{Sc}` (для знака денежной единицы в уникоде), как показано в табл. 1.8 и 1.9.
- Большинство символов указываются для совпадения непосредственно в шаблоне, как, например, буквы `ava` в рассмотренном выше шаблоне.
- Знак **.** обозначает совпадение с любым символом, кроме символов окончания строки (в зависимости от установленных признаков).
- Знак **** служит для экранирования символов, например, выражение `\.` обозначает совпадение с точкой, а выражение `\\` — совпадение с обратной косой чертой.
- Знаки **^** и **$** обозначают совпадение в начале и в конце строки соответственно.
- Если *X* и *Y* являются регулярными выражениями, то выражение *XY* обозначает "любое совпадение с *X*, после которого следует совпадение с *Y*", а выражение *X* `|` *Y* — "любое совпадение с X или Y".

- В выражении X можно применять *квантификаторы* вроде *X*+ (**1** или больше), *X** (**0** или больше) и *X*? (**0** или **1**).
- По умолчанию квантификатор обозначает совпадение с наибольшим количеством возможных повторений, определяющих удачный исход всего сопоставления с шаблоном в целом. Этот режим можно изменять с помощью суффикса **?** (обозначающего минимальное, или нестрогое, совпадение при наименьшем количестве повторений) и суффикса **+** (обозначающего максимальное, строгое или полное совпадение при наибольшем количестве повторений, даже если это чревато неудачным исходом всего сопоставления с шаблоном в целом).
- Например, символьная строка `"cab"` совпадает с шаблоном `[a-z]*ab`, но не с шаблоном `[a-z]*+ab`. В первом случае с выражением `[a-z]*` совпадает только символ **c**, поэтому символы `ab` совпадают с остальной частью шаблона. А во втором, более строгом случае символы `cab` совпадают с выражением `[a-z]*+`, тогда как остальная часть шаблона остается не совпавшей.
- Для определения подвыражений можно использовать *группы*. Все группы следует непременно заключать в круглые скобки, например `([+-]?)([0-9]+)`. После этого можно обратиться к сопоставителю с шаблоном, чтобы возвратить совпадение с каждой группой или обратную ссылку на отдельную группу с помощью выражения с `\n`, где `n` — номер группы, начиная с `\1`.

Таблица 1.8. Синтаксис регулярных выражений

Синтаксис	Описание
Символы	
c	Символ ***c***
\unnnn, \xnn, \0n, \0nn, \0nnn	Кодовая единица с заданным шестнадцатеричным или восьмеричным значением
\t, \n, \r, \f, \a, \e	Управляющие символы табуляции, новой строки, возврата каретки, перевода страницы, предупреждения и перехода
\cc	Управляющий символ, соответствующий символу ***c***
Классы символов	
[C_1C_2...]	Любой символ из последовательности C_1, C_2..., где C_i обозначает отдельные символы, диапазоны символов (C_1–C_2) или классы символов
[^...]	Дополнение к классу символов
[... && ...]	Пересечение двух классов символов
Предопределенные классы символов	
.	Любой символ, кроме символа конца строки (или вообще любой символ, если установлен признак DOTALL)
\d	Цифровой символ **[0-9]**
\D	Нецифровой символ **[^0-9]**
\s	Пробельный символ **[\t\n\r\f\x0B]**
\S	Непробельный символ
\w	Словообразующий символ **[a-zA-Z0-9_]**

Окончание табл. 1.8

Синтаксис	Описание
Предопределенные классы символов	
`\W`	Несловообразующий символ
`\p{`*имя*`}`	Именованный класс символов (см. табл. 1.9)
`\P{`*имя*`}`	Дополнение к именованному классу символов
Обнаружители границ	
`^ $`	Начало и конец ввода (или начало и конец строки в многострочном режиме)
`\b`	Словесная граница
`\B`	Несловесная граница
`\A`	Начало ввода
`\z`	Конец ввода
`\Z`	Конец ввода без учета последнего символа завершения строки
`\G`	Конец предыдущего совпадения
Квантификаторы	
X`?`	Необязательное присутствие выражения *X*
X`*`	Повторение 0 или больше раз выражения *X*
X`+`	Повторение 1 или больше раз выражения *X*
X`{n}` *X*`{n,}` *X*`{n,m}`	Повторение `n` раз, как минимум `n` раз, от `n` до `m` раз выражения *X*
Суффиксы квантификаторов	
`?`	Превращает исходное максимальное совпадение в минимальное
`+`	Превращает исходное максимальное совпадение в полное
Операции над множествами	
XY	Любая строка из выражения *X*, после которой следует любая строка из выражения *Y*
X`|`*Y*	Любая строка из выражения *X* или *Y*
Группирование	
`(`*X*`)`	Фиксация строки, совпадающей с выражением *X*, в виде группы
`\`*n*	Совпадение с *n*-й группой
Управляющие символы	
`\c`	Символ `c` (не должен быть алфавитным символом)
`\Q ... \E`	Цитирование ... дословная передача
`(? ... )`	Специальная конструкция; см. ниже пояснения к классу `Pattern` в описании прикладного интерфейса API

Таблица 1.9. Имена предопределенных классов символов

Имя класса символов	Описание
Lower	Символы нижнего регистра в коде ASCII **[a-z]**
Upper	Символы верхнего регистра в коде ASCII **[A-Z]**
Alpha	Буквенные символы в коде ASCII **[A-Za-z]**
Digit	Цифровые символы в коде ASCII **[0-9]**
Alnum	Буквенные или цифровые символы в коде ASCII **[A-Za-z0-9]**
XDigit	Шестнадцатеричные цифры **[0-9A-Fa-f]**
Print или **Graph**	Печатаемые символы в коде ASCII **[\x21-\x7E]**
Punct	Небуквенные или цифровые символы в коде ASCII **[\p{Print}&&\P{Alnum}]**
ASCII	Все символы в коде ASCII **[\x00-\x7F]**
Cntrl	Управляющие символы в коде ASCII **[\x00-\x1F]**
Blank	Символы пробела или табуляции **[\t]**
Space	Пробельные символы **[\t\n\r\f\0x0B]**
javaLowerCase	Символы нижнего регистра, определяемые методом **Character.isLowerCase()**
javaUpperCase	Символы верхнего регистра, определяемые методом **Character.isUpperCase()**
javaWhitespace	Пробельные символы, определяемые методом **Character.isWhitespace()**
javaMirrored	Зеркально отображаемые символы, определяемые методом **Character.isMirrored()**
In*Block*	Здесь ***Block*** — имя блока символов в уникоде без пробелов, например **Arrows** или **Latin1Supplement**
Is*Script*	Здесь ***Script*** — имя сценария в уникоде без пробелов, например **Common**
Category или **IsCategory**	Здесь **Category** — имя категории символов в уникоде, например **L** (буква) или **Sc** (знак денежной единицы)
Is*Property*	Здесь ***Property*** — одно из имен свойств **Alphabetic**, **Ideographic**, **Letter**, **Lowercase**, **Uppercase**, **Titlecase**, **Punctuation**, **Control**, **White_Space**, **Digit**, **Hex_Digit**, **Noncharacter_Code_Point**, **Assigned**

Например, ниже приведено несколько сложное, но потенциально полезное регулярное выражение (в нем описываются десятичные или шестнадцатеричные целые числа).

```
[+-]?[0-9]+|0[Xx][0-9A-Fa-f]+
```

К сожалению, синтаксис регулярных выражений не полностью стандартизирован и может выглядеть по-разному в различных программах и библиотеках, где они применяются. И хотя существует общее согласие по базовым конструкциям, тем не менее, имеется масса досадных отличий в деталях. В классах, реализующих регулярные выражения в Java, применяется синтаксис, подобный, но все-таки не полностью совпадающий с тем, что применяется в языке Perl. В табл. 1.8 перечислены все

конструкции этого синтаксиса в Java. Более подробные сведения о синтаксисе регулярных выражения можно найти в документации на прикладной интерфейс API для класса `Pattern` или в книге *Mastering Regular Expression, 3d Edition* Джефри Фридла (Jeffrey E. F. Friedl; издательство O'Reilly and Associates, 2006 г.; в русском переводе книга вышла под названием *Регулярные выражения*, 3-е издание в издательстве "Символ-Плюс", 2008 г.).

Самым простым примером применения регулярного выражения является проверка конкретной символьной строки на совпадение с ним. Ниже приведен пример программы, выполняющей такую проверку в Java. Сначала в этой программе из символьной строки, содержащей регулярное выражение, создается объект типа `Pattern`. Затем из этого объекта получается объект типа `Matcher` и вызывается его метод `matches()` следующим образом:

```
Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) . . .
```

В качестве входных данных для сопоставителя с шаблоном может служить объект любого класса, который реализует интерфейс `CharSequence`, например `String`, `StringBuilder` или `CharBuffer`. При компиляции шаблона можно устанавливать один или более признак, как показано в приведенном ниже примере кода.

```
Pattern pattern = Pattern.compile(patternString,
  Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE);
```

Всего поддерживается шесть признаков, которые перечислены ниже.

- `CASE_INSENSITIVE`. Обозначает сопоставление символов с шаблоном без учета регистра. По умолчанию этот признак принимает во внимание только символы в коде US ASCII.
- `UNICODE_CASE`. В сочетании с признаком `CASE_INSENSITIVE` обозначает сопоставление символов с шаблоном, учитывая регистр букв в уникоде.
- `MULTILINE`. Знаки **^** и **$** обозначают сопоставление символов с шаблоном в начале и в конце строки, а не во всех входных данных.
- `UNIX_LINES`. Обозначает распознавание только **'\n'** в качестве символа конца строки при сопоставлении символов с шаблонами **^** и **$** в многострочном режиме.
- `DOTALL`. Обозначает совпадение со знаком **.** всех символов, включая и символы конца строки.
- `CANON_EQ`. Обозначает необходимость учитывать каноническую эквивалентность символов уникода. Например, символ **u**, после которого следует знак ё (трема — диакритический знак над гласной), будет соответствовать символу **ü**.

Если регулярное выражение содержит группы, то объект типа `Matcher` может обнаруживать их границы. Приведенные ниже методы выдают начальный и конечный индексы конкретной группы.

```
int start(int groupIndex)
int end(int groupIndex)
```

Для того чтобы извлечь совпавшую символьную строку, достаточно сделать следующий вызов:

```
String group(int groupIndex)
```

Под нулевой группой подразумеваются все входные данные, а индекс первой фактической группы равен 1. Для получения сведений об общем количестве групп следует вызвать метод `groupCount()`. Вложенные группы упорядочиваются с помощью круглых скобок. Так, если используется шаблон `((1?[0-9]):([0-5][0-9]))[ap]m` и входные данные `11:59am`, то сопоставитель с шаблоном сообщит о перечисленных ниже группах.

Индекс группы	Начальная позиция	Конечная позиция	Строка
0	0	7	11:59am
1	0	5	11:59
2	0	2	11
3	3	5	59

В листинге 1.6 приведен исходный код примера программы, в которой сначала предлагается указать шаблон, а затем — сопоставляемые с ним символьные строки, после чего сообщается, совпадают ли введенные строки с шаблоном. Если же в совпавших введенных строках и шаблоне присутствуют группы, то выводятся границы групп в круглых скобках, как показано в приведенном ниже примере.

```
((11):(59))am
```

Листинг 1.6. Исходный код из файла `regex/RegexTest.java`

```
1  package regex;
2
3  import java.util.*;
4  import java.util.regex.*;
5
6  /**
7   * В этой программе производится проверка на совпадение с регулярным
8   * выражением. Для этого следует ввести шаблон и сопоставляемые с ним
9   * символьные строки, а для выхода из программы — нажать клавишу пробела.
10  * Если шаблон содержит группы, их границы отображаются при совпадении.
11  * @version 1.02 2012-06-02
12  * @author Cay Horstmann
13  */
14    public class RegexTest
15    {
16      public static void main(String[] args) throws PatternSyntaxException
17      {
18        Scanner in = new Scanner(System.in);
19        System.out.println("Enter pattern: ");
20        String patternString = in.nextLine();
21
22        Pattern pattern = Pattern.compile(patternString);
23
24        while (true)
25        {
26          System.out.println("Enter string to match: ");
27          String input = in.nextLine();
28          if (input == null || input.equals("")) return;
29          Matcher matcher = pattern.matcher(input);
```

```
30          if (matcher.matches())
31          {
32            System.out.println("Match");
33            int g = matcher.groupCount();
34            if (g > 0)
35            {
36              for (int i = 0; i < input.length(); i++)
37              {
38                // вывести любые пустые группы
39                for (int j = 1; j <= g; j++)
40                  if (i == matcher.start(j) && i == matcher.end(j))
41                    System.out.print("()");
42                // вывести знак ( в начале непустых групп
43                for (int j = 1; j <= g; j++)
44                  if (i == matcher.start(j) && i != matcher.end(j))
45                    System.out.print('(');
46                System.out.print(input.charAt(i));
47                // вывести знак ) в конце непустых групп
48                for (int j = 1; j <= g; j++)
49                  if (i + 1 != matcher.start(j) && i + 1 == matcher.end(j))
50                    System.out.print(')');
51              }
52              System.out.println();
53            }
54          }
55          else
56            System.out.println("No match");
57        }
58      }
59    }
```

Обычно с регулярным выражением требуется сопоставлять не все входные данные, а только отыскивать в них одну или более совпадающую символьную строку. Для поиска следующего совпадения служит метод `find()` из класса `Matcher`. Если он возвращает логическое значение `true`, то для выяснения протяженности совпадения можно далее вызвать методы `start()` и `end()`.

```
while (matcher.find())
{
  int start = matcher.start();
  int end = matcher.end();
  String match = input.substring(start, end);
  . . .
}
```

В листинге 1.7 приведен пример программы, где этот механизм приводится в действие. В ней отыскиваются и выводятся на экран все гипертекстовые ссылки, присутствующие на веб-странице. Чтобы запустить эту программу на выполнение, в командной строке нужно указать какой-нибудь веб-адрес, например, следующим образом:

```
java HrefMatch http://www.horstmann.com
```

Листинг 1.7. Исходный код из файла `match/HrefMatch.java`

```
1  package match;
2
3  import java.io.*;
4  import java.net.*;
```

```
5  import java.util.regex.*;
6
7  /**
8   * В этой программе отображаются все веб-адреса на веб-странице
9   * путем сопоставления с регулярным выражением, описывающим дескриптор
10  * <a href=...> разметки в коде HTML. Для запуска программы следует ввести:
11  * java match.HrefMatch веб-адрес
12  * @version 1.01 2004-06-04
13  * @author Cay Horstmann
14 */
15 public class HrefMatch
16 {
17   public static void main(String[] args)
18   {
19     try
20     {
21       // извлечь символьную строку с веб-адресом (URL) из командной
22       // строки или использовать URL, выбираемый по умолчанию
23       String urlString;
24       if (args.length > 0) urlString = args[0];
25       else urlString = "http://java.sun.com";
26
27       // открыть поток ввода для чтения URL
28       InputStreamReader in =
29           new InputStreamReader(new URL(urlString).openStream());
30       // прочитать содержимое в построитель символьных строк
31       StringBuilder input = new StringBuilder();
32       int ch;
33       while ((ch = in.read()) != -1)
34         input.append((char) ch);
35
36       // найти все совпадения с шаблоном
37       String patternString =
38           "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>]*)\\s*>";
39       Pattern pattern =
40           Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
41       Matcher matcher = pattern.matcher(input);
42
43       while (matcher.find())
44       {
45         int start = matcher.start();
46         int end = matcher.end();
47         String match = input.substring(start, end);
48         System.out.println(match);
49       }
50     }
51     catch (IOException e)
52     {
53       e.printStackTrace();
54     }
55     catch (PatternSyntaxException e)
56     {
57       e.printStackTrace();
58     }
59   }
60 }
```

Метод `replaceAll()` из класса `Matcher` заменяет все совпадения символов с регулярным выражением символами из замещающей строки. Например, в приведенном ниже фрагменте кода все последовательности цифр заменяются знаком #.

```
Pattern pattern = Pattern.compile("[0-9]+");
Matcher matcher = pattern.matcher(input);
String output = matcher.replaceAll("#");
```

В замещающей строке могут содержаться ссылки на присутствующие в шаблоне группы. Например, ссылка **$n** заменяется *n*-й группой. Для того чтобы включить в текст замены знак **$**, его нужно экранировать с помощью комбинации символов **\$**. Если же имеется символьная строка, содержащая знаки **$** и ****, которые не требуется интерпретировать как замену групп, следует вызвать метод `matcher.replaceAll(Matcher.quoteReplacement(str))`. Метод `replaceFirst()` заменяет только первое совпадение с шаблоном.

И, наконец, в классе `Pattern` имеется также метод `split()`, разбивающий входные данные на массив символьных строк, используя в качестве границ совпадения с регулярным выражением. Например, в приведенном ниже фрагменте кода входные данные разбиваются на лексемы, а в качестве разделителей используются знаки пунктуации, дополнительно (хотя и не обязательно) разделяемые пробелами.

```
Pattern pattern = Pattern.compile("\\s*\\p{Punct}\\s*");
String[] tokens = pattern.split(input);
```

java.util.regex.Pattern 1.4

- `static Pattern compile(String` *`expression`*`)`
- `static Pattern compile(String` *`expression`*`, int` *`flags`*`)`

  Компилируют символьную строку с регулярным выражением в объект шаблона для быстрой обработки совпадений.

Параметры:	*expression*	Регулярное выражение
	flags	Один или более следующих флагов: `CASE_INSENSITIVE`, `UNICODE_CASE`, `MULTILINE`, `UNIX_LINES`, `DOTALL` или `CANON_EQ`

- `Matcher matcher(CharSequence` *`input`*`)`

  Возвращает объект типа `Matcher`, который можно использовать для обнаружения совпадений с шаблоном во входных данных.

- `String[] split(CharSequence` *`input`*`)`
- `String[] split(CharSequence` *`input`*`, int` *`limit`*`)`

  Разбивают строку входных данных на лексемы, причем вид разделителей определяет шаблон. Возвращают массив маркеров. Разделители не являются частью маркеров.

Параметры:	*input*	Символьная строка, разбиваемая на маркеры
	limit	Максимальное количество получаемых строк. Если обнаружены совпадающие разделители в пределах `limit - 1`, в последнем элементе возвращаемого массива размещаются неразбитые входные данные. Если же `limit J 0`, разбиваются все входные данные, а если `limit = 0`, замыкающие пустые строки не вводятся в возвращаемый массив

java.util.regex.Matcher 1.4

- `boolean matches()`

  Возвращает логическое значение `true`, если входные данные совпадают с шаблоном.

- `boolean lookingAt()`

  Возвращает логическое значение `true`, если с шаблоном совпадает начало входных данных.

- `boolean find()`
- `boolean find(int start)`

  Пытаются отыскать следующее совпадение, и если это удается, то возвращают логическое значение `true`.

  Параметры: *start* Индекс, с которого начинается поиск

- `int start()`
- `int end()`

  Возвращают начальную или следующую после конечной позицию текущего совпадения.

- `String group()`

  Возвращает текущее совпадение.

- `int groupCount()`

  Возвращает сведения о количестве групп во входном шаблоне.

- `int start(int groupIndex)`
- `int end(int groupIndex)`

  Возвращают начальную или конечную позицию данной группы в текущем совпадении.

Параметры:	*groupIndex*	Индекс группы (начиная с **1**) или **0** для обозначения полного совпадения

- `String group(int groupIndex)`

  Возвращает символьную строку, совпадающую с заданной группой.

Параметры:	*groupIndex*	Индекс группы (начиная с **1**) или **0** для обозначения полного совпадения

- `String replaceAll(String replacement)`
- `String replaceFirst(String replacement)`

  Возвращают символьную строку, получаемую из входных данных в сопоставителе с шаблоном путем замены всех или только первого совпадения символами из замещающей строки.

Параметры:	`replacement`	Замещающая строка. Может содержать ссылки на группу из шаблона в виде **$n**. Для включения знака **$** служит комбинация **\$**

- `static String quoteReplacement(String str)` **5.0**

  Заключает в кавычки все знаки **** и **$** в символьной строке `str`.

- `Matcher reset()`
- `Matcher reset(CharSequence input)`

  Устанавливают объект типа `Matcher` в исходное состояние. Второй метод вынуждает объект типа `Matcher` обрабатывать другие входные данные. Оба эти метода возвращают ссылку `this` на данный объект.

Из этой главы вы узнали, как выполняются операции ввода-вывода в Java, а также вкратце ознакомились со средствами поддержки регулярных выражений при вводе-выводе. В следующей главе речь пойдет о том, как обрабатываются данные в формате XML.

ГЛАВА 2

XML

В этой главе...

- Общие сведения об XML
- Синтаксический анализ XML-документов
- Проверка достоверности XML-документов
- Поиск информации средствами XPath
- Использование пространств имен
- Потоковые синтаксические анализаторы
- Формирование XML-документов
- Преобразование XML-документов языковыми средствами XSLT

В предисловии к книге *Essential XML* Дона Бокса и др. (Don Box et al.; издательство Addison-Wesley Professional, 2000 г.) говорится, что "...язык XML пришел на смену языку Java, шаблонам проектирования и объектно-ориентированной технологии...". Это, конечно, шутка, но в каждой шутке есть доля правды. В самом деле, язык XML очень удобен для описания и представления структурированных данных, но он не является универсальным средством на все случаи жизни, и для его эффективного использования потребуются также специализированные по предметным областям стандарты и библиотеки. Более того, XML совсем не заменяет, а всего лишь дополняет Java. В конце 1990-х годов компании IBM, Apache и многие другие приступили к созданию на Java библиотек для обработки данных в формате XML. Наиболее важные из этих библиотек вошли в состав платформы Java.

В этой главе описаны основы языка XML, а также инструментальные средства для обработки данных в формате XML, входящие в состав библиотеки Java. Как и прежде, здесь рассматриваются случаи, когда применение XML считается совершенно обоснованным, а также ситуации, в которых можно вполне обойтись и без этого языка, используя другие проверенные временем методики проектирования и программирования.

Введение в XML

В главе 10 первого тома данной книги уже приводились примеры использования *файлов свойств*, описывающих конфигурацию программы. Файл свойств содержит конфигурационные параметры в виде пар, состоящих из имени и значения, как показано ниже.

```
fontname=Times Roman
fontsize=12
windowsize=400 200
color=0 50 100
```

Для считывания такого файла с помощью одного метода можно воспользоваться классом `Properties`. Но это отличное в целом средство не всегда подходит, поскольку во многих случаях формат файла свойств непригоден для описания данных, имеющих сложную структуру. Рассмотрим записи `fontname` и `fontsize` из приведенного выше примера. Удобнее было бы объединить их значения в одном приведенном ниже параметре. Ведь это в большей степени соответствовало бы объектно-ориентированному подходу.

```
font=Times Roman 12
```

Но для синтаксического анализа такого описания шрифта потребуется довольно громоздкий код, поскольку нужно определить, где оканчивается название шрифта и начинается его размер. Дело в том, что файлы свойств имеют так называемую плоскую (двухмерную) иерархию. Иногда программисты предпринимают попытки обойти данное ограничение с помощью составных имен ключей следующим образом:

```
title.fontname=Helvetica
title.fontsize=36
body.fontname=Times Roman
body.fontsize=12
```

Еще один недостаток формата файлов свойств состоит в том, что имена параметров должны быть однозначными. Для хранения последовательности значений придется употребить имена, аналогичные приведенным ниже.

```
menu.item.1=Times Roman
menu.item.2=Helvetica
menu.item.3=Goudy Old Style
```

Подобные недостатки позволяет устранить формат XML. Он служит для представления иерархических структур данных и более гибок по сравнению с плоской табличной структурой файлов свойств. Например, XML-файл с параметрами настройки программы может выглядеть следующим образом:

```
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  <body>
    <font>
      <name>Times Roman</name>
      <size>12</size>
```

```
    </font>
  </body>
  <window>
    <width>400</width>
    <height>200</height>
  </window>
  <color>
    <red>0</red>
    <green>50</green>
    <blue>100</blue>
  </color>
  <menu>
    <item>Times Roman</item>
    <item>Helvetica</item>
    <item>Goudy Old Style</item>
  </menu>
</configuration>
```

Формат XML позволяет без особых затруднений выражать любую иерархическую структуру данных с повторяющимися элементами. XML-файл имеет очень простую и ясную структуру, напоминающую структуру HTML-файла. Дело в том, что оба языка, XML и HTML, созданы на основе стандартного обобщенного языка разметки SGML (Standard Generilized Markup Language).

Язык SGML в 1970-х годах использовался для описания структуры сложных документов в некоторых отраслях промышленности с высокими требованиями к документации, например, в авиастроении. Но из-за присущей ему сложности SGML так и не получил широкого распространения. Основные трудности в употреблении этого языка возникали из-за наличия двух противоречивых целей. С одной стороны, документы должны оформляться в строгом соответствии с правилами, а с другой — необходимо обеспечить простоту и высокую скорость ввода данных с помощью клавиатурных сокращений. Язык XML разработан в виде упрощенной версии SGML для Интернета. И как часто бывает в жизни, чем проще, тем лучше. Поэтому XML сразу же был с большим энтузиазмом воспринят теми специалистами, которые многие годы избегали употреблять SGML.

НА ЗАМЕТКУ! Очень удачно составленное описание стандарта XML можно найти по адресу `www.xml.com/axml/axml.html`.

Несмотря на общие корни, у языков XML и HTML имеется ряд существенных различий.

- В отличие от HTML, в XML учитывается регистр символов, поэтому дескрипторы `<H1>` и `<h1>` в XML считаются разными.
- В HTML некоторые закрывающие дескрипторы могут отсутствовать. Например, составитель HTML-документа может пропустить дескриптор `</p>` или `</li>`, если из контекста ясно, где заканчивается абзац или пункт списка. А в XML это не разрешается.
- Для элементов разметки без тела в XML предусмотрена сокращенная запись открывающего дескриптора, совмещенного с закрывающим. В этом случае открывающий дескриптор заканчивается знаком **/**, например **<img**

src="coffeecup.png"/>. Это означает, что наличие закрывающего дескриптора </img> подразумевается по умолчанию.

- В XML значения атрибутов должны быть заключены в кавычки, а в HTML кавычки могут отсутствовать. Например, дескриптор <applet code="MyApplet.class" width=300 height=300> можно использовать в HTML, но нельзя в XML, где значения атрибутов width и height нужно обязательно заключить в кавычки следующим образом: width="300" и height="300".
- В HTML допускается указывать имена атрибутов без их значений, например <input type="radio" name="language" value="Java" checked>. А в XML все атрибуты должны быть непременно указаны со своими значениями, например checked="true" или checked="checked".

Структура XML-документа

XML-документ должен начинаться с одного из следующих заголовков:

```
<?xml version="1.0"?>
```

или

```
<?xml version="1.0" encoding="UTF-8"?>
```

НА ЗАМЕТКУ! Язык SGML предназначался для обработки документов, поэтому XML-файлы принято называть *документами*, хотя многие XML-файлы описывают такие наборы данных, для которых этот термин не совсем подходит.

Строго говоря, указывать заголовок совсем не обязательно, но все же настоятельно рекомендуется включать его в состав документа. После заголовка обычно следует *определение типа документа* (DTD), как показано ниже.

```
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

Представляя собой важный механизм обеспечения корректности документа, DTD, тем не менее, не является обязательным элементом XML-документа. Более подробно DTD рассматривается далее в этой главе.

И наконец, тело XML-документа содержит *корневой элемент*, который может состоять из других элементов, как показано ниже.

```
<?xml version="1.0"?>
<!DOCTYPE configuration . . .>
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  . . .
</configuration>
```

Каждый элемент разметки может содержать *дочерние элементы*, текст или и то и другое. В приведенном выше примере элемент разметки `font` состоит из двух дочерних элементов: `name` и `size`, причем элемент `name` содержит текст `"Helvetica"`.

СОВЕТ. XML-документы рекомендуется составлять таким образом, чтобы элементы разметки содержали дочерние элементы или текст. Иначе говоря, следует избегать разметки, аналогичной приведенной ниже.

```
<font>
  Helvetica
  <size>36</size>
</font>
```

В спецификации XML такая разметка называется *смешанным содержимым*. Как станет ясно в дальнейшем, синтаксический анализ XML-документа намного упрощается, если избегать в нем смешанного содержимого.

Элементы разметки XML-документов могут содержать атрибуты, как показано ниже.

```
<size unit="pt">36</size>
```

Среди разработчиков XML нет единого мнения о том, когда следует употреблять элементы, а когда — атрибуты. Например, описать шрифт, по-видимому, проще следующим образом:

```
<font name="Helvetica" size="36"/>
```

чем так, как показано ниже.

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

Но в то же время атрибуты гораздо менее удобны. Допустим, в определение размера шрифта нужно добавить единицу измерения. Если использовать для этой цели атрибуты, то единицы измерения придется указать рядом со значением атрибута следующим образом:

```
<font name="Helvetica" size="36 pt"/>
```

Но это означает, что придется написать дополнительный код для синтаксического анализа символьной строки `"36 pt"`, а именно этого стремились избежать создатели XML. Поэтому более простым решением было бы применение атрибута в элементе `size`, как показано ниже.

```
<font>
  <name>Helvetica</name>
  <size unit="pt">36</size>
</font>
```

Широко распространенное эмпирическое правило гласит: атрибуты следует использовать не для указания значений, а только при изменении их интерпретации. Если же непонятно, обозначает ли какой-нибудь атрибут изменение интерпретации значения или нет, то лучше отказаться от атрибута и употребить элемент разметки. Во многих полезных XML-документах атрибуты вообще не употребляются.

НА ЗАМЕТКУ! В HTML существует очень простое правило употребления атрибутов: если данные не отображаются на веб-странице, значит, это атрибут. Рассмотрим следующую гипертекстовую ссылку:

```
<a href="http://java.sun.com">Java Technology</a>
```

Строка "Java Technology" отображается на веб-странице, но URL этой гипертекстовой ссылки не выводится. Впрочем, это правило не совсем подходит для XML-файлов, поскольку данные в XML-файле не всегда предназначены непосредственно для просмотра в удобочитаемом виде.

Элементы разметки и текст являются основными составляющими XML-документов, но в них можно также встретить и ряд других инструкций разметки.

- *Ссылки на символы* в виде `&#десятичное_значение;` или `&#хшестнадцатеричное_значение;`. Например, ссылка `é` или `Ù` обозначает символ **é**.
- *Ссылки на сущности* в виде `&имя;`. Так, ссылки на сущности `<`, `>`, `&`, `"`, `'` имеют предопределенные значения и соответствуют знакам **<**, **>**, **&**, **"** и **'**. В DTD можно также указать другие ссылки на сущности.
- *Разделы CDATA*, разграничиваемые группами символов `<![CDATA[` и `]]>`. Они предназначены для включения строк со знаками **<**, **>** или **&**, которые не следует интерпретировать как символы разметки, как в приведенном ниже примере.

  ```
  <![CDATA[< & > мои излюбленные разделители]]>
  ```

- В разделах CDATA не допускается использование символьных строк вроде **"]]>"**, поэтому их следует употреблять очень осторожно. Зачастую они выполняют функции своего рода *"лазейки"* для внедрения в XML-документ данных в устаревшем формате.
- *Инструкции обработки* — это инструкции для прикладных программ, обрабатывающих XML-документы. Такие инструкции разграничиваются символами **<?** и **?>**, как в приведенном ниже примере.

  ```
  <?xml-stylesheet href="mystyle.css" type="text/css"?>
  ```

- Каждый XML-документ начинается со следующей инструкции обработки:

  ```
  <?xml version="1.0"?>
  ```

- *Комментарии*, разграничиваемые символами **<!--** и **-->** следующим образом:

  ```
  <!-- Это комментарий. -->
  ```

- В комментариях не допускается использовать символьные строки вроде `"--"`. Комментарии предназначены для пользователей, и поэтому в них не следует вводить скрытые команды. Для выполнения команд предназначены инструкции обработки.

Синтаксический анализ XML-документов

Для обработки XML-документа необходимо выполнить его *синтаксический анализ*. *Синтаксическим анализатором* называется программа, которая считывает файл, подтверждает корректность его формата, разбивает данные на составные элементы и предоставляет программисту доступ к ним. Ниже приведены две основные разновидности XML-анализаторов.

- Древовидные анализаторы (например, DOM-анализатор, т.е. анализатор объектной модели документа), которые считывают XML-документ и представляют его в виде древовидной структуры.
- Потоковые анализаторы (например, SAX-анализаторы — простые анализаторы прикладного интерфейса API для XML), которые генерируют события по мере чтения XML-документа.

DOM-анализатор проще в употреблении, поэтому сначала рассматривается именно он. А потоковый анализатор обычно применяется для обработки длинных документов, когда для древовидного представления XML-данных требуется большой объем памяти. Кроме того, его можно употреблять для извлечения отдельных элементов XML-документа без учета контекста. Подробнее об этом — далее, в разделе "Потоковые синтаксические анализаторы".

Интерфейс DOM-анализатора стандартизован консорциумом W3C. Так, пакет `org.w3c.dom` содержит определения типов интерфейсов, в том числе `Document` и `Element`. Различные поставщики, среди которых компании IBM и Apache, разработали собственные варианты DOM-анализаторов, реализующие эти интерфейсы. В прикладном интерфейсе API обработки XML-документов на Java (JAXP) предусмотрена возможность подключения таких анализаторов. Кроме того, в состав пакета JDK входит собственный DOM-анализатор. Именно он и рассматривается далее в этой главе.

Для чтения XML-документа сначала потребуется объект типа `DocumentBuilder`, который можно получить из класса `DocumentBuilderFactory` следующим образом:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

А затем можно приступать к чтению данных из файла, как показано ниже.

```
File f = . . .
Document doc = builder.parse(f);
```

С другой стороны, XML-документ можно прочитать и по указанному URL, как в следующем примере кода:

```
URL u = . . .
Document doc = builder.parse(u);
```

Для чтения XML-документа можно даже указать произвольный поток ввода следующим образом:

```
InputStream in = . . .
Document doc = builder.parse(in);
```

НА ЗАМЕТКУ! Если в качестве источника данных служит произвольный поток ввода, синтаксический анализатор не сможет найти те файлы, расположение которых указано относительно данного документа, например, DTD-файл, находящийся в том же каталоге. Для преодоления этого препятствия достаточно установить так называемый "определитель сущностей".

Объект типа `Document` является внутренним представлением древовидной структуры XML-документа. Он состоит из экземпляров классов, реализующих интерфейс `Node` (Узел) и различные, производные от него интерфейсы. На рис. 2.1 показана иерархия наследования интерфейса `Node`.

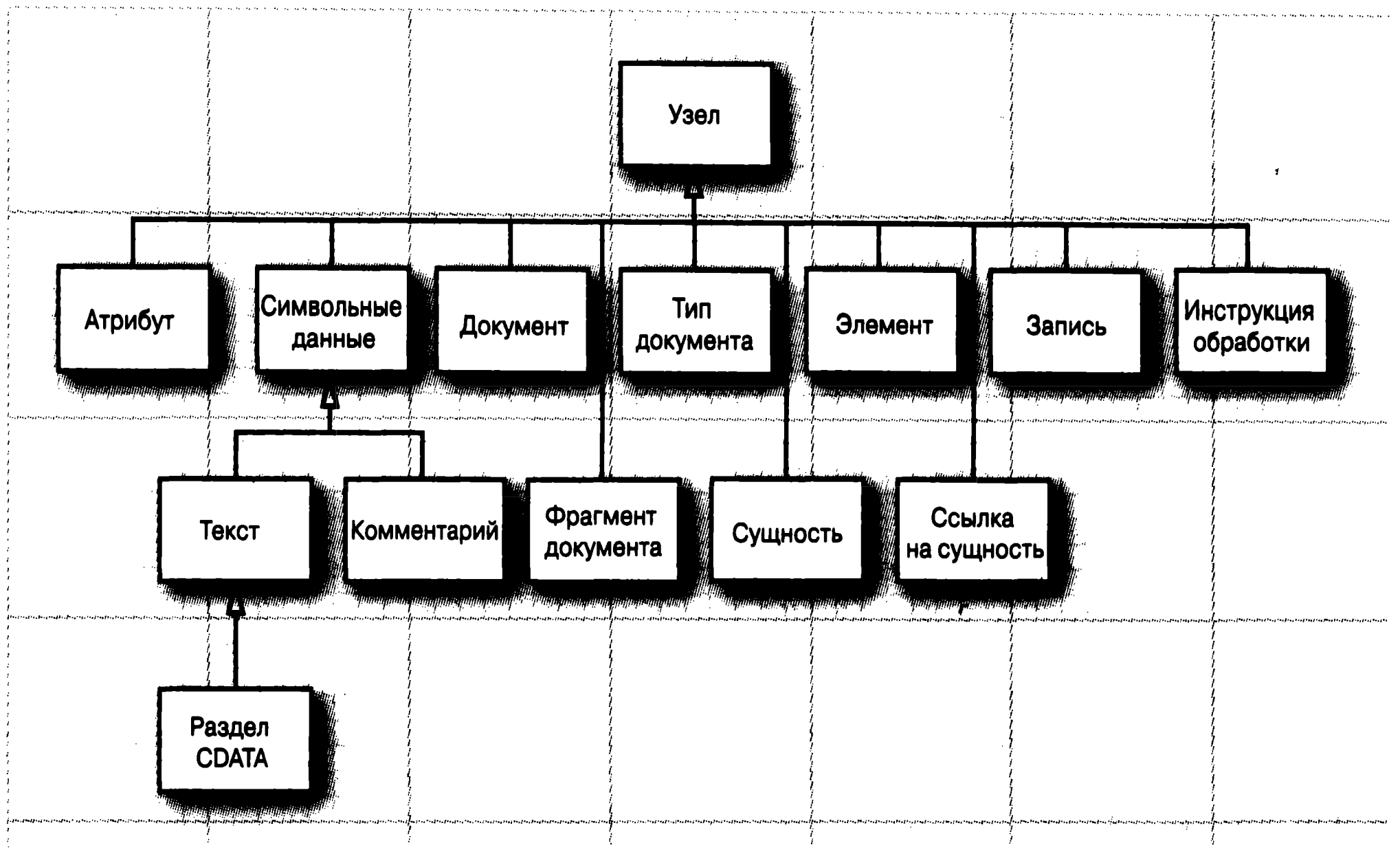


Рис. 2.1. Иерархия наследования интерфейса `Node`

Анализ содержимого документа начинается с вызова метода `getDocument Element()`, который возвращает корневой элемент, как показано ниже.

```
Element root = doc.getDocumentElement();
```

Так, если обрабатывается приведенный ниже XML-документ, то в результате вызова метода `getDocumentElement()` будет получен элемент разметки `font`.

```
<?xml version="1.0"?>
<font>
  . . .
</font>
```

Для извлечения элементов, дочерних по отношению к данному (ими могут быть подчиненные элементы, текст, комментарии или другие узлы), служит метод `getChildNodes()`, возвращающий набор данных типа `NodeList`. Этот тип существовал еще до создания стандартной библиотеки коллекций в Java, и поэтому для работы с ним применяется специальный протокол. Метод `item()` возвращает элемент набора данных по указанному индексу, а метод `getLength()` — общее количество элементов. Таким образом, для перечисления всех дочерних элементов можно воспользоваться следующим кодом:

```
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++)
{
  Node child = children.item(i);
  . . .
}
```

Анализ дочерних элементов следует выполнять очень внимательно. На первый взгляд, приведенный ниже XML-документ содержит два элемента, дочерних по отношению к `font`.

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

Но синтаксический анализатор сообщит, что в разметке данного XML-документа имеется пять дочерних элементов.

- Разделитель между дескрипторами `<font>` и `<name>`.
- Элемент `name`.
- Разделитель между дескрипторами `</name>` и `<size>`.
- Элемент `size`.
- Разделитель между дескрипторами `</size>` и `</font>`.

На рис. 2.2 схематически представлено дерево DOM — объектной модели упомянутого выше документа.

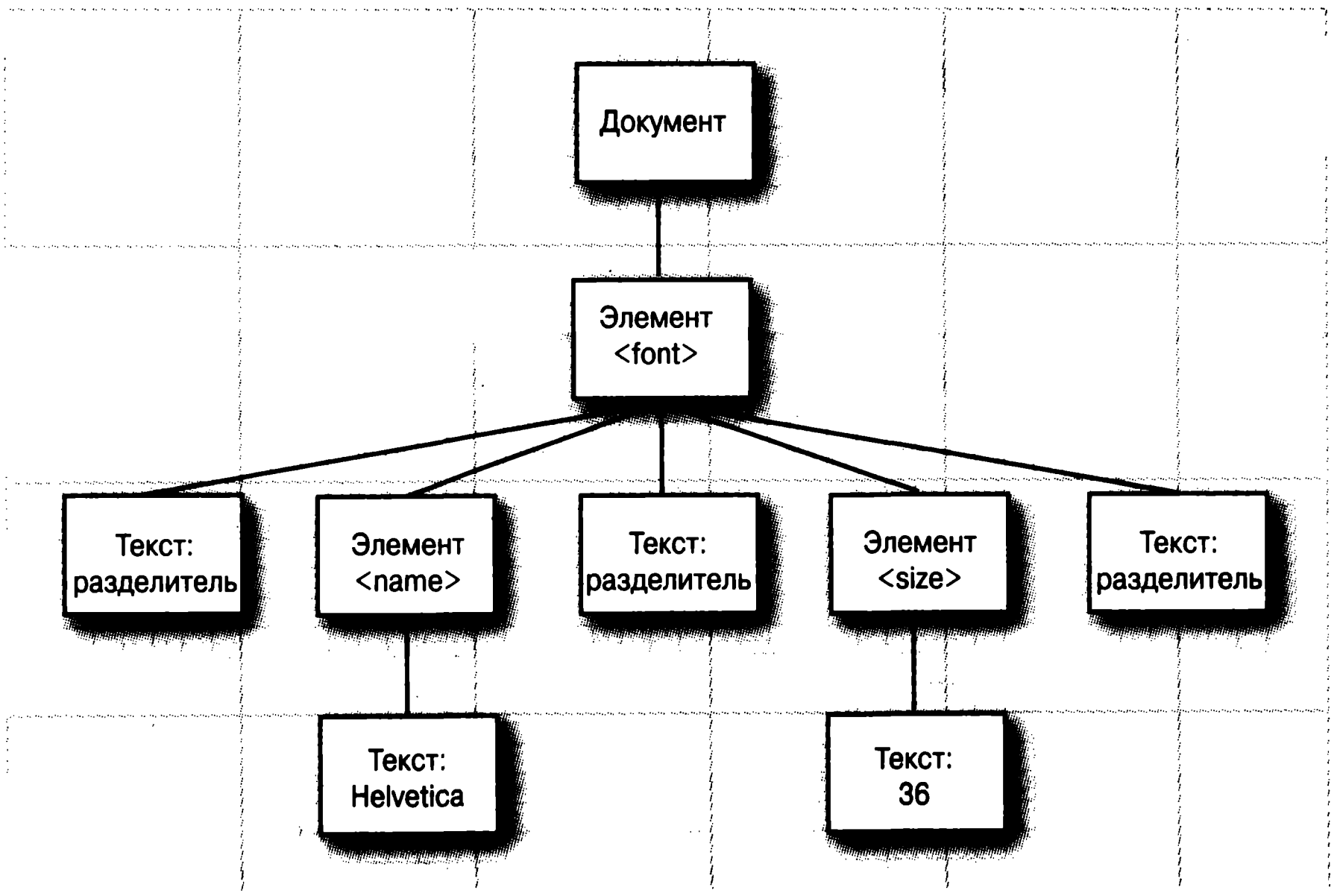


Рис. 2.2. Простое дерево DOM

Если требуется обработать только подчиненные элементы, нужно игнорировать все разделители. Это можно сделать с помощью приведенного ниже кода. В итоге будут выявлены только элементы с именами дескрипторов `name` и `size`.

```
for (int i = 0; i < children.getLength(); i++)
{
  Node child = children.item(i);
  if (child instanceof Element)
  {
```

```
    Element childElement = (Element) child;
    . . .
  }
}
```

Как будет показано в следующем разделе, данную задачу можно решить еще лучше, если воспользоваться DTD. В таком случае синтаксическому анализатору будет известно, у каких именно элементов отсутствуют текстовые узлы в качестве дочерних элементов. Благодаря этому он может подавить разделители автоматически.

При анализе элементов `name` и `size` придется извлечь содержащиеся в них текстовые строки, которые находятся в дочерних узлах типа `Text`. А поскольку заранее известно, что другие дочерние узлы отсутствуют, то можно вызвать метод `getFirstChild()` без перебора содержимого очередной коллекции типа `NodeList`. После этого с помощью метода `getData()` можно извлечь текстовую строку из узла типа `Text`, как показано в приведенном ниже фрагменте кода.

```
for (int i = 0; i < children.getLength(); i++)
{
  Node child = children.item(i);
  if (child instanceof Element)
  {
    Element childElement = (Element) child;
    Text textNode = (Text) childElement.getFirstChild();
    String text = textNode.getData().trim();
    if (childElement.getTagName().equals("name"))
      name = text;
    else if (childElement.getTagName().equals("size"))
      size = Integer.parseInt(text);
  }
}
```

СОВЕТ. По значению, возвращаемому в результате выполнения метода `getData()`, рекомендуется вызвать метод `trim()`. Допустим, составитель XML-документа разместил открывающие и закрывающие дескрипторы в отдельных строках, как показано ниже.

```
<size>
  36
</size>
```

В таком случае синтаксический анализатор включит все пробелы и символы перевода строк в данные из текстового узла, а метод `trim()` удалит их и оставит лишь конкретные данные.

Для извлечения последнего дочернего узла можно воспользоваться методом `getLastChild()`, а для получения следующего родственного узла — методом `getNextSibling()`. С помощью этих методов можно обойти дочерние узлы другим способом, как показано ниже.

```
for (Node childNode = element.getFirstChild();
  childNode != null;
  childNode = childNode.getNextSibling())
{
  . . .
}
```

Для перечисления атрибутов узла следует вызвать метод `getAttributes()`, возвращающий объект типа `NamedNodeMap`, содержащий объекты типа `Node`, описывающие атрибуты. Обход узлов именованного отображения узлов типа `NamedNodeMap`

можно выполнить таким же образом, как и обход списка узлов типа `NodeList`. В таком случае для извлечения имен атрибутов и их значений следует воспользоваться методами `getNodeName()` и `getNodeValue()`, как показано в приведенном ниже фрагменте кода.

```
NamedNodeMap attributes = element.getAttributes();
for (int i = 0; i < attributes.getLength(); i++)
{
  Node attribute = attributes.item(i);
  String name = attribute.getNodeName();
  String value = attribute.getNodeValue();
  . . .
}
```

С другой стороны, если известно имя атрибута, его значение можно извлечь непосредственно следующим образом:

```
String unit = element.getAttribute("unit");
```

Описанный выше способ анализа дерева DOM демонстрируется в примере программы, исходный код которой приведен в листинге 2.1. Для чтения XML-документа следует выбрать пункт меню File⇨Open (Файл⇨Открыть) и указать в диалоговом окне требуемый файл. После этого объект типа `DocumentBuilder` выполнит синтаксический анализ XML-данных и создаст объект типа `Document`. А далее этот объект будет представлен в рабочем окне программы в виде древовидной структуры (рис. 2.3).

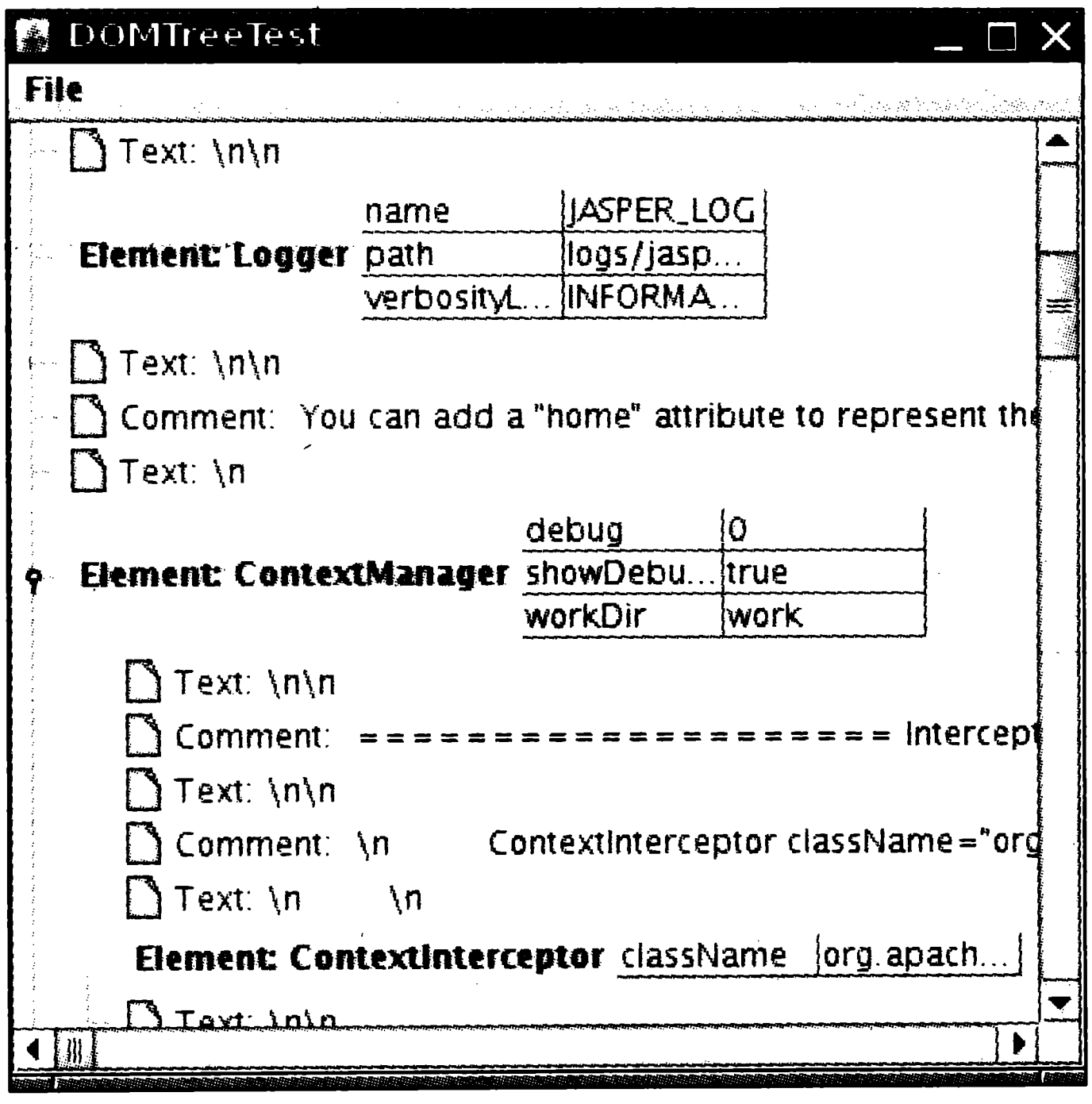


Рис. 2.3. Древовидная структура XML-документа, полученная в результате его синтаксического анализа

В данной древовидной структуре представлены дочерние узлы, окруженные разделителями и комментариями. Для большей наглядности программа отображает все символы перевода строки и возврата каретки в виде комбинаций **\n** и **\r**. (В противном случае они были бы представлены в виде пустых квадратов, используемых в библиотеке Swing для обозначения символов, которые нельзя воспроизвести.)

В данной программе применяются рассматриваемые в главе 6 способы отображения дерева и таблиц атрибутов. Класс `DOMTreeModel` реализует интерфейс `TreeModel`. Метод `getRoot()` возвращает корневой элемент документа, а метод `getChild()` извлекает список дочерних узлов и возвращает узел с заданным индексом. Средство воспроизведения ячеек дерева отображает следующее.

- Для элементов разметки: имя дескриптора и все атрибуты.
- Для символьных данных: интерфейс (`Text`, `Comment` или `CDATASection`), затем данные, где символы перевода строки и возврата каретки представлены в виде комбинаций **\n** и **\r**.
- Для других типов узлов: имя класса, после которого следует результат выполнения метода `toString()`.

Листинг 2.1. Исходный код из файла `dom/TreeViewer.java`

```
1   package dom;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.io.*;
6   import javax.swing.*;
7   import javax.swing.event.*;
8   import javax.swing.table.*;
9   import javax.swing.tree.*;
10  import javax.xml.parsers.*;
11  import org.w3c.dom.*;
12  import org.w3c.dom.CharacterData;
13  /**
14   * В этой программе XML-документ отображается в виде дерева
15   * @version 1.12 2012-06-03
16   * @author Cay Horstmann
17  */
18  public class TreeViewer
19  {
20    public static void main(String[] args)
21    {
22      EventQueue.invokeLater(new Runnable()
23       {
24         public void run()
25        {
26           JFrame frame = new DOMTreeFrame();
27           frame.setTitle("TreeViewer");
28           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29           frame.setVisible(true);
30        }
31       });
32    }
33  }
34
```

```
35  /**
36   * Этот фрейм содержит дерево, отображающее содержимое XML-документа
37  */
38  class DOMTreeFrame extends JFrame
39  {
40    private static final int DEFAULT_WIDTH = 400;
41    private static final int DEFAULT_HEIGHT = 400;
42
43    private DocumentBuilder builder;
44
45    public DOMTreeFrame()
46    {
47      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
48
49      JMenu fileMenu = new JMenu("File");
50      JMenuItem openItem = new JMenuItem("Open");
51      openItem.addActionListener(new ActionListener()
52       {
53         public void actionPerformed(ActionEvent event)
54         {
55           openFile();
56         }
57       });
58      fileMenu.add(openItem);
59
60      JMenuItem exitItem = new JMenuItem("Exit");
61      exitItem.addActionListener(new ActionListener()
62       {
63         public void actionPerformed(ActionEvent event)
64         {
65           System.exit(0);
66         }
67       });
68      fileMenu.add(exitItem);
69
70      JMenuBar menuBar = new JMenuBar();
71      menuBar.add(fileMenu);
72      setJMenuBar(menuBar);
73    }
74
75    /**
76     * Открыть файл и загрузить документ
77    */
78    public void openFile()
79    {
80      JFileChooser chooser = new JFileChooser();
81      chooser.setCurrentDirectory(new File("dom"));
82
83      chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
84        {
85          public boolean accept(File f)
86          {
87            return f.isDirectory() ||
88                f.getName().toLowerCase().endsWith(".xml");
89          }
90
91          public String getDescription()
92          {
```

```
93            return "XML files";
94          }
95        });
96      int r = chooser.showOpenDialog(this);
97      if (r != JFileChooser.APPROVE_OPTION) return;
98      final File file = chooser.getSelectedFile();
99
100     new SwingWorker<Document, Void>()
101       {
102         protected Document doInBackground() throws Exception
103         {
104           if (builder == null)
105           {
106             DocumentBuilderFactory factory =
107                 DocumentBuilderFactory.newInstance();
108             builder = factory.newDocumentBuilder();
109           }
110           return builder.parse(file);
111         }
112         protected void done()
113         {
114           try
115           {
116             Document doc = get();
117             JTree tree = new JTree(new DOMTreeModel(doc));
118             tree.setCellRenderer(new DOMTreeCellRenderer());
119
120             setContentPane(new JScrollPane(tree));
121             validate();
122           }
123           catch (Exception e)
124           {
125             JOptionPane.showMessageDialog(DOMTreeFrame.this, e);
126           }
127         }
128       }.execute();
129     }
130
131   }
132
133   /**
134    * Эта модель дерева описывает древовидную структуру XML-документа
135   */
136   class DOMTreeModel implements TreeModel
137   {
138     private Document doc;
139
140   /**
141    * Строит модель дерева для документа
142    * @param doc Документ
143   */
144   public DOMTreeModel(Document doc)
145   {
146     this.doc = doc;
147   }
148
149   public Object getRoot()
150   {
```

```
151     return doc.getDocumentElement();
152   }
153
154   public int getChildCount(Object parent)
155   {
156     Node node = (Node) parent;
157     NodeList list = node.getChildNodes();
158     return list.getLength();
159   }
160   public Object getChild(Object parent, int index)
161   {
162     Node node = (Node) parent;
163     NodeList list = node.getChildNodes();
164     return list.item(index);
165   }
166
167   public int getIndexOfChild(Object parent, Object child)
168   {
169     Node node = (Node) parent;
170     NodeList list = node.getChildNodes();
171     for (int i = 0; i < list.getLength(); i++)
172       if (getChild(node, i) == child) return i;
173     return -1;
174   }
175
176   public boolean isLeaf(Object node)
177   {
178     return getChildCount(node) == 0;
179   }
180
181   public void valueForPathChanged(TreePath path, Object newValue)
182   {
183   }
184
185   public void addTreeModelListener(TreeModelListener l)
186   {
187   }
188
189   public void removeTreeModelListener(TreeModelListener l)
190   {
191   }
192
193 }
194
195 /**
196  * Этот класс воспроизводит узел XML-документа
197 */
198 class DOMTreeCellRenderer extends DefaultTreeCellRenderer
199 {
200   public Component getTreeCellRendererComponent(
201      JTree tree, Object value,boolean selected, boolean expanded,
202      boolean leaf, int row, boolean hasFocus)
203   {
204     Node node = (Node) value;
205     if (node instanceof Element) return elementPanel((Element) node);
206
207     super.getTreeCellRendererComponent(
208         tree, value, selected, expanded, leaf, row, hasFocus);
```

```
209     if (node instanceof CharacterData)
210        setText(characterString((CharacterData) node));
211     else setText(node.getClass() + ": " + node.toString());
212     return this;
213   }
214
215   public static JPanel elementPanel(Element e)
216   {
217     JPanel panel = new JPanel();
218     panel.add(new JLabel("Element: " + e.getTagName()));
219     final NamedNodeMap map = e.getAttributes();
220     panel.add(new JTable(new AbstractTableModel()
221       {
222         public int getRowCount()
223         {
224           return map.getLength();
225         }
226
227         public int getColumnCount()
228         {
229           return 2;
230         }
231
232         public Object getValueAt(int r, int c)
233         {
234       \   return c ==
235             0 ? map.item(r).getNodeName() : map.item(r).getNodeValue();
236         }
237       }));
238     return panel;
239   }
240
241   public static String characterString(CharacterData node)
242   {
243     StringBuilder builder = new StringBuilder(node.getData());
244     for (int i = 0; i < builder.length(); i++)
245     {
246       if (builder.charAt(i) == '\r')
247       {
248         builder.replace(i, i + 1, "\\r");
249         i++;
250       }
251       else if (builder.charAt(i) == '\n')
252       {
253         builder.replace(i, i + 1, "\\n");
254         i++;
255       }
256       else if (builder.charAt(i) == '\t')
257       {
258         builder.replace(i, i + 1, "\\t");
259         i++;
260       }
261     }
262     if (node instanceof CDATASection) builder.insert(0, "CDATASection: ");
263     else if (node instanceof Text) builder.insert(0, "Text: ");
264     else if (node instanceof Comment) builder.insert(0, "Comment: ");
265
266     return builder.toString();
```

```
267    }
268 }
```

javax.xml.parsers.DocumentBuilderFactory 1.4

- `static DocumentBuilderFactory newInstance()`

  Возвращает экземпляр класса `DocumentBuilderFactory`.

- `DocumentBuilder newDocumentBuilder()`

  Возвращает экземпляр класса `DocumentBuilder`.

javax.xml.parsers.DocumentBuilder 1.4

- `Document parse(File f)`
- `Document parse(String url)`
- `Document parse(InputStream in)`

  Выполняют синтаксический анализ XML-документа, полученного из заданного файла, по указанному URL или из заданного потока ввода. Возвращают результат синтаксического анализа.

org.w3c.dom.Document 1.4

- `Element getDocumentElement()`

  Возвращает корневой элемент разметки документа.

org.w3c.dom.Element 1.4

- `String getTagName()`

  Возвращает имя элемента разметки.

- `String getAttribute(String name)`

  Возвращает значение атрибута с заданным именем или пустую символьную строку, если такой атрибут отсутствует.

org.w3c.dom.Node 1.4

- `NodeList getChildNodes()`

  Возвращает список, содержащий все дочерние узлы данного узла.

- `Node getFirstChild()`
- `Node getLastChild()`

  Возвращают первый или последний дочерний узел данного узла. Если у данного узла отсутствуют дочерние узлы, возвращается пустое значение `null`.

- `Node getNextSibling()`
- `Node getPreviousSibling()`

  Возвращает предыдущий родственный узел. Если у данного узла отсутствуют родственные узлы, возвращается пустое значение `null`.

- `Node getParentNode()`

  Возвращает родительский узел данного узла или пустое значение `null`, если данный узел является узлом документа.

- `NamedNodeMap getAttributes()`

  Возвращает отображение узлов, содержащее узлы типа `Attr` с описаниями всех атрибутов данного узла.

- `String getNodeName()`

  Возвращает имя данного узла. Если узел относится к типу `Attr`, то возвращается имя атрибута.

- `String getNodeValue()`

  Возвращает значение данного узла. Если узел относится к типу `Attr`, то возвращается значение атрибута.

org.w3c.dom.CharacterData 1.4

- `String getData()`

  Возвращает текст, хранящийся в данном узле.

org.w3c.dom.NodeList 1.4

- `int getLength()`

  Возвращает количество узлов в данном списке.

- `Node item(int index)`

  Возвращает узел с заданным индексом. Значение индекса может быть от **0** до **`getLength()` - 1**.

org.w3c.dom.NamedNodeMap 1.4

- `int getLength()`

  Возвращает количество узлов в данном отображении.

- `Node item(int index)`

  Возвращает узел с заданным индексом. Значение индекса может быть от **0** до **`getLength()` -1**.

Проверка достоверности XML-документов

В предыдущем разделе описан способ обхода древовидной структуры DOM-документа. Но если следовать этому способу непосредственно, то потребуется приложить немало усилий для проверки ошибок программным путем. В этом случае придется не только организовать поиск и удаление лишних разделителей между элементами, но и проверить, содержит ли документ предполагаемые узлы. Рассмотрим в качестве примера следующий элемент разметки:

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

При чтении первого же дочернего узла неожиданно обнаруживается, что это текстовый узел, содержащий разделитель `"\n"`. Пропуская текстовые узлы, нетрудно дойти до узла первого элемента, где нужно проверить, имеет ли его дескриптор имя `name`. Затем требуется выяснить, имеет ли он дочерний узел типа `Text`. После этого можно переместиться к следующему дочернему узлу без разделителя и выполнить такую же проверку. Но что делать, если составитель XML-документа изменит порядок расположения дочерних узлов или добавит еще один дочерний элемент? С одной стороны, для проверки всех возможных ошибок придется написать очень громоздкий код, а с другой — исключить такую проверку было бы слишком опрометчиво.

Правда, к числу главных преимуществ XML-анализатора относится его способность автоматически проверять корректность структуры документа. В таком случае анализ XML-документа значительно упрощается. Так, если известно, что фрагмент `font` успешно прошел проверку, то несложно получить два дочерних узла, выполнить приведение к типу `Text`, а затем извлечь текстовые данные без дополнительной проверки.

Для указания структуры документа можно предоставить определение DTD или XML Schema. Определение DTD или XML Schema содержит правила, регламентирующие структуру документа. Оно задает допустимые дочерние узлы элементов и атрибутов каждого элемента. Например, определение DTD может содержать следующее правило:

```
<!ELEMENT font (name,size)>
```

Это правило выражает следующее ограничение: у элемента разметки `font` всегда должны быть два дочерних узла: `name` и `size`. А средствами языка XML Schema то же самое ограничение записывается следующим образом:

```
<xsd:element name="font">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
  </xsd:sequence>
</xsd:element>
```

Язык XML Schema позволяет формулировать более изощренные условия проверки достоверности, чем определения DTD. Например, элемент разметки `size` должен содержать целочисленное значение. В отличие от DTD, в XML Schema используется синтаксис XML, что упрощает обработку файлов со схемами XML-документов.

Язык XML Schema был изначально разработан для замены определений DTD. Но на момент написания данной книги определения DTD все еще широко применялись.

Язык XML Schema довольно сложный, и далеко не все анализаторы способны его поддерживать. На самом деле некоторых пользователей настолько путает сложность XML Schema, что они предпочитают применять альтернативные средства проверки достоверности документов. В качестве примера такого средства можно привести язык Relax NG (`http://www.relaxng.org`).

В следующем разделе подробно обсуждаются определения DTD, а затем вкратце рассматриваются основные средства поддержки XML Schema. После этого будет представлен пример, наглядно демонстрирующий, насколько проверка достоверности XML-документов упрощает их обработку.

Определения типов документов

Существует несколько способов предоставить определение типа документа (DTD). В частности, DTD можно ввести в начале XML-документа так, как показано ниже.

```
<?xml version="1.0"?>
<!DOCTYPE configuration [
  <!ELEMENT configuration . . .>
  другие правила
  . . .
]>
<configuration>
  . . .
</configuration>
```

Как видите, эти правила заключаются в квадратные скобки объявления `DOCTYPE`. Тип документа должен соответствовать имени корневого элемента (в данном примере — `configuration`). Размещать DTD в самом документе неудобно, поскольку такие определения могут быть очень длинными. Следовательно, DTD имеет смысл хранить в отдельном файле. А для связывания DTD с XML-документами можно использовать приведенные ниже объявления `SYSTEM`, в которых указываются имена внешних файлов.

```
<!DOCTYPE configuration SYSTEM "config.dtd">
```

или

```
<!DOCTYPE configuration SYSTEM "http://myserver.com/config.dtd">
```

ВНИМАНИЕ! Если для указания внешнего файла, содержащего DTD, служит относительный URL (например, `"config.dtd"`), то вместо потока ввода типа `InputStream` синтаксическому анализатору следует предоставить объект типа `File` или `URL`. Если же требуется синтаксический анализ данных из потока ввода, то синтаксическому анализатору нужно предоставить определитель сущностей, как поясняется в следующем далее примечании.

И наконец, можно воспользоваться механизмом обозначения хорошо известных определений DTD, унаследованным из языка SGML, как в приведенном ниже примере. Если XML-процессору известен способ обнаружения DTD с помощью идентификатора `PUBLIC`, то обращаться по указанному URL не обязательно.

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

НА ЗАМЕТКУ! Если используется DOM-анализатор и требуется организовать поддержку идентификатора PUBLIC, следует вызвать метод `setEntityResolver()` из класса `DocumentBuilder`. С помощью этого метода устанавливается экземпляр класса, реализующего интерфейс `EntityResolver`. В данном интерфейсе определен единственный метод `resolveEntity()`. Ниже приведена типичная реализация этого метода.

```
class MyEntityResolver implements EntityResolver
{
  public InputSource resolveEntity(String publicID,
    String systemID)
  {
    if (publicID.equals(известный идентификатор))
      return new InputSource(данные DTD);
    else
      return null; // использовать стандартное поведение
  }
}
```

Источник вводимых данных можно создать из класса `InputStream`, `Reader` или `String`.

Рассмотрим теперь различные правила, которые могут задаваться в DTD. Правило `ELEMENT` задает дочерние узлы данного элемента в виде регулярного выражения, составляющие которого перечислены в табл. 2.1.

Таблица 2.1. Правила для содержимого документа

Правило	Назначение
E*	0 или больше элементов ***E***
***E**+*	1 или больше элементов ***E***
***E**?*	0 или 1 элемент ***E***
$E_1 \mid E_2 \mid \ldots \mid E_n$	Один из элементов $E_1, E_2, \ldots E_n$
$E_1, E_2, \ldots E_n$	Элемент E_1, после которого следуют элементы $E_2, \ldots E_n$
#PCDATA	Текст
$(\#PCDATA \mid E_1 \mid E_2 \mid \ldots \mid E_n)^*$	0 или больше вхождений текста и элементы $E_1, E_2, \ldots E_n$, следующие в любом порядке (смешанное содержимое)
ANY	Любой дочерний узел
EMPTY	Дочерние узлы отсутствуют

Рассмотрим несколько простых примеров. Следующее правило указывает на то, что элемент разметки `menu` может содержать **0** или больше элементов **item**:

```
<!ELEMENT menu (item)*>
```

Согласно приведенным ниже правилам, шрифт описывается именем, после которого следует размер шрифта. Имя и размер шрифта являются текстовыми элементами.

```
<!ELEMENT font (name,size)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT size (#PCDATA)>
```

Сокращение `PCDATA` обозначает проанализированные символьные данные. Данные называются *проанализированными*, поскольку синтаксический анализатор обрабатывает текстовую строку и ищет знак **<**, обозначающий начало нового дескриптора, или же знак **&**, обозначающий начало сущности. Спецификация элемента может

содержать регулярные выражения, в том числе вложенные и сложные. В качестве примера ниже приведено правило, описывающее структуру этой главы. Каждая глава (chapter) начинается с введения (intro), после которого следует один или несколько разделов, состоящих из заголовка (heading), одного или нескольких абзацев (para), рисунков (image), таблиц (table) или примечаний (note).

```
<!ELEMENT chapter (intro,(heading,(para|image|table|note)+)+)
```

Но далеко не всегда правила обеспечивают достаточную гибкость. Очень часто элементы содержат текст, и тогда допускаются только два варианта. Во-первых, в составе элемента допускается наличие только текста:

```
<!ELEMENT name (#PCDATA)>
```

И во-вторых, элемент может содержать *любое* сочетание *текста* и *дескрипторов*, располагаемых в произвольном порядке:

```
<!ELEMENT para (#PCDATA|em|strong|code)*>
```

А другие правила типа #PCDATA не допускаются. Например, приведенное ниже выражение неверно. Такие правила следует переписать заново, введя еще один элемент разметки caption, содержащий надпись, или допустив любое сочетание элементов разметки image и текста.

```
<!ELEMENT captionedImage (image,#PCDATA)>
```

Подобное ограничение упрощает работу XML-анализатора при синтаксическом анализе *смешанного содержимого* (текста и дескрипторов). Но в этом случае содержимое становится неконтролируемым, поэтому рекомендуется составлять такие определения DTD, которые содержат только элементы разметки или же ничего, кроме текста.

НА ЗАМЕТКУ! Не совсем верно считать, что в правилах DTD можно указывать произвольные регулярные выражения. XML-анализатор может отклонить сложные наборы правил, которые до определенного момента не дают однозначного результата. Таким, например, является регулярное выражение ((x,y)|(x,z)), потому что, обнаружив **x**, анализатор не сможет сразу выяснить, какой из двух альтернативных вариантов выбрать. Это выражение следует переписать в следующем виде: **(x,(y|z))**. Но некоторые определения нельзя изменить, например ((x,y)*x?). Синтаксический анализатор Sun не выдает никаких предупреждений при обнаружении подобных определений DTD. Он просто выбирает первый совпадающий вариант, что может привести к неверной интерпретации правильных входных данных. На практике определения, подобные приведенным выше, встречаются крайне редко, поскольку большинство определений DTD очень просты.

Для описания допустимых атрибутов элементов разметки используется приведенный ниже синтаксис. В табл. 2.2 приведены допустимые типы атрибутов, а в табл. 2.3 — синтаксис поведения атрибутов по умолчанию.

```
<!ATTLIST элемент атрибут тип поведение_по_умолчанию>
```

Ниже показаны две типичные спецификации атрибутов.

```
<!ATTLIST font style (plain|bold|italic|bold-italic) "plain">
<!ATTLIST size unit CDATA #IMPLIED>
```

В первом примере спецификации для элемента разметки font указан атрибут style, который может иметь четыре допустимых значения. По умолчанию используется значение plain. Во втором примере спецификации для элемента разметки **size** указан атрибут **unit**, который может содержать любую последовательность символов.

Таблица 2.2. Типы атрибутов

Тип	Назначение
CDATA	Произвольная символьная строка
(A_1|A_2| ... |A_n)	Один из строковых атрибутов **A_1**, **A_2**, ... **A_n**
NMTOKEN, NMTOKENS	Одна или более лексема, соответствующая имени
ID	Однозначный идентификатор
IDREF, IDREFS	Одна или более ссылка на однозначный идентификатор
ENTITY, ENTITIES	Одна или более непроанализированная сущность

Таблица 2.3. Поведение атрибутов по умолчанию

Поведение по умолчанию	Назначение
#REQUIRED	Атрибут является обязательным
#IMPLIED	Атрибут не является обязательным
A	Атрибут не является обязательным; анализатор возвращает значение **A**, если атрибут не указан
#FIXED A	Атрибут не должен быть указан или должен быть равен **A**; но в любом случае анализатор возвращает значение **A**

НА ЗАМЕТКУ! Для описания данных рекомендуется применять элементы разметки, а не атрибуты. В соответствии с этой рекомендацией стиль шрифта должен содержаться в отдельном элементе разметки, например, в следующем виде: `<font><style>plain</style>...</font>`. Но атрибуты обладают несомненным преимуществом при использовании перечислений, потому что анализатор может проверять допустимость тех или иных переменных. Так, если стиль шрифта является атрибутом, то анализатор проверяет наличие найденного стиля среди четырех указанных допустимых значений и выбирает значение по умолчанию, если ничего не указано.

Обработка атрибута типа `CDATA` несколько отличается от обработки атрибута типа `#PCDATA` и практически никак не связана с разделами `<![CDATA[...]]>`. Значение атрибута сначала *нормализуется*, т.е. синтаксический анализатор обрабатывает ссылки на символы и сущности (как, например, **é** или **<**) и заменяет разделители пробелами.

Тип `NMTOKEN` аналогичен типу `CDATA`, но в нем не допускается использование большинства символов, отличающихся от букв и цифр, а также разделителей. Кроме того, удаляются все начальные и конечные разделители. Тип `NMTOKENS` представляет собой список лексем имен с разделителями.

Тип атрибута `ID` означает лексему имени, однозначную для данного документа. Однозначность лексемы имени проверяется синтаксическим анализатором. Применение данного типа атрибута демонстрируется в рассматриваемом далее примере программы. Тип `IDREF` означает ссылку на идентификатор, уже существующий в данном документе, наличие которого также проверяется синтаксическим анализатором. Атрибут типа `IDREFS` содержит список ссылок на идентификаторы.

Атрибут типа `ENTITY` указывает на "непроанализированную внешнюю сущность". Этот тип унаследован от SGML и редко применяется на практике. В спецификации языка XML, доступной по адресу `www.xml.com/axml/axml.html`, приводится пример применения подобного типа атрибута.

Определение DTD может также содержать определения *сущностей*, или сокращений, которые заменяются в процессе синтаксического анализа. Характерный пример применения сущностей можно найти в описании пользовательского интерфейса веб-браузера Mozilla/Netscape 6. Данное описание отформатировано в соответствии с требованиями XML и содержит определения сущностей, подобные приведенному ниже.

```
<!ENTITY back.label "Back">
```

Далее в тексте документа могут встретиться ссылки на эти сущности, как показано в приведенном ниже примере.

```
<menuitem label="&back.label;"/>
```

В таком случае синтаксический анализатор заменяет ссылку на сущность замещающей строкой. Для интернационализации прикладной программы потребуется лишь изменить определение самой сущности. Другие примеры применения сущностей более сложны и менее распространены. Дополнительные сведения по данному вопросу можно найти в спецификации языка XML.

На этом краткое введение в определения DTD завершается. Рассмотрим далее способы настройки синтаксического анализатора, чтобы воспользоваться всеми преимуществами DTD. Для этого нужно сначала сообщить фабрике построителей документов о необходимости активизировать проверку следующим образом:

```
factory.setValidating(true);
```

Все построители документов, производимые этой фабрикой, проверяют соответствие входных данных определению DTD. Наиболее полезным результатом такой проверки является игнорирование всех разделителей в элементе разметки. Рассмотрим в качестве примера следующий фрагмент разметки в коде XML:

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

Синтаксический анализатор, не выполняющий проверку, возвращает все разделители между элементами разметки `font`, `name` и `size`. Он поступает так потому, что ему неизвестно, какое из следующих правил описывает дочерние узлы элемента разметки `font`:

```
(name,size)
(#PCDATA,name,size)*
```

или

```
ANY
```

Если же в определении DTD указано правило (**`name`**, **`size`**), то синтаксическому анализатору должно быть известно, что разделитель между этими элементами не является текстом. Построитель документа не будет учитывать разделители, если вызвать следующий метод:

```
factory.setIgnoringElementContentWhitespace(true);
```

Теперь нет никаких сомнений, что узел **`font`** имеет два дочерних узла. Следовательно, нет никаких оснований организовывать приведенный ниже громоздкий цикл.

```
for (int i = 0; i < children.getLength(); i++)
{
  Node child = children.item(i);
  if (child instanceof Element)
  {
    Element childElement = (Element) child;
    if (childElement.getTagName().equals("name")) . . .
    else if (childElement.getTagName().equals("size")) . . .
  }
}
```

Вместо этого для доступа к первому и второму дочерним узлам можно написать следующий фрагмент кода:

```
Element nameElement = (Element) children.item(0);
Element sizeElement = (Element) children.item(1);
```

Именно в этих случаях особенно полезны определения DTD. Как видите, при использовании DTD в программу уже не нужно включать сложные фрагменты кода, выполняющие проверку, потому что синтаксический анализатор проделает эту работу при получении документа.

СОВЕТ. Приступая к обработке XML-документов, многие программисты игнорируют описания DTD и пытаются самостоятельно анализировать древовидную структуру DOM. Чтобы убедиться в неэффективности такого подхода, достаточно только взглянуть на приведенные выше фрагменты кода.

Если анализатор сообщает об ошибке, то прикладная программа должна каким-то образом отреагировать на это, в частности, зарегистрировать ошибку, сообщить о ней пользователю или сгенерировать исключение, чтобы прекратить синтаксический анализ. Поэтому при использовании средств проверки содержимого XML-документа следует также установить обработчик исключений, создав объект, класс которого реализует интерфейс `ErrorHandler`. В этом интерфейсе объявлены следующие три метода:

```
void warning(SAXParseException exception)
void error(SAXParseException exception)
void fatalError(SAXParseException exception)
```

Для установки обработчика исключений служит следующий метод `setErrorHandler()` из класса `DocumentBuilder`:

```
builder.setErrorHandler(handler);
```

javax.xml.parsers.DocumentBuilder 1.4

- `void setEntityResolver(EntityResolver` *resolver*`)`

  Устанавливает определитель сущностей, упоминаемых в анализируемых XML-документах.

- `void setErrorHandler(ErrorHandler` *handler*`)`

  Устанавливает обработчик исключений для выдачи предупреждений и сообщений об ошибках, возникающих при синтаксическом анализе XML-документов.

org.xml.sax.EntityResolver 1.4

- public InputSource resolveEntity(String *publicID*, String *systemID*)

  Возвращает источник вводимых данных, содержащий данные, определяемые заданными идентификаторами, или пустое значение null, указывающее на то, что определителю сущностей неизвестно, как обработать данное конкретное имя. Параметр *publicID* может принимать пустое значение null, если открытые идентификаторы не предоставляются.

org.xml.sax.InputSource 1.4

- InputSource(InputStream *in*)
- InputSource(Reader *in*)
- InputSource(String systemID)

  Создают источник вводимых *данных* на основании указанного потока ввода, объекта типа Reader или системного идентификатора (обычно это относительный или абсолютный URL).

org.xml.sax.ErrorHandler 1.4

- void fatalError(SAXParseException *exception*)
- void error(SAXParseException *exception*)
- void warning(SAXParseException *exception*)

  Эти методы нужно переопределить для создания собственного обработчика неустранимых ошибок, устранимых ошибок или предупреждений.

org.xml.sax.SAXParseException 1.4

- int getLineNumber()
- int getColumnNumber()

  Возвращают номер строки или столбца в конце вводимых данных, при обработке которых возникло исключение.

javax.xml.parsers.DocumentBuilderFactory 1.4

- boolean isValidating()
- void setValidating(boolean *value*)

  Возвращают или устанавливают свойство validating для фабрики. Если это свойство принимает логическое значение true, то созданные фабрикой синтаксические анализаторы будут выполнять проверку входных данных.

- `boolean isIgnoringElementContentWhitespace()`
- `void setIgnoringElementContentWhitespace(boolean` *`value`*`)`

  Получают или устанавливают значение, определяющее, следует ли игнорировать разделители между элементами разметки. Логическое значение `true` указывает на то, что созданные фабрикой синтаксические анализаторы будут игнорировать разделители в том случае, если для элемента разметки не задано смешанное содержимое (например, сочетание элементов разметки с атрибутами типа `#PCDATA`).

Схема XML-документов

Поскольку синтаксис языка XML Schema сложнее, чем у определений DTD, рассмотрим лишь основные характеристики этого языка. Дополнительные сведения о нем можно найти в учебном руководстве, доступном по адресу `http://www.w3.org/TR/xmlschema-0`. Для того чтобы включить в документ ссылку на файл схемы типа XML Schema, нужно указать в корневом элементе соответствующие атрибуты, как в приведенном ниже примере.

```
<?xml version="1.0"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="config.xsd">
  . . .
</configuration>
```

В данном случае в составе корневого элемента содержатся сведения о том, что при проверке документа должен использоваться файл схемы `config.xsd`. Если же в XML-документе применяются пространства имен, то синтаксис немного усложняется. Подробнее об этом можно узнать из учебного руководства по указанному выше адресу. (Префикс `xsi` представляет собой *псевдоним пространства имен*, как поясняется ниже, в разделе "Использование пространств имен".)

Схема определяет тип каждого элемента. Это может быть *простой тип* (символьная строка с ограничениями на форматирование) или же *сложный тип*. Некоторые простые типы встроены в XML Schema. Примеры таких типов приведены ниже.

```
xsd:string
xsd:int
xsd:boolean
```

НА ЗАМЕТКУ! Здесь и далее используется префикс **`xsd:`**, обозначающий пространство имен XML Schema Definition. Некоторые авторы применяют для этой же цели префикс **`xs:`**.

По желанию можно определить собственные простые типы. Ниже приведен пример определения перечислимого типа.

```
<xsd:simpleType name="StyleType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="PLAIN" />
    <xsd:enumeration value="BOLD" />
    <xsd:enumeration value="ITALIC" />
    <xsd:enumeration value="BOLD_ITALIC" />
  </xsd:restriction>
</xsd:simpleType>
```

Определяя элемент разметки, следует указать его тип следующим образом:

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="size" type="xsd:int"/>
<xsd:element name="style" type="StyleType"/>
```

Тип ограничивает возможные варианты содержимого элемента. Например, проверка следующих элементов разметки даст положительный результат:

```
<size>10</size>
<style>PLAIN</style>
```

А приведенные ниже элементы разметки будут отвергнуты синтаксическим анализатором.

```
<size>default</size>
<style>SLANTED</style>
```

Простые типы можно объединять в сложные:

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element ref="name"/>
    <xsd:element ref="size"/>
    <xsd:element ref="style"/>
  </xsd:sequence>
</xsd:complexType>
```

где `FontType` — последовательность элементов разметки `name`, `size` и `style`. В данном определении использован атрибут `ref`, для которого ссылки на определения находятся в схеме. Допускаются также вложенные определения, как в приведенном ниже примере.

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
    <xsd:element name="style" type="StyleType">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="PLAIN" />
          <xsd:enumeration value="BOLD" />
          <xsd:enumeration value="ITALIC" />
          <xsd:enumeration value="BOLD_ITALIC" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Обратите внимание на то, что для элемента `style` использовано *анонимное определение типа*. Языковая конструкция `xsd:sequence` является аналогом операции сцепления определений DTD, а конструкция `xsd:choice` равнозначна операции `|`. Так, приведенная ниже схема разметки заменяет тип `email | phone` в DTD.

```
<xsd:complexType name="contactinfo">
  <xsd:choice>
    <xsd:element ref="email"/>
    <xsd:element ref="phone"/>
  </xsd:choice>
</xsd:complexType>
```

Для повторяющихся элементов можно использовать атрибуты `minoccurs` и `maxoccurs`. Например, аналогом типа `item*` в DTD является следующая схема разметки:

```
<xsd:element name="item" type=". . ." minoccurs="0" maxoccurs="unbounded">
```

Для определения атрибутов в разметку определений `complexType` нужно ввести элементы `xsd:attribute`:

```
<xsd:element name="size">
  <xsd:complexType>
    . . .
    <xsd:attribute name="unit" type="xsd:string" use="optional" default="cm"/>
  </xsd:complexType>
</xsd:element>
```

Приведенной выше схеме разметки равнозначен следующий оператор в определении DTD:

```
<!ATTLIST size unit CDATA #IMPLIED "cm">
```

Определения элемента и типа схемы разметки размещаются в элементе `xsd:schema` следующим образом:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  . . .
</xsd:schema>
```

Синтаксический анализ XML-документа, разметка которого определяется по заданной схеме, выполняется практически так же, как и синтаксический анализ XML-документа, для разметки которого служит определение DTD, за исключением нескольких отличий.

1. Следует непременно активизировать средства поддержки пространств имен в XML-файлах, даже если они и не используются в XML-документах.

   ```
   factory.setNamespaceAware(true);
   ```

2. Необходимо подготовить фабрику для обработки схем разметки. Сделать это позволяют приведенные ниже выражения.

   ```
   final String JAXP_SCHEMA_LANGUAGE =
       "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
   final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
   factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
   ```

3. Анализатор *не* отвергает разделители в элементах разметки. Это серьезное упущение, относительно которого существует разногласие, является ли оно ошибкой. В рассматриваемом далее примере схемы разметки компоновки из листинга 2.4 демонстрируется один из способов разрешения подобного разногласия.

Практический пример применения XML-документов

В этом разделе рассматривается пример применения XML-документа на практике. Как отмечалось в главе 9 первого тома, диспетчер сеточно-контейнерной компоновки типа `GridBagLayout` лучше всего подходит для размещения компонентов библиотеки Swing. Но многие не решаются применять его не столько в силу его сложности, сколько из-за большого объема рутинной работы. При использовании данного диспетчера компоновки вместо создания большого количества повторяющихся

фрагментов кода гораздо удобнее было бы разместить все инструкции в текстовом файле. В этом разделе показано, как пользоваться разметкой в коде XML для описания сеточно-контейнерной компоновки и синтаксического анализа файлов с подобным описанием.

Сеточно-контейнерная компоновка состоит из рядов и столбцов, подобно HTML-таблице. Поэтому ее можно описать в виде последовательности рядов, составленных из отдельных ячеек:

```
<gridbag>
  <row>
    <cell>...</cell>
    <cell>...</cell>
    . . .
  </row>
  <row>
    <cell>...</cell>
    <cell>...</cell>
    . . .
  </row>
  . . .
</gridbag>
```

В файле `gridbag.dtd` указываются следующие правила:

```
<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
```

Некоторые ячейки могут распространяться на несколько рядов и столбцов. В диспетчере сеточно-контейнерной компоновки это достигается путем установки ограничений в виде значений больше **1** в полях `gridwidth` и `gridheight`. А в разметке используются атрибуты с теми же именами следующим образом:

```
<cell gridwidth="2" gridheight="2">
```

Аналогично вводятся атрибуты, определяющие другие ограничения: `fill`, `anchor`, `gridx`, `gridy`, `weightx`, `weighty`, `ipadx` и `ipady`, как показано ниже. (Ограничение `insets` не используется в данном примере, поскольку его нельзя представить с помощью простого типа, хотя организовать поддержку не составит особого труда.)

```
<cell fill="HORIZONTAL" anchor="NORTH">
```

Для большинства атрибутов используются те же значения по умолчанию, что и в конструкторе без аргументов класса `GridBagConstraints`:

```
<!ATTLIST cell gridwidth CDATA "1">
<!ATTLIST cell gridheight CDATA "1">
<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
<!ATTLIST cell anchor (CENTER|NORTH|NORTHEAST|EAST
    |SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
. . .
```

Значения `gridx` и `gridy` задаются и обрабатываются иначе, поскольку указывать их вручную оказывается неудобно, да и чревато ошибками. По этим причинам они указываются как необязательные следующим образом:

```
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
```

Если же эти значения не указаны, то в программе они будут определены следующим образом: в нулевом столбце по умолчанию будет задано нулевое значение

gridx, а во всех остальных столбцах — предыдущее значение gridx плюс предыдущее значение gridwidth. По умолчанию значение gridy всегда совпадает с номером ряда. Поэтому значения gridx и gridy, как правило, указывать не нужно. Но если компонент простирается на несколько столбцов, то значение gridx придется указывать, если требуется пропустить данный компонент.

НА ЗАМЕТКУ! Знатоки сеточно-контейнерной компоновки скажут, что для автоматического определения значений gridx и gridy можно было бы воспользоваться константами RELATIVE и REMAINDER. Мы попытались это сделать, но получить такое же расположение элементов, как и в диалоговом окне выбора шрифта, приведенном на рис. 2.4, нам не удалось. После внимательного изучения исходного кода класса GridBagLayout мы пришли к выводу, что этот класс просто не в состоянии восстанавливать абсолютные позиции.

Во время работы рассматриваемой здесь программы анализируются атрибуты и задаются ограничения для сеточно-контейнерной компоновки. Например, для определения ширины сетки в программе служит следующая строка кода:

```
constraints.gridwidth = Integer.parseInt(e.getAttribute("gridwidth"));
```

Программе не нужно беспокоиться о пропущенных атрибутах, поскольку синтаксический анализатор автоматически предоставит значение по умолчанию, если оно не указано в документе.

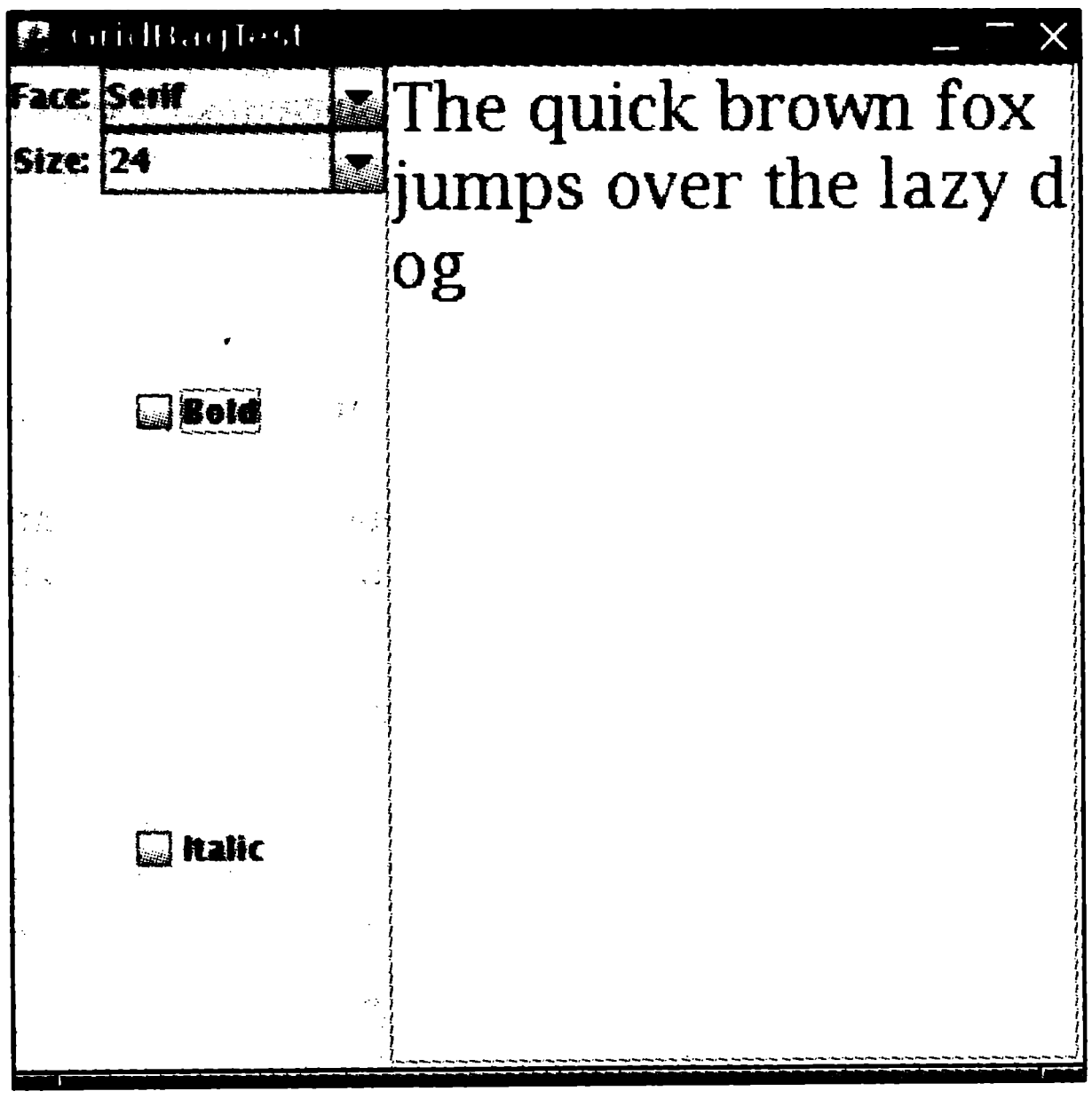


Рис. 2.4. Диалоговое окно выбора шрифта, расположение компонентов в котором определяется в XML-документе

Для того чтобы выяснить, были ли указаны атрибуты gridx и gridy, вызывается метод getAttribute() и проверяется, возвращает ли он пустую символьную строку:

```
String value = e.getAttribute("gridy");
if (value.length() == 0) // использовать по умолчанию
```

```
  constraints.gridy = r;
else
  constraints.gridy = Integer.parseInt(value);
```

Оказывается, что удобнее предоставить пользователю возможность указывать произвольные объекты в ячейках. Это позволяет задавать даже такие типы, как, например, границы, которые не относятся к компонентам. Нужно лишь, чтобы объекты принадлежали классу, отвечающему следующим требованиям к компонентам JavaBeans: он должен иметь конструктор без аргументов и свойства, задаваемые парами методов получения и установки. (Компоненты JavaBeans рассматриваются в главе 8.) Компонент JavaBeans определяется именем класса и свойствами следующим образом:

```
<!ELEMENT bean (class, property*)>
<!ELEMENT class (#PCDATA)>
```

Свойство содержит имя и значение, как показано ниже.

```
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
```

Значение может быть целым, логическим, строковым или другим компонентом JavaBeans:

```
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Ниже в качестве характерного примера приведена разметка компонента `JLabel` со свойством `text`, которое принимает строковое значение `"Face: "`.

```
<bean>
  <class>javax.swing.JLabel</class>
  <property>
    <name>text</name>
    <value><string>Face: </string></value>
  </property>
</bean>
```

На первый взгляд, излишне заключать символьную строку в дескрипторы `<string>`. Почему бы не указывать правило `#PCDATA` для символьных строк, а дескрипторы применять только для других типов? Но в таком случае придется использовать смешанное содержание и менее строгое правило для элемента `value`:

```
<!ELEMENT value (#PCDATA|int|boolean|bean)*>
```

Значение свойства задается в рассматриваемой здесь программе с помощью класса `BeanInfo`, где перечисляются описатели свойств компонента JavaBeans. Сначала осуществляется поиск свойства по заданному имени, а затем вызывается метод его установки с требующимся значением в качестве параметра.

Прочитав описание пользовательского интерфейса, программа получает достаточно сведений для создания компонентов пользовательского интерфейса и их расположения. Но такой интерфейс, безусловно, не будет действовать, поскольку в нем отсутствуют обработчики событий. Для ввода обработчиков событий нужно обнаружить соответствующие компоненты. Поэтому для каждого компонента JavaBeans поддерживается необязательный атрибут `ID`:

```
<!ATTLIST bean id ID #IMPLIED>
```

В качестве примера ниже приведена разметка комбинированного списка с указанием его идентификатора.

```
<bean id="face">
  <class>javax.swing.JComboBox</class>
</bean>
```

Напомним, что синтаксический анализатор проверяет однозначность идентификаторов. А присоединить обработчики событий к компонентам пользовательского интерфейса можно следующим образом:

```
gridbag = new GridBagPane("fontdialog.xml");
setContentPane(gridbag);
JComboBox face = (JComboBox) gridbag.get("face");
face.addListener(listener);
```

НА ЗАМЕТКУ! В рассматриваемом здесь примере разметка в коде XML служит только для описания компоновки компонента, а вводить обработчики событий в код Java можно самостоятельно. Но можно пойти еще дальше, включив код в описание, составленное в формате XML. Для этой цели лучше всего подходит такой язык создания сценариев, как JavaScript, а также его интерпретатор Rhino (`http://www.mozilla.org/rhino`).

В примере программы, исходный код которой приведен в *листинге 2.2*, демонстрируется применение класса `GridBagPane` для выполнения всей трудоемкой и рутинной работы по подготовке сеточно-контейнерной компоновки. Расположение компонентов определяется в XML-файле, содержимое которого приведено в *листинге 2.4*, а внешний вид данной компоновки — на рис. 2.4. В данной программе инициализируются только комбинированные списки, которые слишком сложны, чтобы применять к ним механизм установки свойств компонентов JavaBeans, поддерживаемый в классе `GridBagPane`, а также присоединяются обработчики событий. В классе `GridBagPane`, исходный код которого приведен в *листинге 2.3*, выполняется синтаксический анализ разметки из XML-файла, создаются и располагаются компоненты в рабочем окне программы. А в *листинге 2.5* представлено определение DTD.

В данной программе поддерживается также обработка схемы XML-документа вместо определения DTD. Для этого программу нужно запустить с указанием файла схемы, например, так, как показано ниже. Сама же схема приведена в *листинге 2.6*.

```
java GridBagTest fontdialog-schema.xml
```

В данной программе демонстрируется типичный пример применения формата XML. Этот формат позволяет выражать достаточно сложные взаимосвязи между объектами. А XML-анализатор берет на себя всю рутинную работу по проверке достоверности документа и установке значений по умолчанию.

Листинг 2.2. Исходный код из файла `read/GridBagTest.java`

```
1  package read;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.io.*;
6  import javax.swing.*;
7  /**
8   * В этой программе демонстрируется применение XML-документа
```

```
 * для описания сеточно-контейнерной компоновки
 * @version 1.11 2012-06-03
 * @author Cay Horstmann
*/
public class GridBagTest
{
  public static void main(String[] args)
  {
    EventQueue.invokeLater(new Runnable()
      {
        public void run()
        {
          JFileChooser chooser = new JFileChooser("read");
          chooser.showOpenDialog(null);
          File file = chooser.getSelectedFile();
          JFrame frame = new FontFrame(file);
          frame.setTitle("GridBagTest");
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          frame.setVisible(true);
        }
      });
  }
}

/**
 * Этот фрейм содержит диалоговое окно для выбора шрифтов,
 * описываемое в XML-файле file
 * @param filename Файл, содержащий компоненты пользовательского,
 *                 интерфейса, располагаемые в диалоговом окне
*/
class FontFrame extends JFrame
{
  private GridBagPane gridbag;
  private JComboBox<String> face;
  private JComboBox<String> size;
  private JCheckBox bold;
  private JCheckBox italic;

  @SuppressWarnings("unchecked")
  public FontFrame(File file)
  {
    gridbag = new GridBagPane(file);
    add(gridbag);

    face = (JComboBox<String>) gridbag.get("face");
    size = (JComboBox<String>) gridbag.get("size");
    bold = (JCheckBox) gridbag.get("bold");
    italic = (JCheckBox) gridbag.get("italic");

    face.setModel(new DefaultComboBoxModel<String>(new String[]
      { "Serif","SansSerif", "Monospaced", "Dialog", "DialogInput" }));

    size.setModel(new DefaultComboBoxModel<String>(new String[] { "8",
       "10", "12", "15", "18", "24", "36", "48" }));

    ActionListener listener = new ActionListener()
      {
        public void actionPerformed(ActionEvent event)
```

```
67          {
68            setSample();
69          }
70        };
71      face.addActionListener(listener);
72      size.addActionListener(listener);
73      bold.addActionListener(listener);
74      italic.addActionListener(listener);
75
76      setSample();
77      pack();
78    }
79
80    /**
81     * Этот метод выделяет образец текста выбранным шрифтом
82    */
83    public void setSample()
84    {
85      String fontFace = face.getItemAt(face.getSelectedIndex());
86      int fontSize =
87          Integer.parseInt(size.getItemAt(size.getSelectedIndex()));
88      JTextArea sample = (JTextArea) gridbag.get("sample");
89      int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
90          + (italic.isSelected() ? Font.ITALIC : 0);
91
92      sample.setFont(new Font(fontFace, fontStyle, fontSize));
93      sample.repaint();
94    }
95  }
```

Листинг 2.3. Исходный код из файла `read/GridBagPane.java`

```
1   package read;
2
3   import java.awt.*;
4   import java.beans.*;
5   import java.io.*;
6   import java.lang.reflect.*;
7   import javax.swing.*;
8   import javax.xml.parsers.*;
9   import org.w3c.dom.*;
10
11  /**
12   * Для описания сеточно-контейнерной компоновки отдельных
13   * компонентов этой панели служит XML-файл
14  */
15  public class GridBagPane extends JPanel
16  {
17    private GridBagConstraints constraints;
18
19    /**
20     * Строит панель по сеточно-контейнерной компоновке
21     * @param filename Имя XML-файла, описывающего компоненты
22     *                 панели и их расположение
23    */
24    public GridBagPane(File file)
25    {
```

```
26      setLayout(new GridBagLayout());
27      constraints = new GridBagConstraints();
28
29      try
30      {
31        DocumentBuilderFactory factory =
32           DocumentBuilderFactory.newInstance();
33        factory.setValidating(true);
34
35        if (file.toString().contains("-schema"))
36        {
37          factory.setNamespaceAware(true);
38          final String JAXP_SCHEMA_LANGUAGE =
39            "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
40          final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
41          factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
42        }
43
44          factory.setIgnoringElementContentWhitespace(true);
45
46          DocumentBuilder builder = factory.newDocumentBuilder();
47          Document doc = builder.parse(file);
48          parseGridbag(doc.getDocumentElement());
49        }
50        catch (Exception e)
51        {
52          e.printStackTrace();
53        }
54    }
55
56    /**
57     * Получает компонент по заданному имени
58     * @param name Имя компонента
59     * @return Возвращает компонент с заданным именем или пустое
60     *         значение null, если в данной компоновке панели
61     *         отсутствует компонент с заданным именем
62    */
63    public Component get(String name)
64    {
65      Component[] components = getComponents();
66      for (int i = 0; i < components.length; i++)
67      {
68        if (components[i].getName().equals(name)) return components[i];
69      }
70      return null;
72    }
73    /**
74     * Выполняет синтаксический анализ элемента
75     * сеточно-контейнерной компоновки
76     * @param e Элемент сеточно-контейнерной компоновки
77    */
78    private void parseGridbag(Element e)
79    {
80      NodeList rows = e.getChildNodes();
81      for (int i = 0; i < rows.getLength(); i++)
82      {
83        Element row = (Element) rows.item(i);
84        NodeList cells = row.getChildNodes();
```

```
85          for (int j = 0; j < cells.getLength(); j++)
86          {
87            Element cell = (Element) cells.item(j);
88            parseCell(cell, i, j);
89          }
90        }
91      }
92
93      /**
94       * Выполняет синтаксический анализ элемента ячейки
95       * @param e Элемент ячейки
96       * @param r Ряд ячейки
97       * @param c Столбец ячейки
98      */
99      private void parseCell(Element e, int r, int c)
100     {
101       // получить атрибуты
102
103     String value = e.getAttribute("gridx");
104     if (value.length() == 0) // использовать по умолчанию
105     {
106       if (c == 0) constraints.gridx = 0;
107       else constraints.gridx += constraints.gridwidth;
108     }
109     else constraints.gridx = Integer.parseInt(value);
110
111     value = e.getAttribute("gridy");
112     if (value.length() == 0) // использовать по умолчанию
113     constraints.gridy = r;
114     else constraints.gridy = Integer.parseInt(value);
115
116     constraints.gridwidth =
117         Integer.parseInt(e.getAttribute("gridwidth"));
118     constraints.gridheight =
119         Integer.parseInt(e.getAttribute("gridheight"));
120     constraints.weightx =
121         Integer.parseInt(e.getAttribute("weightx"));
122     constraints.weighty =
123         Integer.parseInt(e.getAttribute("weighty"));
124     constraints.ipadx =
125         Integer.parseInt(e.getAttribute("ipadx"));
126     constraints.ipady =
127         Integer.parseInt(e.getAttribute("ipady"));
128     // использовать рефлексию для получения целых значений
129     // из статических полей
130     Class<GridBagConstraints> cl = GridBagConstraints.class;
131
132     try
133     {
134       String name = e.getAttribute("fill");
135       Field f = cl.getField(name);
136       constraints.fill = f.getInt(cl);
137
138       name = e.getAttribute("anchor");
139       f = cl.getField(name);
140       constraints.anchor = f.getInt(cl);
141     }
142     catch (Exception ex) // методы рефлексии могут генерировать
```

```
143                                 // различные исключения
144       {
145         ex.printStackTrace();
146       }
147
148       Component comp = (Component) parseBean((Element) e.getFirstChild());
149       add(comp, constraints);
150    }
151
152    /**
153     * Выполняет синтаксический анализ элемента JavaBeans
154     * @param e a Элемент JavaBeans
155    */
156    private Object parseBean(Element e)
157    {
158       try
159       {
160          NodeList children = e.getChildNodes();
161          Element classElement = (Element) children.item(0);
162          String className =
163             ((Text) classElement.getFirstChild()).getData();
164
165          Class<?> cl = Class.forName(className);
166
167          Object obj = cl.newInstance();
168
169          if (obj instanceof Component)
170             ((Component) obj).setName(e.getAttribute("id"));
171
172          for (int i = 1; i < children.getLength(); i++)
173          {
174             Node propertyElement = children.item(i);
175             Element nameElement =
177                (Element) propertyElement.getFirstChild();
176             String propertyName =
178                ((Text) nameElement.getFirstChild()).getData();
179
180             Element valueElement =
181                (Element) propertyElement.getLastChild();
182             Object value = parseValue(valueElement);
183             BeanInfo beanInfo = Introspector.getBeanInfo(cl);
184             PropertyDescriptor[] descriptors =
185                beanInfo.getPropertyDescriptors();
186             boolean done = false;
187             for (int j = 0; !done && j < descriptors.length; j++)
188             {
189                if (descriptors[j].getName().equals(propertyName))
190                {
191                   descriptors[j].getWriteMethod().invoke(obj, value);
192                   done = true;
193                }
194             }
195          }
196          return obj;
197       }
198       catch (Exception ex) // методы рефлексии могут генерировать
199                            // различные исключения
200       {
```

```
201       ex.printStackTrace();
202       return null;
203     }
204   }
205
206   /**
207    * Выполняет синтаксический анализ элемента значения
208    * @param e Элемент значения
209   */
210   private Object parseValue(Element e)
211   {
212     Element child = (Element) e.getFirstChild();
213     if (child.getTagName().equals("bean")) return parseBean(child);
214     String text = ((Text) child.getFirstChild()).getData();
215     if (child.getTagName().equals("int")) return new Integer(text);
216     else if (child.getTagName().equals("boolean"))
217         return new Boolean(text);
218     else if (child.getTagName().equals("string")) return text;
219     else return null;
220   }
221 }
```

Листинг 2.4. Исходный код из файла `read/fontdialog.xml`

```
1 <?xml version="1.0"?>
2 <!DOCTYPE gridbag SYSTEM "gridbag.dtd">
3 <gridbag>
4   <row>
5     <cell anchor="EAST">
6       <bean>
7         <class>javax.swing.JLabel</class>
8         <property>
9           <name>text</name>
10          <value><string>Face: </string></value>
11        </property>
12      </bean>
13    </cell>
14    <cell fill="HORIZONTAL" weightx="100">
15      <bean id="face">
16        <class>javax.swing.JComboBox</class>
17      </bean>
18    </cell>
19    <cell gridheight="4" fill="BOTH" weightx="100" weighty="100">
20      <bean id="sample">
21        <class>javax.swing.JTextArea</class>
22        <property>
23          <name>text</name>
24          <value><string>The quick brown fox jumps
25                 over the lazy dog</string></value>
26        </property>
27        <property>
28          <name>editable</name>
29          <value><boolean>false</boolean></value>
30        </property>
31        <property>
32          <name>lineWrap</name>
33          <value><boolean>true</boolean></value>
34        </property>
```

```
35          <property>
36            <name>border</name>
37            <value>
38              <bean>
39                <class>javax.swing.border.EtchedBorder</class>
40              </bean>
41            </value>
42          </property>
43        </bean>
44      </cell>
45    </row>
46    <row>
47      <cell anchor="EAST">
48        <bean>
49          <class>javax.swing.JLabel</class>
50          <property>
51            <name>text</name>
52             <value><string>Size: </string></value>
53           </property>
54         </bean>
55       </cell>
56       <cell fill="HORIZONTAL" weightx="100">
57         <bean id="size">
58           <class>javax.swing.JComboBox</class>
59         </bean>
60      </cell>
61    </row>
62    <row>
63     <cell gridwidth="2" weighty="100">
64       <bean id="bold">
65         <class>javax.swing.JCheckBox</class>
66         <property>
67           <name>text</name>
68           <value><string>Bold</string></value>
69         </property>
70        </bean>
71      </cell>
72    </row>
74    <row>
75      <cell gridwidth="2" weighty="100">
76        <bean id="italic">
77          <class>javax.swing.JCheckBox</class>
78          <property>
79            <name>text</name>
80            <value><string>Italic</string></value>
81          </property>
82         </bean>
83       </cell>
84     </row>
85 </gridbag>
```

Листинг 2.5. Исходный код из файла `read/gridbag.dtd`

```
1 <!ELEMENT gridbag (row)*>
2 <!ELEMENT row (cell)*>
3 <!ELEMENT cell (bean)>
4
5 <!ATTLIST cell gridx CDATA #IMPLIED>
```

```
6 <!ATTLIST cell gridy CDATA #IMPLIED>
7 <!ATTLIST cell gridwidth CDATA "1">
8 <!ATTLIST cell gridheight CDATA "1">
9 <!ATTLIST cell weightx CDATA "0">
10 <!ATTLIST cell weighty CDATA "0">
11 <!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
12 <!ATTLIST cell anchor (CENTER|NORTH|NORTHEAST|EAST|
13     SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
14 <!ATTLIST cell ipadx CDATA "0">
15 <!ATTLIST cell ipady CDATA "0">
16 <!ELEMENT bean (class, property*)>
17 <!ATTLIST bean id ID #IMPLIED>
18
19 <!ELEMENT class (#PCDATA)>
20 <!ELEMENT property (name, value)>
21 <!ELEMENT name (#PCDATA)>
22 <!ELEMENT value (int|string|boolean|bean)>
23 <!ELEMENT int (#PCDATA)>
24 <!ELEMENT string (#PCDATA)>
25 <!ELEMENT boolean (#PCDATA)>
```

Листинг 2.6. Исходный код из файла `read/gridbag.xsd`

```
1 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2
3   <xsd:element name="gridbag" type="GridBagType"/>
4
5   <xsd:element name="bean" type="BeanType"/>
6
7   <xsd:complexType name="GridBagType">
8     <xsd:sequence>
9       <xsd:element name= "row"
10           type="RowType" minOccurs="0" maxOccurs="unbounded"/>
11    </xsd:sequence>
12  </xsd:complexType>
13
14  <xsd:complexType name="RowType">
15    <xsd:sequence>
16     <xsd:element name= "cell"
17          type="CellType" minOccurs="0" maxOccurs="unbounded"/>
18    </xsd:sequence>
19  </xsd:complexType>
20
21  <xsd:complexType name="CellType">
22    <xsd:sequence>
23      <xsd:element ref="bean"/>
24    </xsd:sequence>
25    <xsd:attribute name="gridx" type="xsd:int" use="optional"/>
26    <xsd:attribute name="gridy" type="xsd:int" use="optional"/>
27    <xsd:attribute name="gridwidth"
28         type="xsd:int" use="optional" default="1" />
29    <xsd:attribute name="gridheight"
30         type="xsd:int" use="optional" default="1" />
31    <xsd:attribute name="weightx"
32         type="xsd:int" use="optional" default="0" />
33    <xsd:attribute name="weighty"
34         type="xsd:int" use="optional" default="0" />
35    <xsd:attribute name="fill" use="optional" default="NONE">
```

```
36          <xsd:simpleType>
37            <xsd:restriction base="xsd:string">
38              <xsd:enumeration value="NONE" />
39              <xsd:enumeration value="BOTH" />
40              <xsd:enumeration value="HORIZONTAL" />
41              <xsd:enumeration value="VERTICAL" />
42            </xsd:restriction>
43          </xsd:simpleType>
44        </xsd:attribute>
45        <xsd:attribute name="anchor" use="optional" default="CENTER">
46          <xsd:simpleType>
47            <xsd:restriction base="xsd:string">
48              <xsd:enumeration value="CENTER" />
49              <xsd:enumeration value="NORTH" />
50              <xsd:enumeration value="NORTHEAST" />
51              <xsd:enumeration value="EAST" />
52              <xsd:enumeration value="SOUTHEAST" />
53              <xsd:enumeration value="SOUTH" />
54              <xsd:enumeration value="SOUTHWEST" />
55              <xsd:enumeration value="WEST" />
56              <xsd:enumeration value="NORTHWEST" />
57            </xsd:restriction>
58          </xsd:simpleType>
59        </xsd:attribute>
60        <xsd:attribute name="ipady"
61             type="xsd:int" use="optional" default="0" />
62        <xsd:attribute name="ipadx"
63             type="xsd:int" use="optional" default="0" />
64      </xsd:complexType>
65
66      <xsd:complexType name="BeanType">
67        <xsd:sequence>
68          <xsd:element name="class" type="xsd:string"/>
69          <xsd:element name="property"
70               type="PropertyType" minOccurs="0" maxOccurs="unbounded"/>
71        </xsd:sequence>
72        <xsd:attribute name="id" type="xsd:ID" use="optional" />
73      </xsd:complexType>
74
75      <xsd:complexType name="PropertyType">
76        <xsd:sequence>
77          <xsd:element name="name" type="xsd:string"/>
78          <xsd:element name="value" type="ValueType"/>
79        </xsd:sequence>
80      </xsd:complexType>
81
82      <xsd:complexType name="ValueType">
83        <xsd:choice>
84          <xsd:element ref="bean"/>
85          <xsd:element name="int" type="xsd:int"/>
86          <xsd:element name="string" type="xsd:string"/>
87          <xsd:element name="boolean" type="xsd:boolean"/>
88        </xsd:choice>
89      </xsd:complexType>
90  </xsd:schema>
```

Поиск информации средствами XPath

Если требуется найти информацию в XML-документе, то с этой целью придется непременно организовать обход древовидной структуры DOM. Язык XPath упрощает доступ к узлам дерева. Допустим, имеется следующий XML-документ:

```
<configuration>
  . . .
  <database>
    <username>dbuser</username>
    <password>secret</password>
      . . .
  </database>
</configuration>
```

Для того чтобы получить из него имя пользователя базы данных, достаточно вычислить следующее выражение XPath:

```
/configuration/database/username
```

Сделать это много проще, чем организовывать непосредственный обход древовидной структуры DOM для выполнения следующих действий.

1. Получить узел документа.
2. Перечислить его дочерние узлы.
3. Обнаружить элемент разметки `database`.
4. Получить его первый дочерний элемент `username`.
5. Получить его первый дочерний узел `text`.
6. Получить из него данные.

Язык XPath позволяет описывать *ряд узлов* в XML-документе. Например, в следующем выражении описывается ряд элементов `row`, которые являются дочерними для корневого элемента разметки `gridbag`:

```
/gridbag/row
```

Для выбора конкретного элемента служит операция `[]`. Так, в следующем выражении определяется первый дочерний элемент разметки (отчет индексов начинается с единицы):

```
/gridbag/row[1]
```

Для получения значений атрибутов служит операция `@`. Например, в следующем выражении XPath описывается атрибут `anchor` первой ячейки в первом ряду:

```
/gridbag/row[1]/cell[1]/@anchor
```

И наконец, в приведенном ниже выражении XPath описываются все узлы атрибутов `anchor` для элементов разметки `cell`, которые располагаются в элементах разметки `row`, а те, в свою очередь, являются дочерними для корневого элемента разметки `gridbag`.

```
/gridbag/row/cell/@anchor
```

В XPath имеется ряд функций, упрощающих работу с документом. Например, в следующем выражении определяется количество элементов `row`, дочерних для корневого элемента `gridbag`:

```
count(/gridbag/row)
```

Примеры выражений XPath, в том числе и довольно сложных, можно найти в спецификации этого языка по адресу `http://www.w3c.org/TR/xpath`. Имеется также очень удачно составленное руководство по XPath, доступное по адресу http://www.zvon.org/xxl/XPathTutorial/General_rus/examples.html.

В версии Java SE 5.0 внедрен прикладной интерфейс API для обработки выражений XPath. С этой целью нужно сначала создать объект типа `XPath`, используя класс `XPathFactory`:

```
XPathFactory xpfactory = XPathFactory.newInstance();
path = xpfactory.newXPath();
```

После этого вызывается приведенный ниже метод `evaluate()`, обрабатывающий выражения XPath. С помощью одного объекта типа `XPath` можно обработать несколько выражений.

```
String username = path.evaluate("/configuration/database/username", doc);
```

В данной форме метод `evaluate()` возвращает результат в виде символьной строки. Это удобно для получения текста, например, из узла `username`, как показано выше. Если из выражения XPath получается ряд узлов, то для их обработки можно сделать следующий вызов:

```
NodeList nodes =
    (NodeList) path.evaluate("/gridbag/row", doc, XPathConstants.NODESET);
```

Если же в итоге получается один узел, то в качестве третьего параметра метода `evaluate()` следует указать значение константы `XPathConstants.NODE`, как показано ниже.

```
Node node = (Node) path.evaluate("/gridbag/row[1]", doc, XPathConstants.NODE);
```

А если в итоге получается количество узлов, то в качестве третьего параметра метода `evaluate()` следует указать значение константы `XPathConstants.NUMBER` следующим образом:

```
int count = ((Number) path.evaluate("count(/gridbag/row)",
    doc, XPathConstants.NUMBER)).intValue();
```

Совсем не обязательно начинать поиск с корневого узла документа. В качестве исходной точки можно выбрать любой узел и даже перечень узлов. Например, получив узел в результате вычисления приведенного выше выражения, можно сделать следующий вызов:

```
result = path.evaluate(expression, node);
```

В примере программы, исходный код которой приведен в листинге 2.7, демонстрируется порядок вычисления выражений XPath. Загрузите сначала XML-файл и введите выражение. Затем выберите тип выражения и щелкните на кнопке Evaluate (Вычислить). Результат вычисления введенного выражения появится в нижней части окна (рис. 2.5).

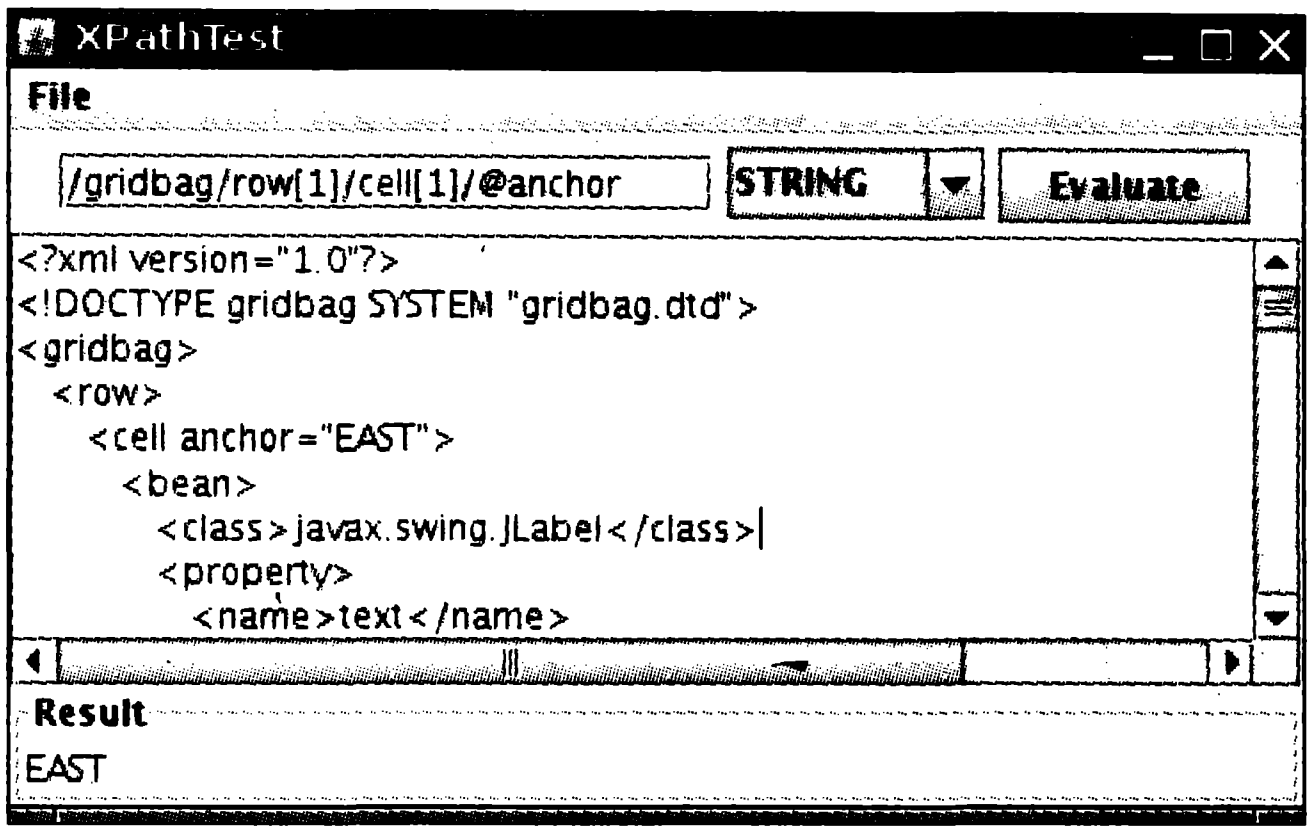


Рис. 2.5. Вычисление выражений XPath

Листинг 2.7. Исходный код из файла xpath/XPathTester.java

```
1   package xpath;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.io.*;
6   import java.nio.file.*;
7   import javax.swing.*;
8   import javax.swing.border.*;
9   import javax.xml.namespace.*;
10  import javax.xml.parsers.*;
11  import javax.xml.xpath.*;
12  import org.w3c.dom.*;
13  import org.xml.sax.*;
14
15  /**
16   * В этой программе вычисляются выражения XPath
17   * @version 1.01 2007-06-25
18   * @author Cay Horstmann
19  */
20  public class XPathTester
21  {
22    public static void main(String[] args)
23    {
24      EventQueue.invokeLater(new Runnable()
25        {
26          public void run()
27          {
28            JFrame frame = new XPathFrame();
29            frame.setTitle("XPathTest");
30            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31            frame.setVisible(true);
32          }
33        });
34    }
35  }
```

```
36
37  /**
38   * В этом фрейме отображается XML-документ, панель для ввода
39   * выражений XPath, а также текстовое поле для вывода результата
40  */
41  class XPathFrame extends JFrame
42  {
43    private DocumentBuilder builder;
44    private Document doc;
45    private XPath path;
46    private JTextField expression;
47    private JTextField result;
48    private JTextArea docText;
49    private JComboBox<String> typeCombo;
50
51    public XPathFrame()
52    {
53      JMenu fileMenu = new JMenu("File");
54      JMenuItem openItem = new JMenuItem("Open");
55      openItem.addActionListener(new ActionListener()
56        {
57          public void actionPerformed(ActionEvent event)
58          {
59            openFile();
60          }
61        });
62      fileMenu.add(openItem);
63
64      JMenuItem exitItem = new JMenuItem("Exit");
65      exitItem.addActionListener(new ActionListener()
66        {
67          public void actionPerformed(ActionEvent event)
68          {
69            System.exit(0);
70          }
71        });
72      fileMenu.add(exitItem);
73
74      JMenuBar menuBar = new JMenuBar();
75      menuBar.add(fileMenu);
76      setJMenuBar(menuBar);
77      ActionListener listener = new ActionListener()
78        {
79          public void actionPerformed(ActionEvent event)
80          {
81            evaluate();
82          }
83        };
84      expression = new JTextField(20);
85      expression.addActionListener(listener);
86      JButton evaluateButton = new JButton("Evaluate");
87      evaluateButton.addActionListener(listener);
88
89      typeCombo = new JComboBox<String>(new String[] {
90        "STRING", "NODE", "NODESET", "NUMBER", "BOOLEAN" });
91      typeCombo.setSelectedItem("STRING");
92
```

```
93        JPanel panel = new JPanel();
94        panel.add(expression);
95        panel.add(typeCombo);
96        panel.add(evaluateButton);
97        docText = new JTextArea(10, 40);
98        result = new JTextField();
99        result.setBorder(new TitledBorder("Result"));
100
101       add(panel, BorderLayout.NORTH);
102       add(new JScrollPane(docText), BorderLayout.CENTER);
103       add(result, BorderLayout.SOUTH);
104
105       try
106       {
107         DocumentBuilderFactory factory =
108             DocumentBuilderFactory.newInstance();
109         builder = factory.newDocumentBuilder();
110       }
111       catch (ParserConfigurationException e)
112       {
113         JOptionPane.showMessageDialog(this, e);
114       }
115
116       XPathFactory xpfactory = XPathFactory.newInstance();
117       path = xpfactory.newXPath();
118       pack();
119     }
120
121     /**
122       * Открыть файл и загрузить документ
123     */
124     public void openFile()
125     {
126       JFileChooser chooser = new JFileChooser();
127       chooser.setCurrentDirectory(new File("xpath"));
128
129       chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
130         {
131           public boolean accept(File f)
132           {
133             return f.isDirectory() ||
134                    f.getName().toLowerCase().endsWith(".xml");
135           }
136
137           public String getDescription()
138           {
139             return "XML files";
140           }
141         });
142       int r = chooser.showOpenDialog(this);
143       if (r != JFileChooser.APPROVE_OPTION) return;
144       File file = chooser.getSelectedFile();
145       try
146       {
147         docText.setText(new String(Files.readAllBytes(file.toPath())));
148         doc = builder.parse(file);
149       }
```

```
150     catch (IOException e)
151     {
152       JOptionPane.showMessageDialog(this, e);
153     }
154     catch (SAXException e)
155     {
156       JOptionPane.showMessageDialog(this, e);
157     }
158   }
159
160   public void evaluate()
161   {
162     try
163     {
164       String typeName = (String) typeCombo.getSelectedItem();
165       QName returnType =
166           (QName) XPathConstants.class.getField(typeName).get(null);
167       Object evalResult =
168           path.evaluate(expression.getText(), doc, returnType);
169       if (typeName.equals("NODESET"))
170       {
171         NodeList list = (NodeList) evalResult;
172         StringBuilder builder = new StringBuilder();
173         builder.append("{");
174         for (int i = 0; i < list.getLength(); i++)
175         {
176           if (i > 0) builder.append(", ");
177           builder.append("" + list.item(i));
178         }
179         builder.append("}");
180         result.setText("" + builder);
181       }
182       else result.setText("" + evalResult);
183     }
184     catch (XPathExpressionException e)
185     {
186       result.setText("" + e);
187     }
188     catch (Exception e) // исключение при рефлексии
189     {
190       e.printStackTrace();
191     }
192   }
193 }
```

javax.xml.xpath.XPathFactory 5.0

- `static XPathFactory newInstance()`

  Возвращает экземпляр класса `XPathFactory`, используемый для создания объектов типа `XPath`.

- `XPath newXpath()`

  Создает объект типа `XPath`, который можно использовать для обработки выражений XPath.

javax.xml.xpath.XPath 5.0

- `String evaluate(String` *`expression`*`, Object` *`startingPoint`*`)`

  Вычисляет выражение, начиная поиск с заданной исходной точки. В качестве исходной точки может быть указан узел или перечень узлов. Если в результате вычисления данного выражения получается узел или ряд узлов, то возвращаемая символьная строка содержит данные из всех дочерних текстовых узлов.

- `Object evaluate(String` *`expression`*`, Object` *`startingPoint`*`,`
  `QName` *`resultType`*`)`

  Вычисляет выражение, начиная поиск с заданной исходной точки. В качестве исходной точки может быть указан узел или перечень узлов. В качестве параметра *`resultType`* задается одна из следующих констант, определяемых в классе `XPathConstants`: `STRING`, `NODE`, `NODESET`, `NUMBER` или `BOOLEAN`. Возвращаемое значение относится к типу `String`, `Node`, `NodeList`, `Number` или `Boolean`.

Использование пространств имен

Во избежание конфликтов при использовании одинаковых имен в языке Java предусмотрены пакеты. Разные классы могут иметь одинаковые имена, если они находятся в разных пакетах. А в XML для различения одинаково именуемых элементов и атрибутов используется механизм *пространств имен*. Пространство имен обозначается с помощью универсального идентификатора ресурсов (URI), как показано в приведенном ниже примере.

```
http://www.w3.org/2001/XMLSchema
uuid:1c759aed-b748-475c-ab68-10679700c4f2
urn:com:books-r-us
```

Чаще всего для этой цели используется формат URL по протоколу HTTP. Следует, однако, иметь в виду, что URL в данном случае выполняет лишь роль идентификатора. Приведенные ниже URL обозначают *разные* пространства имен, хотя веб-сервер интерпретировал бы их как указатели на один и тот же документ.

```
http://www.horstmann.com/corejava
http://www.horstmann.com/corejava/index.html
```

Более того, URL, определяющий пространство имен, может не указывать на конкретный документ. XML-анализатор и не пытается найти что-нибудь по этому адресу. Тем не менее по URL, обозначающему пространство имен, принято располагать документ с описанием назначения этого пространства. Например, по адресу `http://www.w3c.org/2001/XMLSchema` находится документ с описанием схем XML-документов.

Почему для обозначения пространства имен применяются URL? Очевидно, что в таком случае легче гарантировать их однозначность. В самом деле, для подлинного URL однозначность имени узла сети гарантируется структурой системы доменных имен, а однозначность остальной части URL должна обеспечить надлежащая организация файловой системы. Именно из этих соображений для многих пакетов выбраны доменные имена с обратным порядком следования доменов.

Конечно, для длинных имен проще обеспечить однозначность, но использовать их в программе не совсем удобно. В Java для этой цели предусмотрен механизм импорта

пакетов, в результате чего в исходном тексте программы присутствуют в основном имена классов. Аналогичный механизм предусмотрен и в XML, как показано ниже. В итоге элемент и все его дочерние узлы становятся частью заданного пространства имен.

```
<элемент xmlns="URI_пространства_имен">
  дочерные_узлы
</элемент>
```

При необходимости дочерний узел может обеспечить себе отдельное пространство имен, как показано в приведенном ниже примере. В данном случае первый дочерний узел и дочерние узлы предыдущего уровня принадлежат второму пространству имен.

```
<элемент xmlns="URI_пространства_имен_1">
  <дочерний_узел xmlns="URI_пространства_имен_2">
    дочерние_узлы_предыдущего_уровня
  </дочерний_узел>
    другие дочерние_узлы
</элемент>
```

Этот простой механизм подходит только в том случае, если требуется одно пространство имен или же если пространства имен вложены друг в друга естественным образом. В иных случаях более предпочтительным оказывается альтернативный механизм, аналог которого отсутствует в Java. Для пространства имен можно использовать *префикс*, т.е. короткий идентификатор, выбираемый для конкретного документа. Ниже приведен типичный пример применения префикса **xsd** в файле схемы типа XML Schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="gridbag" type="GridBagType"/>
  . . .
</xsd:schema>
```

Атрибут `xmlns:`*`префикс`*`="`*`URI_пространства_имен`*`"` определяет пространство имен и префикс. В данном примере префиксом является символьная строка `"xsd"`. Таким образом, атрибут `xsd:schema` фактически означает следующее: указанная схема (`schema`) находится в пространстве имен `http://www.w3.org/2001/XMLSchema`.

НА ЗАМЕТКУ! Пространство имен родительского элемента наследуется только дочерними элементами. Атрибуты без явного указания префикса не считаются частью пространства имен. Рассмотрим следующий вымышленный пример:

```
<configuration xmlns="http://www.horstmann.com/corejava"
  xmlns:si="http://www.bipm.fr/enus/3_SI/si.html">
  <size value="210" si:unit="mm"/>
  . . .
</configuration>
```

В данном примере элементы разметки `configuration` и `size` являются частью пространства имен, определяемого по следующему URI: `http://www.horstmann.com/corejava`. Атрибут `si:unit` является частью пространства имен по следующему URI: `http://www.bipm.fr/enus/3_SI/si.html`. Но атрибут `value` не принадлежит ни одному из этих пространств имен.

Обращение синтаксического анализатора с пространствами имен поддается контролю. По умолчанию в DOM-анализаторе от компании Sun пространства имен во внимание не принимаются. Для того чтобы включить режим управления пространствами имен, нужно вызвать метод `setNamespaceAware()` из класса `DocumentBuilderFactory` следующим образом:

```
factory.setNamespaceAware(true);
```

После этого все созданные данной фабрикой конструкторы будут поддерживать пространства имен. У каждого узла имеются следующие три свойства.

- *Уточненное имя* с префиксом, возвращаемое методами `getNodeName()`, `getTagName()` и т.д.
- URI пространства имен, возвращаемый методом `getNamespaceURI()`.
- *Локальное имя* без префикса, возвращаемое методом `getLocalName()`.

Обратимся к конкретному примеру. Допустим, синтаксический анализатор обнаруживает следующий элемент разметки:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

В таком случае он сообщает о наличии следующих свойств узла:

- Уточненное имя: `xsd:schema`.
- URI пространства имен: `http://www.w3.org/2001/XMLSchema`.
- Локальное имя: `schema`.

НА ЗАМЕТКУ! Если режим управления пространствами имен отключен, методы `getNamespaceURI()` и `getLocalName()` возвращают пустое значение `null`.

org.w3c.dom.Node 1.4

- `String getLocalName()`

  Возвращает локальное имя (без префикса) или пустое значение `null`, если режим управления пространствами имен отключен.

- `String getNamespaceURI()`

  Возвращает URI пространства имен или пустое значение `null`, если узел не является частью пространства имен или режим управления пространствами имен отключен.

javax.xml.parsers.DocumentBuilderFactory 1.4

- `boolean isNamespaceAware()`
- `void setNamespaceAware(boolean value)`

  Получают или устанавливают значение свойства, определяющего режим управления пространствами имен. Если установлено логическое значение `true` параметра *`value`*, то в генерируемых фабрикой синтаксических анализаторах будет включен режим управления пространствами имен.

Потоковые синтаксические анализаторы

DOM-анализатор считывает XML-документ и представляет его в виде древовидной структуры данных. Для большинства приложений оказывается достаточно и модели DOM. Но она совершенно неэффективна для обработки крупного документа, когда алгоритм обработки слишком прост, чтобы оперативно анализировать узлы, не

просматривая все дерево в целом. В подобных случаях следует применять потоковые синтаксические анализаторы.

В последующих разделах будут рассмотрены потоковые анализаторы, доступные в библиотеке Java: почтенный SAX-анализатор и более современный StAX-анализатор, который появился в версии Java SE 6. SAX-анализатор использует обратные вызовы событий, а StAX-анализатор предоставляет итератор событий синтаксического анализа. Последний оказывается более удобным в употреблении.

Применение SAX-анализатора

SAX-анализатор уведомляет о событиях, наступающих в ходе синтаксического анализа компонентов данных, вводимых из XML-документа. Но сам документ не хранится в памяти, а создание структуры из вводимых данных возлагается на обработчики событий. На самом деле в основу работы DOM-анализатора положен тот же самый принцип, что и для SAX-анализатора: дерево DOM строится по мере приема событий, наступающих при синтаксическом анализе.

Для того чтобы воспользоваться SAX-анализатором, нужно создать обработчик событий, определяющий действия для обработки различных событий, наступающих при синтаксическом анализе. В интерфейсе `ContentHandler` определен ряд перечисленных ниже методов обратного вызова, к которым обращается анализатор в ходе синтаксического анализа XML-документа.

- Методы `startElement()` и `endElement()` вызываются всякий раз, когда получается открывающий и закрывающий дескриптор.
- Метод `characters()` вызывается при получении символьных данных.
- Методы `startDocument()` и `endDocument()` вызываются в начале и в конце документа.

Например, при синтаксическом анализе следующего фрагмента разметки:

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

SAX-анализатор генерирует обратные вызовы целого ряда перечисленных ниже методов.

1. Метод `startElement()`: для элемента разметки `font`.
2. Метод `startElement()`: для элемента разметки `name`.
3. Метод `characters()`: для текстового содержимого `Helvetica`.
4. Метод `endElement()`: для элемента разметки `name`.
5. Метод `startElement()`: для элемента разметки `size` с атрибутом `units="pt"`.
6. Метод `characters()`: для числового содержимого **36**.
7. Метод `endElement()`: для элемента разметки `size`.
8. Метод `endElement()`: для элемента разметки `font`.

Для выполнения требуемых действий при синтаксическом анализе содержимого вводимого файла необходимо переопределить эти методы. В примере программы, исходный код которой приведен в листинге 2.8, выводятся сведения обо всех гипертекстовых ссылках типа `<a href="...">`, найденных в HTML-файле. В ней

переопределен метод `startElement()` обработчика событий и реализована проверка всех гипертекстовых ссылок с именем `a` и атрибутом `href`. Подобный код применяется в поисковых роботах, которые автоматически выявляют новые веб-страницы для индексации, переходя по ссылкам.

НА ЗАМЕТКУ! К сожалению, многие HTML-страницы совершенно не соответствуют формату XML, и поэтому рассматриваемая здесь программа не способна произвести их синтаксический анализ. Тем не менее большинство веб-страниц, созданных консорциумом W3C, составлены на XHTML (диалекте HTML, соответствующем формату XML). Поэтому этими веб-страницами можно воспользоваться для тестирования данной программы. Например, для составления списка всех гипертекстовых ссылок по соответствующим URL на веб-странице `http://www.w3c.org/MarkUp` нужно ввести следующую команду:

```
java SAXTest http://www.w3c.org/MarkUp
```

Рассматриваемая здесь программа служит характерным примером употребления SAX-анализатора. В ней полностью игнорируется контекст, в котором находится элемент разметки `a`, а также не сохраняется древовидная структура документа. Для получения SAX-анализатора используется следующий фрагмент кода:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

После этого документ можно обработать следующим образом:

```
parser.parse(source, handler)
```

где `source` — источник вводимых данных, который может быть файлом, символьной строкой с URL или потоком ввода, а `handler` — подкласс, производный от класса `DefaultHandler`. В классе `DefaultHandler` определены методы, объявленные в следующих четырех интерфейсах:

```
ContentHandler
DTDHandler
EntityResolver
ErrorHandler
```

Эти методы не выполняют никаких действий. В рассматриваемой здесь программе определен обработчик событий, в котором для поиска элементов с именем `a` и атрибутом `href` переопределен метод `startElement()` из интерфейса `ContentHahdler`, как показано ниже.

```
DefaultHandler handler = new DefaultHandler()
  {
    public void startElement(String namespaceURI, String lname,
        String qname, Attributes attrs)
    {
      if (lname.equalsIgnoreCase("a") && attrs != null)
      {
        for (int i = 0; i < attrs.getLength(); i++)
        {
          String aname = attrs.getLocalName(i);
          if (aname.equalsIgnoreCase("href"))
            System.out.println(attrs.getValue(i));
        }
      }
    }
};
```

Методу `startElement()` передаются три параметра, описывающие имя элемента. В частности, параметр `qname` сообщает уточненное имя в форме *`префикс:локальное_имя`*. Если включен режим управления пространствами имен, то параметры `namespaceURI` и `lname` описывают пространство имен и локальное (неуточненное) имя.

Как и при употреблении DOM-анализатора, режим управления пространствами имен исходно отключен. Для его включения нужно вызвать метод `setNamespaceAware()` из фабричного класса `SAXParserFactory` следующим образом:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();
```

В рассматриваемой здесь программе преодолевается еще одно распространенное препятствие. В начале XHTML-файла обычно находится дескриптор, содержащий ссылку на описание DTD, которое требуется загрузить синтаксическому анализатору. Очевидно, что консорциуму W3C явно не улыбалась перспектива обслуживать миллиарды копий файлов вроде `www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd`. Поэтому они вообще отказались от такого обслуживания, но на момент написания этой книги описание DTD все же обслуживалось, хотя и очень медленно. Если же вам не требуется проверка достоверности документа, просто сделайте следующий вызов:

```
factory.setFeature(
  "http://apache.org/xml/features/nonvalidating/load-external-dtd", false);
```

Итак, в листинге 2.8 приведен исходный код простейшего поискового робота. Далее в этой главе рассматривается еще один интересный пример применения SAX-анализатора. Простой способ превратить источник данных, несовместимый с форматом XML, в XML-документ — уведомить о SAX-событиях, о которых будет затем сообщать XML-анализатор. Более подробно эти вопросы рассматриваются далее, в разделе, посвященном XSLT — языку преобразований XML-документов.

Листинг 2.8. Исходный код из файла `sax/SAXTest.java`

```
1  package sax;
2
3  import java.io.*;
4  import java.net.*;
5  import javax.xml.parsers.*;
6  import org.xml.sax.*;
7  import org.xml.sax.helpers.*;
8
9  /**
10  * В этой программе демонстрируется применение SAX-анализатора.
11  * Программа выводит все гиперссылки из веб-страницы формата XHTML
12  * Использование: java SAXTest url
13  * @version 1.00 2001-09-29
14  * @author Cay Horstmann
15 */
16 public class SAXTest
17 {
18   public static void main(String[] args) throws Exception
19   {
20     String url;
21     if (args.length == 0)
22     {
23       url = "http://www.w3c.org";
```

```
24        System.out.println("Using " + url);
25      }
26      else url = args[0];
27
28      DefaultHandler handler = new DefaultHandler()
29        {
30          public void startElement(String namespaceURI, String lname,
31              String qname, Attributes attrs)
32          {
33            if (lname.equals("a") && attrs != null)
34            {
35              for (int i = 0; i < attrs.getLength(); i++)
36              {
37                String aname = attrs.getLocalName(i);
38                if (aname.equals("href"))
39                    System.out.println(attrs.getValue(i));
40              }
41            }
42          }
43        };
44
45        SAXParserFactory factory = SAXParserFactory.newInstance();
46        factory.setNamespaceAware(true);
47        factory.setFeature(
48    "http://apache.org/xml/features/nonvalidating/load-external-dtd", false);
49        SAXParser saxParser = factory.newSAXParser();
50        InputStream in = new URL(url).openStream();
51        saxParser.parse(in, handler);
52      }
53  }
```

javax.xml.parsers.SAXParserFactory 1.4

- `static SAXParserFactory newInstance()`

  Возвращает экземпляр класса `SAXParserFactory`.

- `SAXParser newSAXParser()`

  Возвращает экземпляр класса `SAXParser`.

- `boolean isNamespaceAware()`
- `void setNamespaceAware(boolean value)`

  Получают или устанавливают значение свойства `namespaceAware`, определяющего режим управления пространствами имен в фабрике. Если в этом свойстве установлено логическое значение `true`, то режим управления пространствами имен активизирован для синтаксических анализаторов, генерируемых фабрикой.

- `boolean isValidating()`
- `void setValidating(boolean value)`

  Получают или устанавливают значение свойства `validating`, определяющего режим проверки достоверности вводимых данных в фабрике. Если в этом свойстве установлено логическое значение `true`, то режим проверки достоверности вводимых данных активизирован для синтаксических анализаторов, генерируемых фабрикой.

javax.xml.parsers.SAXParser 1.4

- `void parse(File f, DefaultHandler handler)`
- `void parse(String url, DefaultHandler handler)`
- `void parse(InputStream in, DefaultHandler handler)`

  Выполняют синтаксический анализ XML-документа, полученного из файла по указанному URL или из потока ввода, а также оповещают заданный обработчик о событиях, наступающих в ходе синтаксического анализа.

org.xml.sax.ContentHandler 1.4

- `void startDocument()`
- `void endDocument()`

  Вызываются в начале и в конце XML-документа соответственно.

- `void startElement(String uri, String lname, String qname, Attributes attr)`
- `void endElement(String uri, String lname, String qname)`

  Вызываются в начале и в конце элемента разметки соответственно.

  Параметры:

uri	URI пространства имен (если включен режим управления пространствами имен)
lname	Локальное имя без префикса (если включен режим управления пространствами имен)
qname	Имя элемента (если отключен режим управления пространствами имен) или уточненное имя с префиксом (если анализатор сообщает уточненные имена, помимо локальных имен)

- `void characters(char[] data, int start, int length)`

  Вызывается при получении символьных данных.

  Параметры:

data	Массив символьных данных
start	Индекс первого символа в массиве, составляющем часть сообщаемых символьных данных
length	Длина сообщаемой символьной строки

org.xml.sax.Attributes 1.4

- `int getLength()`
  Возвращает количество атрибутов, хранящихся в коллекции атрибутов.
- `String getLocalName(int` *index*`)`
  Возвращает локальное имя (без префикса) атрибута по указанному индексу или пустую символьную строку, если для синтаксического анализатора не включен режим управления пространствами имен.
- `String getURI(int` *index*`)`
  Возвращает URI пространства имен для атрибута по указанному индексу или пустую символьную строку, если узел не является частью пространства имен или же если для синтаксического анализатора не включен режим управления пространствами имен.
- `String getQName(int` *index*`)`
  Возвращает уточненное имя (с префиксом) атрибута по указанному индексу или пустую символьную строку, если уточненное имя не сообщено синтаксическим анализатором.
- `String getValue(int` *index*`)`
- `String getValue(String` *qname*`)`
- `String getValue(String` *uri*`, String` *lname*`)`
  Возвращают значение атрибута по указанному индексу, уточненное имя или URI пространства имен вместе с локальным именем. Если такое значение отсутствует, то возвращается пустое значение `null`.

Применение StAX-анализатора

StAX-анализатор является "извлекающим синтаксическим анализатором". Вместо того чтобы устанавливать обработчик событий, достаточно произвести перебор событий в следующем цикле:

```
InputStream in = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);
while (parser.hasNext())
{
  int event = parser.next();
  вызвать методы синтаксического анализатора, чтобы
  получить подробные сведения о событии
}
```

Например, в ходе синтаксического анализа следующего фрагмента разметки:

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

синтаксический анализатор выдает перечисленные ниже события.

1. `START_ELEMENT`, имя элемента: `font`.
2. `CHARACTERS`, содержимое: пробел.
3. `START_ELEMENT`, имя элемента: `name`.
4. `CHARACTERS`, содержимое: `Helvetica`.

5. END_ELEMENT, имя элемента: name.
6. CHARACTERS, содержимое: пробел.
7. START_ELEMENT, имя элемента: size.
8. CHARACTERS, содержимое: **36**.
9. END_ELEMENT, имя элемента: size.
10. CHARACTERS, содержимое: пробел.
11. END_ELEMENT, имя элемента: font.

Чтобы проанализировать значения атрибутов, следует вызвать соответствующие методы из класса XMLStreamReader. Например, при вызове следующего метода получается атрибут units текущего элемента:

```
String units = parser.getAttributeValue(null, "units");
```

По умолчанию режим управления пространствами имен включен. Отключить его можно, видоизменив фабрику следующим образом:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, false);
```

В листинге 2.9 приведен исходный код программы поискового робота, реализованной вместе со StAX-анализатором. Нетрудно заметить, что исходный код этой программы намного проще, чем код аналогичной программы со SAX-анализатором, поскольку в данном случае не нужно организовывать обработку событий.

Листинг 2.9. Исходный код из файла stax/StAXTest.java

```
1  package stax;
2
3  import java.io.*;
4  import java.net.*;
5  import javax.xml.stream.*;
6
7  /**
8   * В этой программе демонстрируется применение StAX-анализатора.
9   * Программа выводит все гиперссылки из веб-страницы формата XHTML
10  * Использование: java StAXTest url
11  * @author Cay Horstmann
12  * @version 1.0 2007-06-23
13 */
14 public class StAXTest
15 {
16   public static void main(String[] args) throws Exception
17   {
18     String urlString;
19     if (args.length == 0)
20     {
21       urlString = "http://www.w3c.org";
22       System.out.println("Using " + urlString);
23     }
24     else urlString = args[0];
25     URL url = new URL(urlString);
26     InputStream in = url.openStream();
27     XMLInputFactory factory = XMLInputFactory.newInstance();
28     XMLStreamReader parser = factory.createXMLStreamReader(in);
```

```
29      while (parser.hasNext())
30      { .
31        int event = parser.next();
32        if (event == XMLStreamConstants.START_ELEMENT)
33        {
34          if (parser.getLocalName().equals("a"))
35          {
36            String href = parser.getAttributeValue(null, "href");
37            if (href != null)
38              System.out.println(href);
39          }
40        }
41      }
42    }
43 }
```

javax.xml.stream.XMLInputFactory 6

- `static XMLInputFactory newInstance()`

  Возвращает экземпляр класса `XMLInputFactory`.

- `void setProperty(String name, Object value)`

  Задает свойство для данной фабрики или генерирует исключение типа `IllegalArgumentException`, если свойство не поддерживается или не допускает установку заданного значения. В реализации Java SE поддерживаются следующие свойства, допускающие установку логических значений:

• `"javax.xml.stream.isValidating"`	При логическом значении `false` (по умолчанию) документ не проверяется. В спецификации это свойство не требуется
• `"javax.xml.stream.isNamespaceAware"`	При логическом значении `true` (по умолчанию) обрабатываются пространства имен. В спецификации это свойство не требуется
• `"javax.xml.stream.isCoalescing"`	При логическом значении `false` (по умолчанию) соседние символы не объединяются
• `"javax.xml.stream.isReplacingEntityReferences"`	При логическом значении `true` (по умолчанию) ссылки на сущности заменяются и сообщаются в виде символьных данных

- "javax.xml.stream.isSupportingExternalEntities" При логическом значении true (по умолчанию) разрешаются внешние сущности. В спецификации не определяется никаких значений этого свойства по умолчанию
- "javax.xml.stream.supportDTD" При логическом значении true (по умолчанию) об описаниях DTD сообщается как о событиях

- XMLStreamReader createXMLStreamReader(InputStream *in*)
- XMLStreamReader createXMLStreamReader(InputStream *in*, String *characterEncoding*)
- XMLStreamReader createXMLStreamReader(Reader *in*)
- XMLStreamReader createXMLStreamReader(Source *in*)

Создают синтаксический анализатор, читающий данные из заданного потока ввода, потока чтения или источника JAXP.

javax.xml.stream.XMLStreamReader 6

- boolean hasNext()

  Возвращает логическое значение true, если существует другое событие синтаксического анализа.

- int next()

  Задает состояние синтаксического анализатора для последующего события синтаксического анализа и возвращает одну из следующих констант: START_ELEMENT, END_ELEMENT, CHARACTERS, START_DOCUMENT, END_DOCUMENT, CDATA, COMMENT, SPACE (игнорируемый пробел), PROCESSING_INSTRUCTION, ENTITY_REFERENCE, DTD.

- boolean isStartElement()
- boolean isEndElement()
- boolean isCharacters()
- boolean isWhiteSpace()

  Возвращают логическое значение true, если текущее событие связано с начальным элементом, конечным элементом, символьными данными или разделителем.

- QName getName()

- `String getLocalName()`

  Получают имя элемента в событии `START_ELEMENT` или `END_ELEMENT`.

- `String getText()`

  Возвращает символы события `CHARACTERS`, `COMMENT` или `CDATA`, замещающее значение для константы `ENTITY_REFERENCE` или внутреннее подмножество DTD.

- `int getAttributeCount()`
- `QName getAttributeName(int index)`
- `String getAttributeLocalName(int index)`
- `String getAttributeValue(int index)`

  Получает подсчет количества атрибутов, а также имена и значения атрибутов, при условии, что текущим событием является `START_ELEMENT`.

- `String getAttributeValue(String namespaceURI, String name)`

  Получает значение атрибута по данному имени, при условии, что текущим событием является `START_ELEMENT`. Если параметр *namespaceURI* принимает пустое значение `null`, пространство имен не проверяется.

Формирование XML-документов

После изучения способов написания программ на Java, предназначенных для чтения XML-документов, рассмотрим способы написания программ, выполняющих обратное действие, т.е. формирующих данные для вывода в формате XML. Разумеется, XML-документ можно сформировать, введя в программу последовательность вызовов метода `print()` и выводя с их помощью элементы разметки, атрибуты и текст. Но для создания такого громоздкого кода потребуется много времени и усилий. Кроме того, такой код, как правило, изобилует ошибками. Очень легко, например, ошибиться при употреблении специальных знаков " или < в значениях атрибутов и в тексте.

Намного удобнее создать древовидную структуру DOM, представляющую документ, а затем записать ее содержимое в файл. Подробности такого подхода к формированию XML-документов обсуждаются в последующих разделах.

XML-документы без пространств имен

Для того чтобы построить древовидную структуру DOM, нужно сначала сформировать пустой документ с помощью метода `newDocument()` из класса `DocumentBuilder` следующим образом:

```
Document doc = builder.newDocument();
```

Затем следует вызвать метод `createElement()` из класса `Document`, чтобы построить элементы документа, как показано ниже.

```
Element rootElement  = doc.createElement(rootName);
Element childElement = doc.createElement(childName);
```

Далее создаются текстовые узлы с помощью метода `createTextNode()`:

```
Text textNode = doc.createTextNode(textContents);
```

После этого в документ вводится корневой элемент и дочерние узлы связываются с родительскими:

```
doc.appendChild(rootElement);
rootElement.appendChild(childElement);
childElement.appendChild(textNode);
```

При построении древовидной структуры DOM атрибуты элементов задаются с помощью метода `setAttribute()` из класса `Element` следующим образом:

```
rootElement.setAttribute(name, value);
```

XML-документы с пространствами имен

Если используются пространства имен, то процедура формирования XML-документа оказывается несколько иной. Сначала фабрика построителей документов устанавливается в режим управления пространствами имен, а затем создается построитель документов, как показано ниже.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
builder = factory.newDocumentBuilder();
```

Далее для создания любых узлов вместо метода `createElement()` вызывается метод `createElementNS()`:

```
String namespace = "http://www.w3.org/2000/svg";
Element rootElement = doc.createElementNS(namespace, "svg");
```

Если узел имеет уточненное имя с префиксом пространства имен, то любые требующиеся атрибуты с префиксом `xmlns` создаются автоматически. Так, если требуется ввести данные формата SVG в XHTML-документ, для этой цели можно построить соответствующий элемент аналогично приведенному ниже.

```
Element svgElement = doc.createElement(namespace, "svg:svg")
```

А когда этот элемент записывается, он превращается в следующий элемент разметки:

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg">
```

Если же требуется установить атрибуты элемента разметки, имена которых находятся в отдельном пространстве имен, для этой цели вызывается метод `setAttributeNS()` из класса `Element`:

```
rootElement.setAttributeNS(namespace, qualifiedName, value);
```

Запись XML-документов

Как ни странно, записать древовидную структуру DOM в поток вывода не так-то просто. Для этой цели проще всего воспользоваться прикладным интерфейсом API языка XSLT (Extensible Stylesheet Language Transformations — расширяемый язык преобразования XML-документов). Более подробно данный язык рассматривается в последнем разделе этой главы, а до тех пор допустим, что приведенный

ниже код каким-то волшебным образом позволяет получить данные, выводимые в формате XML.

Над документом выполняется холостое преобразование, а результат записывается в поток вывода. Для того чтобы включить узел DOCTYPE в выводимые данные, нужно также указать идентификаторы SYSTEM и PUBLIC в качестве свойств вывода.

```
// построить холостое преобразование
Transformer t = TransformerFactory.newInstance().newTransformer();
// установить свойства вывода, чтобы получить узел DOCTYPE
t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, systemIdentifier);
t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, publicIdentifier);
// установить отступ
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty(OutputKeys.METHOD, "xml");
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
// выполнить холостое преобразование и вывести результат в файл
t.transform(new DOMSource(doc), new StreamResult(new FileOutputStream(file)));
```

Еще один способ записи XML-документов состоит в применении интерфейса LSSerializer. Для получения экземпляра класса, реализующего этот интерфейс, служит следующий фрагмент кода:

```
DOMImplementation impl = doc.getImplementation();
DOMImplementationLS implLS = (DOMImplementationLS) impl.getFeature("LS", "3.0");
LSSerializer ser = implLS.createLSSerializer();
```

Если требуется ввести пробелы и разрывы строк в документ, для этого достаточно установить следующий признак:

```
ser.getDomConfig().setParameter("format-pretty-print", true);
```

И тогда преобразовать документ в символьную строку не составит особого труда:

```
String str = ser.writeToString(doc);
```

Если же требуется вывести документ непосредственно в файл, для этого нужно создать объект типа LSOutput следующим образом:

```
LSOutput out = implLS.createLSOutput();
out.setEncoding("UTF-8");
out.setByteStream(Files.newOutputStream(path));
ser.write(doc, out);
```

Пример формирования файла в формате SVG

В листинге 2.10, приведенном в следующем разделе, представлен исходный код примера программы для вывода XML-документа. Эта программа рисует картину в модернистском стиле из произвольного набора прямоугольников разного цвета (рис. 2.6). Для сохранения результатов используется формат SVG (Scalable Vector Format — масштабируемая векторная графика). По существу, формат SVG является разновидностью формата XML и служит для описания сложной графики в машинно-независимом виде. Дополнительные сведения об этом формате можно найти по адресу http://www.w3c.org/Graphics/SVG. Для просмотра файлов в формате SVG достаточно воспользоваться любым современным браузером.

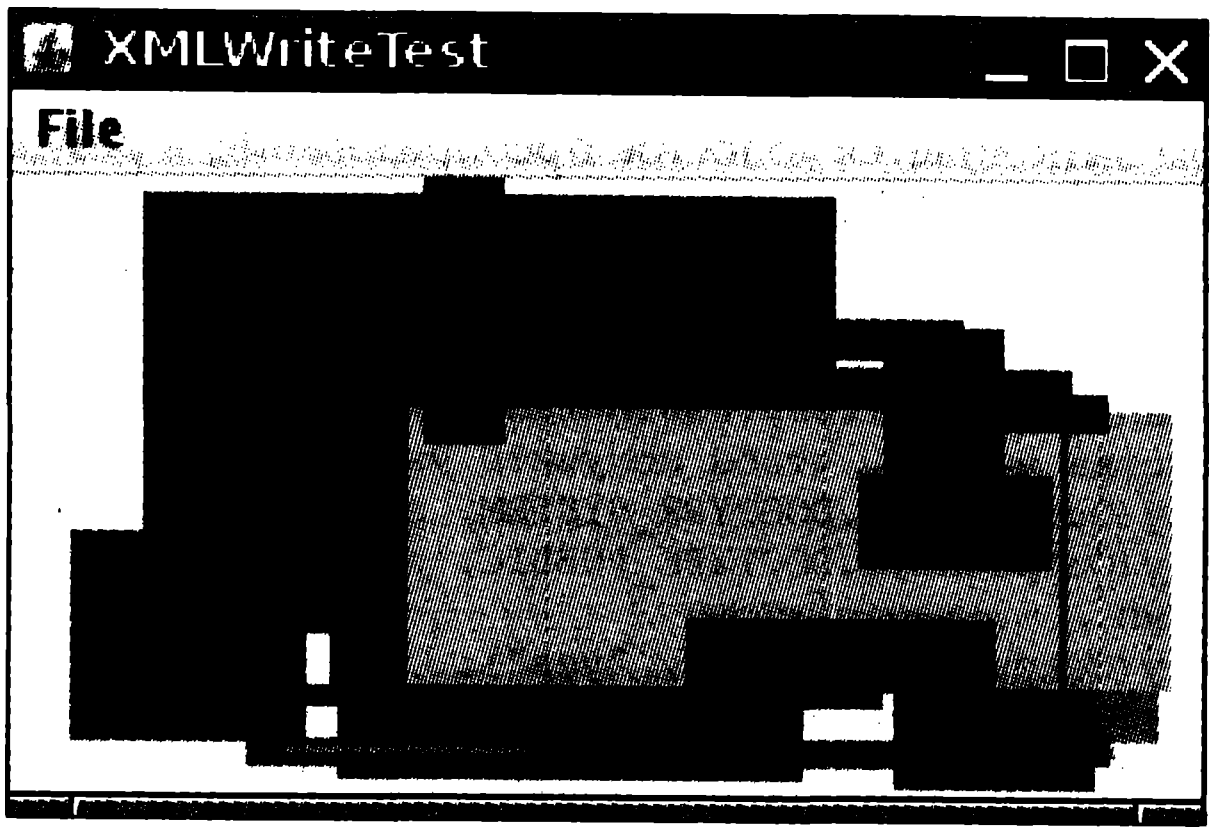


Рис. 2.6. Изображение в модернистском стиле, сохраняемое в формате SVG

Мы не будем вдаваться в подробности формата SVG, отсылая интересующихся за дополнительными сведениями по указанному выше адресу. Для целей рассматриваемого здесь примера достаточно знать, каким образом набор цветных прямоугольников размечается в формате SVG. Ниже приведен пример такой разметки.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
  "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd">
<svg xmlns="http://www.w3.org/2000/svg" width="300" height="150">
<rect x="231" y="61" width="9" height="12" fill="#6e4a13"/>
<rect x="107" y="106" width="56" height="5" fill="#c406be"/>
. . .
</svg>
```

Как следует из приведенного ниже фрагмента разметки, каждый прямоугольник описывается в виде узла `rect`, атрибуты которого задают координаты, ширину, высоту и цвет прямоугольника. Цвет заливки прямоугольников обозначается в виде значений основных цветов RGB в шестнадцатеричном формате.

НА ЗАМЕТКУ! В формате SVG широко применяются атрибуты. На самом деле некоторые из них имеют очень сложную структуру. В качестве примера ниже приведен элемент разметки контура.

```
<path d="M 100 100 L 300 100 L 200 300 z">
```

Здесь **M** обозначает команду `moveto` (перейти), **L** — команду `lineto` (нарисовать линию), **z** — команду `closepath` (замкнуть контур). Вероятно, создатели формата SVG не особенно доверяли формату XML. Очевидно, что вместо таких сложных атрибутов следовало бы использовать элементы разметки в коде XML.

javax.xml.parsers.DocumentBuilder 1.4

- `Document newDocument()`

  Возвращает пустой документ.

org.w3c.dom.Document 1.4

- `Element createElement(String name)`
- `Element createElementNS(String uri, String qname)`

  Создают элемент с заданным именем.

- `Text createTextNode(String data)`

  Создает текстовый узел с указанными данными.

org.w3c.dom.Node 1.4

- `Node appendChild(Node child)`

  Присоединяет узел к списку его дочерних узлов. Возвращает присоединенный узел.

org.w3c.dom.Element 1.4

- `void setAttribute(String name, String value)`
- `void setAttributeNS(String uri, String qname, String value)`

  Устанавливают заданное значение в атрибуте с указанным именем.

  Параметры:

uri	URI пространства имен или пустое значение `null`
qname	Уточненное имя. Если оно имеет префикс, то параметр `uri` не должен принимать пустое значение `null`
value	Значение атрибута

javax.xml.transform.TransformerFactory 1.4

- `static TransformerFactory newInstance()`

  Возвращает экземпляр класса `TransformerFactory`.

- `Transformer newTransformer()`

  Возвращает экземпляр класса `Transformer`, выполняющий тождественное (холостое) преобразование, не предполагающее никаких действий.

javax.xml.transform.Transformer 1.4

- `void setOutputProperty(String name, String value)`

  Задает свойство вывода. Перечень этих свойств можно найти по адресу `http://www.w3c.org/TR/xslt#output`. А ниже перечислены наиболее употребительные свойства вывода.

• `doctype-public`	Идентификатор `PUBLIC`, используемый в объявлении `DOCTYPE`
• `doctype-system`	Идентификатор `SYSTEM`, используемый в объявлении `DOCTYPE`
• `indent`	Принимает значение `"yes"` или `"no"`
• `method`	Принимает значение `"xml"`, `"html"`, `"text"` или специальное строковое значение

- `void transform(Source from, Result to)`

  Выполняет преобразование XML-документа.

javax.xml.transform.dom.DOMSource 1.4

- `DOMSource(Node n)`

  Создает источник данных из заданного узла. Обычно *n* обозначает узел документа.

javax.xml.transform.stream.StreamResult 1.4

- `StreamResult(File f)`
- `StreamResult(OutputStream out)`
- `StreamResult(Writer out)`
- `StreamResult(String systemID)`

  Создают поток вывода результатов преобразования на основе указанного файла, потока вывода, потока записи или системного идентификатора (как правило, это относительный или абсолютный URL).

Запись XML-документов средствами StAX

В предыдущем разделе было показано, как XML-документ формируется путем записи древовидной структуры DOM. Но если древовидная структура DOM нигде больше не используется, то такой способ оказывается не особенно эффективным. Потоковый прикладной интерфейс для XML (StAX) позволяет записывать древовидную структуру непосредственно в формате XML. Для этого следует создать поток записи типа `XMLStreamWriter` из потока вывода типа `OutputStream`, как показано ниже.

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(out);
```

А для того чтобы создать и вывести заголовок XML-документа, нужно вызвать сначала метод

```
writer.writeStartDocument()
```

а затем метод

```
writer.writeStartElement(name);
```

Далее, для вывода атрибутов следует вызвать приведенный ниже метод.

```
writer.writeAttribute(name, value);
```

Теперь можно вывести дочерние элементы разметки, снова вызвав метод `writeStartElement()`, или записать символы, вызвав следующий метод:

```
writer.writeCharacters(text);
```

После записи всех дочерних узлов следует вызвать приведенный ниже метод, который закроет текущий элемент разметки.

```
writer.writeEndElement();
```

Чтобы записать элемент разметки без дочерних элементов (например, элемент `<img ...>`), следует вызвать следующий метод:

```
writer.writeEmptyElement(name);
```

И наконец, для завершения записи в конце документа вызывается приведенный ниже метод. Этот метод закрывает все открытые элементы разметки.

```
writer.writeEndDocument();
```

Как и при применении DOM/XSLT, в данном случае можно не особенно беспокоиться о пропуске символов в значениях атрибутов и символьных данных. Но в этом случае существует вероятность того, что XML-документ будет сформирован не совсем удачно, например, со многими корневыми узлами. Кроме того, в текущей версии прикладного интерфейса StAX не поддерживается вывод XML-документов с отступами.

В примере программы из листинга 2.10 демонстрируется применение каждого из рассмотренных выше способов записи XML-документов. А в листингах 2.11 и 2.12 приведен исходный код классов фрейма и его компонентов для рисования заполняемых цветом прямоугольников.

Листинг 2.10. Исходный код из файла `write/XMLWriteTest.java`

```
1  package write;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  /**
7   * В этой программе демонстрируется запись XML-документа в файл.
8   * Сохраняемый файл описывает модернистский рисунок в формате SVG
9  * @version 1.11 2012-01-26
10 * @author Cay Horstmann
11 */
12 public class XMLWriteTest
13 {
14   public static void main(String[] args)
15   {
16     EventQueue.invokeLater(new Runnable()
```

```
17      {
18        public void run()
19        {
20          JFrame frame = new XMLWriteFrame();
21          frame.setTitle("XMLWriteTest");
22          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23          frame.setVisible(true);
24        }
25      });
26   }
27 }
```

Листинг 2.11. Исходный код из файла `write/XMLWriteFrame.java`

```
1  package write;
2
3  import java.awt.event.*;
4  import java.beans.*;
5  import java.io.*;
6  import java.nio.file.*;
7  import javax.swing.*;
8  import javax.xml.stream.*;
9  import javax.xml.transform.*;
10 import javax.xml.transform.dom.*;
11 import javax.xml.transform.stream.*;
12 import org.w3c.dom.*;
13
14 /**
15    * Фрейм с компонентом для отображения модернистского рисунка
16 */
17 public class XMLWriteFrame extends JFrame
18 {
19   private RectangleComponent comp;
20   private JFileChooser chooser;
21
22   public XMLWriteFrame()
23   {
24     chooser = new JFileChooser();
25
26     // ввести компонент в фрейм
27     comp = new RectangleComponent();
28     add(comp);
29     // установить строку меню
30     JMenuBar menuBar = new JMenuBar();
31     setJMenuBar(menuBar);
32
33     JMenu menu = new JMenu("File");
34     menuBar.add(menu);
35
36     JMenuItem newItem = new JMenuItem("New");
37     menu.add(newItem);
38     newItem.addActionListener(
39       EventHandler.create(ActionListener.class, comp, "newDrawing"));
40     JMenuItem saveItem = new JMenuItem("Save with DOM/XSLT");
41     menu.add(saveItem);
42     saveItem.addActionListener(
43       EventHandler.create(ActionListener.class, this, "saveDocument"));
```

```
44      JMenuItem saveStAXItem = new JMenuItem("Save with StAX");
45      menu.add(saveStAXItem);
46      saveStAXItem.addActionListener(
47        EventHandler.create(ActionListener.class, this, "saveStAX"));
48      JMenuItem exitItem = new JMenuItem("Exit");
49      menu.add(exitItem);
50      exitItem.addActionListener(new ActionListener()
51        {
52          public void actionPerformed(ActionEvent event)
53          {
54            System.exit(0);
55          }
56        });
57       pack();
58    }
59
60    /**
61      * Сохраняет рисунок в формате SVG, используя DOM/XSLT
62    */
63    public void saveDocument() throws TransformerException, IOException
64    {
65      if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
66      File file = chooser.getSelectedFile();
67      Document doc = comp.buildDocument();
68      Transformer t = TransformerFactory.newInstance().newTransformer();
69      t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
70        "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd");
71      t.setOutputProperty(
72        OutputKeys.DOCTYPE_PUBLIC, "-//W3C//DTD SVG 20000802//EN");
73      t.setOutputProperty(OutputKeys.INDENT, "yes");
74      t.setOutputProperty(OutputKeys.METHOD, "xml");
75      t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
76      t.transform(new DOMSource(doc),
77        new StreamResult(Files.newOutputStream(file.toPath())));
78    }
79    /**
80     * Сохраняет рисунок в формате SVG, используя StAX
81    */
82    public void saveStAX() throws IOException, XMLStreamException
83    {
84      if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
85      File file = chooser.getSelectedFile();
86      XMLOutputFactory factory = XMLOutputFactory.newInstance();
87      XMLStreamWriter writer =
88        factory.createXMLStreamWriter(Files.newOutputStream(file.toPath()));
89      try
90      {
91        comp.writeDocument(writer);
92      }
93        finally
94        {
95          writer.close(); // не закрывается автоматически
96        }
97    }
98 }
```

Листинг 2.12. Исходный код из файла `write/RectangleComponent.java`

```
1   package write;
2
3   import java.awt.*;
4   import java.awt.geom.*;
5   import java.util.*;
6   import javax.swing.*;
7   import javax.xml.parsers.*;
8   import javax.xml.stream.*;
9   import org.w3c.dom.*;
10
11  /**
12   * Компонент, отображающий ряд окрашенных прямоугольников
13  */
14  public class RectangleComponent extends JComponent
15  {
16    private static final int DEFAULT_WIDTH = 300;
17    private static final int DEFAULT_HEIGHT = 200;
18
19    private java.util.List<Rectangle2D> rects;
20    private java.util.List<Color> colors;
21    private Random generator;
22    private DocumentBuilder builder;
23    public RectangleComponent()
24    {
25      rects = new ArrayList<>();
26      colors = new ArrayList<>();
27      generator = new Random();
28
29      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
30      factory.setNamespaceAware(true);
31      try
32      {
33        builder = factory.newDocumentBuilder();
34      }
35      catch (ParserConfigurationException e)
36      {
37        e.printStackTrace();
38      }
39    }
40
41    /**
42      * Создать новый произвольный прямоугольник
43    */
44    public void newDrawing()
45    {
46      int n = 10 + generator.nextInt(20);
47      rects.clear();
48      colors.clear();
49      for (int i = 1; i <= n; i++)
50      {
51        int x = generator.nextInt(getWidth());
52        int y = generator.nextInt(getHeight());
53        int width = generator.nextInt(getWidth() - x);
54        int height = generator.nextInt(getHeight() - y);
55        rects.add(new Rectangle(x, y, width, height));
```

```
56          int r = generator.nextInt(256);
57          int g = generator.nextInt(256);
58          int b = generator.nextInt(256);
59          colors.add(new Color(r, g, b));
60        }
61        repaint();
62      }
63
64      public void paintComponent(Graphics g)
65      {
66        if (rects.size() == 0) newDrawing();
67        Graphics2D g2 = (Graphics2D) g;
68
69        // нарисовать все прямоугольники
70        for (int i = 0; i < rects.size(); i++)
71        {
72          g2.setPaint(colors.get(i));
73          g2.fill(rects.get(i));
74        }
75      }
76
77      /**
78       * Создает документ формата SVG с текущим рисунком
79       * @return Возвращает дерево DOM для документа формата SVG
80      */
81      public Document buildDocument()
82      {
83        String namespace = "http://www.w3.org/2000/svg";
84        Document doc = builder.newDocument();
85        Element svgElement = doc.createElementNS(namespace, "svg");
86        doc.appendChild(svgElement);
87        svgElement.setAttribute("width", "" + getWidth());
88        svgElement.setAttribute("height", "" + getHeight());
89        for (int i = 0; i < rects.size(); i++)
90        {
91          Color c = colors.get(i);
92          Rectangle2D r = rects.get(i);
93          Element rectElement = doc.createElementNS(namespace, "rect");
94          rectElement.setAttribute("x", "" + r.getX());
95          rectElement.setAttribute("y", "" + r.getY());
96          rectElement.setAttribute("width", "" + r.getWidth());
97          rectElement.setAttribute("height", "" + r.getHeight());
98          rectElement.setAttribute("fill", colorToString(c));
99          svgElement.appendChild(rectElement);
100       }
101       return doc;
102     }
103
104     /**
105      * Записывает документ формата SVG с текущим рисунком
106      * @param writer Место назначения документа
107     */
108     public void writeDocument(XMLStreamWriter writer)
109           throws XMLStreamException
110     {
111       writer.writeStartDocument();
112       writer.writeDTD(
```

```
113    "<!DOCTYPE svg PUBLIC \"-//W3C//DTD SVG 20000802//EN\" " +
114    "\"http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd\">");
115      writer.writeStartElement("svg");
116      writer.writeDefaultNamespace("http://www.w3.org/2000/svg");
117      writer.writeAttribute("width", "" + getWidth());
118      writer.writeAttribute("height", "" + getHeight());
119      for (int i = 0; i < rects.size(); i++)
120      {
121        Color c = colors.get(i);
122        Rectangle2D r = rects.get(i);
123        writer.writeEmptyElement("rect");
124        writer.writeAttribute("x", "" + r.getX());
125        writer.writeAttribute("y", "" + r.getY());
126        writer.writeAttribute("width", "" + r.getWidth());
127        writer.writeAttribute("height", "" + r.getHeight());
128        writer.writeAttribute("fill", colorToString(c));
129      }
130      writer.writeEndDocument(); // закрывает элемент разметки svg
131    }
132
133    /**
134     * Преобразует значение цвета в шестнадцатеричный формат
135     * @param c Цвет
136     * @return Возвращает символьную строку в формате #rrggbb
137    */
138    private static String colorToString(Color c)
139    {
140      StringBuffer buffer = new StringBuffer();
141      buffer.append(Integer.toHexString(c.getRGB() & 0xFFFFFF));
142      while (buffer.length() < 6)
143      buffer.insert(0, '0');
144      buffer.insert(0, '#');
145      return buffer.toString();
146    }
147
148    public Dimension getPreferredSize()
149      { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
150  }
```

javax.xml.stream.XMLOutputFactory 6

- `static XMLOutputFactory newInstance()`

  Возвращает экземпляр класса `XMLOutputFactory`.

- `XMLStreamWriter createXMLStreamWriter(OutputStream` *in*`)`
- `XMLStreamWriter createXMLStreamWriter(OutputStream` *in*`,`
  `String` *characterEncoding*`)`
- `XMLStreamWriter createXMLStreamWriter(Writer` *in*`)`
- `XMLStreamWriter createXMLStreamWriter(Result` *in*`)`

  Создают поток, записывающий данные в указанный поток вывода, поток записи или поток вывода результатов типа JAXP.

javax.xml.stream.XMLStreamWriter 6

- `void writeStartDocument()`
- `void writeStartDocument(String xmlVersion)`
- `void writeStartDocument(String encoding, String xmlVersion)`

  Записывают инструкцию обработки вначале XML-документа. Следует иметь в виду, что параметр `encoding` указывается только для записи атрибута. Он не задает кодировку символов в выводимых данных.
- `void setDefaultNamespace(String namespaceURI)`
- `void setPrefix(String prefix, String namespaceURI)`

  Задают пространство имен по умолчанию или пространство имен, связанное с префиксом. Объявление действительно для текущего элемента или же для корня документа, если ни одного элемента не было записано.
- `void writeStartElement(String localName)`
- `void writeStartElement(String namespaceURI, String localName)`

  Записывают первоначальный дескриптор, заменяя параметр *namespaceURI* связанным префиксом.
- `void writeEndElement()`

  Закрывает текущий элемент разметки.
- `void writeEndDocument()`

  Закрывает все открытые элементы разметки.
- `void writeEmptyElement(String localName)`
- `void writeEmptyElement(String namespaceURI, String localName)`

  Записывают самозакрывающийся дескриптор, заменяя параметр `namespaceURI` связанным префиксом.
- `void writeAttribute(String localName, String value)`
- `void writeAttribute(String namespaceURI, String localName, String value)`

  Записывают атрибут для текущего элемента разметки, заменяя параметр `namespaceURI` связанным префиксом.
- `void writeCharacters(String text)`
- Записывает символьные данные.
- `void writeCData(String text)`
- Записывает раздел CDATA.
- `void writeDTD(String dtd)`
- Записывает символьную строку *dtd*, которая должна содержать объявление `DOCTYPE`.
- `void writeComment(String comment)`
- Записывает комментарий.
- `void close()`
- Закрывает записывающий поток.

Преобразование XML-документов языковыми средствами XSLT

Язык преобразования XML-документов (XSLT) позволяет определять правила преобразования подобных документов в другие форматы, включая простой текст, XHTML или любую другую разновидность формата XML. Язык XSLT обычно применяется для перевода из одной машиночитаемой разновидности формата XML в другую машиночитаемую или удобочитаемую (для человека) разновидность этого формата.

Для этой цели нужно создать таблицу стилей XSLT, описывающую преобразование XML-документов в какой-нибудь другой формат. Процессор XSLT сначала читает XML-документ и таблицу стилей, а затем выдает желаемый результат (рис. 2.7).

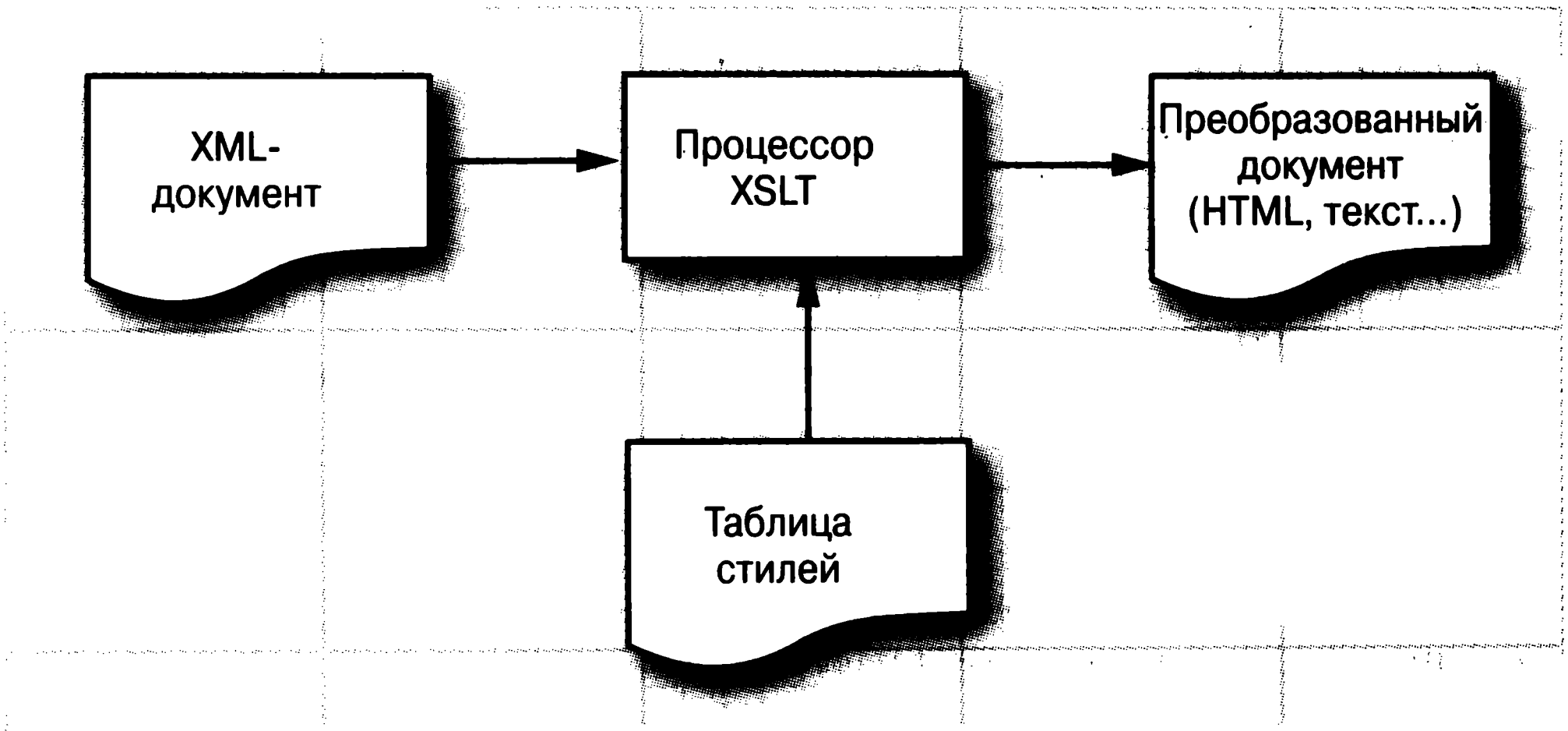


Рис. 2.7. Преобразование XML-документа языковыми средствами XMLT

Рассмотрим характерный пример преобразования XML-документа с записями о сотрудниках в HTML-документ. Ниже приведена разметка исходного XML-документа.

```
<staff>
  <employee>
    <name>Carl Cracker</name>
    <salary>75000</salary>
    <hiredate year="1987" month="12" day="15"/>
  </employee>
  <employee>
    <name>Harry Hacker</name>
    <salary>50000</salary>
    <hiredate year="1989" month="10" day="1"/>
  </employee>
  <employee>
    <name>Tony Tester</name>
    <salary>40000</salary>
    <hiredate year="1990" month="3" day="15"/>
  </employee>
</staff>
```

Из этой разметки желательно получить следующую HTML-таблицу:

```
<table border="1">
<tr>
```

```
<td>Carl Cracker</td><td>$75000.0</td><td>1987-12-15</td>
</tr>
<tr>
<td>Harry Hacker</td><td>$50000.0</td><td>1989-10-1</td>
</tr>
<tr>
<td>Tony Tester</td><td>$40000.0</td><td>1990-3-15</td>
</tr>
</table>
```

Спецификация языка XSLT имеет чрезвычайно сложную структуру, а ее описанию посвящена довольно обширная литература. Поэтому ограничимся обсуждением лишь некоторых элементов, которые используются в рассматриваемом здесь примере. Дополнительные сведения по данному вопросу можно найти в упоминавшейся ранее книге *Essential XML* Дона Бокса и др., а описание спецификации XSLT — по адресу `http://www.w3c.org/TR/xslt`.

Вот как выглядит таблица стилей с шаблонами преобразования:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html"/>
    шаблон1
    шаблон2
    . . .
</xsl:stylesheet>
```

В данном примере элемент разметки `xsl:output` содержит атрибут `method` со значением `"html"` для преобразования в формат HTML. Для данного атрибута можно также задать значения `xml` и `text`. Типичный шаблон имеет следующий вид:

```
<xsl:template match="/staff/employee">
  <tr><xsl:apply-templates/></tr>
</xsl:template>
```

Значением атрибута `match` является выражение XPath. Данный шаблон означает, что при обнаружении узла в древовидной структуре `/staff/employee` нужно выполнить следующие действия.

1. Сформировать символьную строку с открывающим дескриптором `<tr>`.
2. Применять шаблоны при обработке дочерних узлов данного узла.
3. После обработки всех дочерних узлов сформировать символьную строку с закрывающим дескриптором `</tr>`.

Иными словами, этот шаблон заключает каждую запись о сотрудниках в дескрипторы строк HTML-таблицы.

Процессор XSLT начинает обработку XML-документа с проверки корневого элемента разметки. Если узел совпадает с одним из шаблонов, соответствующий шаблон сразу же применяется. (При совпадении с несколькими шаблонами используется шаблон с наибольшей степенью соответствия. Дополнительные сведения по данному вопросу приведены в спецификации языка XSLT по адресу `http://www.w3.org/TR/xslt`.) Если совпадение с шаблоном не обнаружено, то процессор XSLT выполняет действие, задаваемое по умолчанию. Так, содержимое текстовых узлов по умолчанию включается в выводимый результат, а для элементов разметки выводимый результат не формируется, но продолжается обработка дочерних элементов.

В качестве примера ниже приведен шаблон преобразования узлов name из XML-документа с данными о сотрудниках.

```
<xsl:template match="/staff/employee/name">
  <td><xsl:apply-templates/></td>
</xsl:template>
```

Как видите, шаблон формирует дескрипторы <td>...</td> и предписывает процессору XSLT рекурсивно обойти дочерние узлы элемента разметки name. У этого элемента имеется только один дочерний текстовый узел. Когда процессор обходит узел, он возвращает его текстовое содержимое, если, конечно, отсутствуют другие совпавшие шаблоны.

Для копирования значений атрибутов в выводимый результат придется задать более сложный шаблон:

```
<xsl:template match="/staff/employee/hiredate">
  <td><xsl:value-of select="@year"/>-<xsl:value-of
  select="@month"/>-<xsl:value-of select="@day"/></td>
</xsl:template>
```

При обработке узла hiredate по этому шаблону будут сформированы перечисленные ниже компоненты.

1. Дескриптор <td>.
2. Значение атрибута year.
3. Дефис.
4. Значение атрибута month.
5. Дефис.
6. Значение атрибута day.
7. Дефис.
8. Дескриптор </td>.

Оператор xsl:value-of вычисляет строковое значение множества узлов, указываемого с помощью значения XPath атрибута select. В этом случае путь определяется относительно текущего узла. Множество узлов преобразуется в символьную строку путем сцепления строковых значений из всех узлов. Строковым значением атрибута является его значение, строковым значением текстового узла — его содержимое, а строковым значением элемента разметки — сцепление строковых значений его дочерних узлов (но не атрибутов).

В листинге 2.13 приведена таблица стилей для преобразования XML-документа с записями о сотрудниках в HTML-таблицу.

Листинг 2.13. Исходный код из файла transform/makehtml.xsl

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <xsl:stylesheet
4    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5    version="1.0">
6
7    <xsl:output method="html"/>
8
```

```
9     <xsl:template match="/staff">
10      <table border="1"><xsl:apply-templates/></table>
11    </xsl:template>
12
13    <xsl:template match="/staff/employee">
14      <tr><xsl:apply-templates/></tr>
15    </xsl:template>
16
17    <xsl:template match="/staff/employee/name">
18      <td><xsl:apply-templates/></td>
19    </xsl:template>
20
21    <xsl:template match="/staff/employee/salary">
22      <td>$<xsl:apply-templates/></td>
23    </xsl:template>
24
25    <xsl:template match="/staff/employee/hiredate">
26      <td><xsl:value-of select="@year"/>-<xsl:value-of
27        select="@month"/>-<xsl:value-of select="@day"/></td>
28    </xsl:template>
29
30  </xsl:stylesheet>
```

А в листинге 2.14 приведены шаблоны для различных преобразований того же самого XML-документа в текст в знакомом уже формате файла свойств, как показано ниже.

```
employee.1.name=Carl Cracker
employee.1.salary=75000.0
employee.1.hiredate=1987-12-15
employee.2.name=Harry Hacker
employee.2.salary=50000.0
employee.2.hiredate=1989-10-1
employee.3.name=Tony Tester
employee.3.salary=40000.0
employee.3.hiredate=1990-3-15
```

Листинг 2.14. Исходный код из файла `transform/makeprop.xsl`

```
1   <?xml version="1.0"?>
2
3   <xsl:stylesheet
4     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5     version="1.0">
6
7     <xsl:output method="text" omit-xml-declaration="yes"/>
8
9     <xsl:template match="/staff/employee">
10  employee.<xsl:value-of select="position()"
11  />.name=<xsl:value-of select="name/text()"/>
12  employee.<xsl:value-of select="position()"
13  />.salary=<xsl:value-of select="salary/text()"/>
14  employee.<xsl:value-of select="position()"
15  />.hiredate=<xsl:value-of select="hiredate/@year"
16  />-<xsl:value-of select="hiredate/@month"
17  />-<xsl:value-of select="hiredate/@day"/>
18    </xsl:template>
19
20  </xsl:stylesheet>
```

В данном примере используется функция `position()`, которая выдает расположение текущего узла относительно родительского узла. Для того чтобы получить выводимый результат в совершенно другом виде, достаточно внести соответствующие изменения в таблицу стилей. Таким образом, XML-документ можно спокойно применять для представления данных, не особенно заботясь, в каком именно формате они требуются в прикладной программе. Для формирования данных в нужном формате достаточно воспользоваться средствами XSLT.

Преобразования средствами XSLT совсем не трудно организовать на платформе Java. С этой целью нужно создать сначала отдельную фабрику преобразователей для каждой таблицы стилей, а затем получить объект типа `Transformer` и передать ему преобразуемые данные, как показано ниже.

```
File styleSheet = new File(filename);
StreamSource styleSource = new StreamSource(styleSheet);
Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
t.transform(source, result);
```

Параметры, передаваемые методу `transform()`, представляют собой экземпляры классов, реализующих интерфейсы `Source` и `Result`. Так, у интерфейса `Source` имеются реализации источника данных в следующих классах:

```
DOMSource
SAXSource
StAXSource
StreamSource
```

Потоковый источник данных типа `StreamSource` можно создать из файла, потока ввода, потока чтения или URL, а источник данных типа `DOMSource` — из узла древовидной структуры DOM. Так, в предыдущем примере программы выполнялось следующее тождественное преобразование:

```
t.transform(new DOMSource(doc), result);
```

А в рассматриваемом здесь примере программы применяется другой, более интересный подход. Вместо уже существующего XML-файла создается поток, читающий XML-данные, имитируя их SAX-анализ с инициированием соответствующих SAX-событий. По существу, поток чтения XML-данных выполняет чтение из однородного входного файла со следующим содержимым:

```
Carl Cracker|75000.0|1987|12|15
Harry Hacker|50000.0|1989|10|1
Tony Tester|40000.0|1990|3|15
```

По мере обработки вводимых XML-данных в потоке чтения инициируются SAX-события. Ниже представлен фрагмент исходного кода метода `parse()` из класса `EmployeeReader`, реализующего интерфейс `XMLReader`.

```
AttributesImpl attributes = new AttributesImpl();
handler.startDocument();
handler.startElement("", "staff", "staff", attributes);
while ((line = in.readLine()) != null)
{
  handler.startElement("", "employee", "employee", attributes);
  StringTokenizer t = new StringTokenizer(line, "|");
  handler.startElement("", "name", "name", attributes);
  String s = t.nextToken();
  handler.characters(s.toCharArray(), 0, s.length());
  handler.endElement("", "name", "name");
```

```
   . . .
   handler.endElement("", "employee", "employee");
}
handler.endElement("", rootElement, rootElement);
handler.endDocument();
```

Источник типа `SAXSource` для преобразования данных создается из потока чтения XML-данных следующим образом:

```
t.transform(new SAXSource(new EmployeeReader(),
   new InputSource(new FileInputStream(filename))), result);
```

Такой искусный прием как нельзя лучше подходит для преобразования унаследованных данных в формат XML. Безусловно, для большинства приложений XSLT вводимые данные уже находятся в формате XML. А это позволяет просто вызвать метод `transform()` для объекта типа `SAXSource`:

```
t.transform(new StreamSource(file), result);
```

Результатом такого преобразования оказывается объект одного из следующих трех классов, реализующих интерфейс `Result` в библиотеке Java:

```
DOMResult
SAXResult
StreamResult
```

Для сохранения результата в виде древовидной структуры DOM используется объект типа `DocumentBuilder`. С его помощью генерируется новый узел документа, заключаемый в оболочку типа `DOMResult`, как показано ниже.

```
Document doc = builder.newDocument();
t.transform(source, new DOMResult(doc));
```

И наконец, для вывода полученного результата в файл используется объект типа `StreamResult`:

```
t.transform(source, new StreamResult(file));
```

В листинге 2.15 приведен весь исходный код рассмотренного здесь примера программы.

Листинге 2.15. Исходный код из файла `transform/TransformTest.java`

```
1   package transform;
2
3   import java.io.*;
4   import java.nio.file.*;
5   import java.util.*;
6   import javax.xml.transform.*;
7   import javax.xml.transform.sax.*;
8   import javax.xml.transform.stream.*;
9   import org.xml.sax.*;
10  import org.xml.sax.helpers.*;
11
12  /**
13   * В этой программе демонстрируются преобразования средствами XSLT.
14   * Преобразования выполняются над записями о сотрудниках из файла
15   * employee.dat в формат XML. Для этого нужно указать таблицу стилей
16   * в командной строке, например, следующим образом:
17   * java TransformTest makeprop.xsl
18   * @version 1.02 2012-06-04
```

```
19   * @author Cay Horstmann
20  */
21  public class TransformTest
22  {
23    public static void main(String[] args) throws Exception
24    {
25      Path path;
26      if (args.length > 0) path = Paths.get(args[0]);
27      else path = Paths.get("transform", "makehtml.xsl");
28      try (InputStream styleIn = Files.newInputStream(path))
29      {
30        StreamSource styleSource = new StreamSource(styleIn);
31        Transformer t =
32          TransformerFactory.newInstance().newTransformer(styleSource);
33        t.setOutputProperty(OutputKeys.INDENT, "yes");
34        t.setOutputProperty(OutputKeys.METHOD, "xml");
35        t.setOutputProperty(
36          "{http://xml.apache.org/xslt}indent-amount", "2");
37        try (InputStream docIn =
38          Files.newInputStream(Paths.get("transform", "employee.dat")))
39        {
40          t.transform(new SAXSource(new EmployeeReader(),
41            new InputSource(docIn)), new StreamResult(System.out));
42        }
43      }
44    }
45  }
46
47  /**
48   * Этот класс читает однородный файл employee.dat и уведомляет о
49   * событиях в SAX-анализаторе, как будто он сам выполнил
50   * синтаксический анализ XML-документа
51  */
52  class EmployeeReader implements XMLReader
53  {
54    private ContentHandler handler;
55
56    public void parse(InputSource source) throws IOException, SAXException
57    {
58      InputStream stream = source.getByteStream();
59      BufferedReader in = new BufferedReader(new InputStreamReader(stream));
60      String rootElement = "staff";
61      AttributesImpl atts = new AttributesImpl();
62
63      if (handler == null) throw new SAXException("No content handler");
64
65      handler.startDocument();
66      handler.startElement("", rootElement, rootElement, atts);
67      String line;
68      while ((line = in.readLine()) != null)
69      {
70        handler.startElement("", "employee", "employee", atts);
71        StringTokenizer t = new StringTokenizer(line, "|");
72
73        handler.startElement("", "name", "name", atts);
74        String s = t.nextToken();
75        handler.characters(s.toCharArray(), 0, s.length());
76        handler.endElement("", "name", "name");
```

```
77
78          handler.startElement("", "salary", "salary", atts);
79          s = t.nextToken();
80          handler.characters(s.toCharArray(), 0, s.length());
81          handler.endElement("", "salary", "salary");
82
83          atts.addAttribute("", "year", "year", "CDATA", t.nextToken());
84          atts.addAttribute("", "month", "month", "CDATA", t.nextToken());
85          atts.addAttribute("", "day", "day", "CDATA", t.nextToken());
86          handler.startElement("", "hiredate", "hiredate", atts);
87          handler.endElement("", "hiredate", "hiredate");
88          atts.clear();
89
90          handler.endElement("", "employee", "employee");
91        }
92
93        handler.endElement("", rootElement, rootElement);
94          handler.endDocument();
95      }
96
97      public void setContentHandler(ContentHandler newValue)
99      {
100       handler = newValue;
101     }
102
103     public ContentHandler getContentHandler()
104     {
105       return handler;
106     }
107
108     // следующие методы являются всего лишь холостыми реализациями
109     public void parse(String systemId) throws IOException, SAXException
110     {
111     }
112
113     public void setErrorHandler(ErrorHandler handler)
114     {
115     }
116
117     public ErrorHandler getErrorHandler()
118     {
119       return null;
120     }
121
122     public void setDTDHandler(DTDHandler handler)
123     {
124     }
125
126     public DTDHandler getDTDHandler()
127     {
128       return null;
129     }
130     public void setEntityResolver(EntityResolver resolver)
131     {
132     }
133
134     public EntityResolver getEntityResolver()
135     {
```

```
136       return null;
137    }
138
139    public void setProperty(String name, Object value)
140    {
141    }
142
143    public Object getProperty(String name)
144    {
145       return null;
146    }
147
148    public void setFeature(String name, boolean value)
149    {
150    }
151
152    public boolean getFeature(String name)
153    {
154       return false;
155    }
156 }
```

javax.xml.transform.TransformerFactory 1.4

- `Transformer newTransformer(Source` *styleSheet*`)`

  Возвращает экземпляр класса `Transformer`, считывающий таблицу стилей из указанного источника.

javax.xml.transform.stream.StreamSource 1.4

- `StreamSource(File` *f*`)`
- `StreamSource(InputStream` *in*`)`
- `StreamSource(Reader` *in*`)`
- `StreamSource(String` *systemID*`)`

  Создают потоковый источник данных из указанного файла, потока ввода, потока чтения или системного идентификатора (обычно это относительный или абсолютный URL).

javax.xml.transform.sax.SAXSource 1.4

- `SAXSource(XMLReader` *reader*`, InputSource` *source*`)`

  Создает SAX-источник, получающий вводимые данные из указанного источника, используя заданный поток чтения для синтаксического анализа данных.

org.xml.sax.XMLReader 1.4

- `void setContentHandler(ContentHandler` *handler*`)`

  Устанавливает обработчик, который уведомляется о событиях, наступающих при синтаксическом анализе вводимых данных.

- `void parse(InputSource` *source*`)`

  Анализирует данные, вводимые из указанного источника, и передает обработчику содержимого события, наступающие при синтаксическом анализе этих данных.

javax.xml.transform.dom.DOMResult 1.4

- `DOMResult(Node` *n*`)`
- Создает источник данных из заданного узла. Обычно в качестве параметра *n* указывается узел документа.

org.xml.sax.helpers.AttributesImpl 1.4

- `DOMResult(Node` *n*`)`

  Создает источник данных из заданного узла. Обычно в качестве параметра `n` указывается узел документа.

- `void addAttribute(String` *uri*`, String` *lname*`, String` *qname*`, String` *type*`, String` *value*`)`

  Вводит атрибут в коллекцию атрибутов.

  Параметры:

uri	URI пространства имен
lname	Локальное имя без префикса
qname	Уточненное имя с префикс
type	Один из следующих типов: `"CDATA"`, `"ID"`, `"IDREF"`, `"IDREFS"`, `"NMTOKEN"`, `"NMTOKENS"`, `"ENTITY"`, `"ENTITIES"` или `"NOTATION"`
value	Значение атрибута

- `void clear()`

  Удаляет все атрибуты из данной коллекции.

Этим примером завершается обсуждение особенностей поддержки XML в библиотеке Java. Теперь у вас должно сложиться ясное представление о возможностях XML, включая автоматизированный синтаксический анализ и проверку достоверности, а также эффективный механизм преобразования XML-документов. Естественно, что всю эту технологию вам удастся поставить себе на службу, если вы тщательно

разработаете свои форматы XML. Для этого вы должны обеспечить способность ваших форматов XML удовлетворять всем насущным производственным потребностям, их устойчивость с течением времени, а также выяснить готовность ваших деловых партнеров принимать от вас XML-документы. Решение всех этих вопросов может оказаться гораздо сложнее, чем умелое обращение с синтаксическими анализаторами, определениями DTD или преобразованиями, выполняемыми средствами XSLT.

В следующей главе будут обсуждаться вопросы сетевого программирования на платформе Java. Сначала мы рассмотрим основные положения о сетевых сокетах, а затем перейдем к высокоуровневым протоколам для электронной почты и Всемирной паутины.

ГЛАВА 3

Работа в сети

В этой главе...

- Подключение к серверу
- Реализация серверов
- Прерываемые сокеты
- Получение данных из Интернета
- Отправка электронной почты

Эта глава начинается с описания основных понятий для работы в сети, а затем в ней рассматриваются примеры написания программ на Java, позволяющих устанавливать соединения с серверами. Из нее вы узнаете, как осуществляется реализация сетевых клиентов и серверов. А завершается глава рассмотрением вопросов передачи почтовых сообщений из программы на Java и сбора данных с веб-сервера.

Подключение к серверу

Прежде чем приступать к написанию первой сетевой программы, надо познакомиться с отличным инструментальным средством для отладки сетевых программ: утилитой `telnet`. Следует, однако, иметь в виду, что она не относится к числу предварительно устанавливаемых компонентов операционной системы, поэтому обязательно проверьте, установлена ли она на вашем компьютере. Если утилиту `telnet` нельзя запустить из командной строки, еще раз запустите программу установки избранной операционной системы и выберите эту утилиту из списка предлагаемых для установки компонентов.

НА ЗАМЕТКУ! Вместе с ОС Windows Vista устанавливается и утилита `telnet`, но по умолчанию она не активизирована. Чтобы активизировать ее, откройте панель управления, перейдите в раздел **Программы**, щелкните на ссылке **Добавление или удаление компонентов Windows** и установите флажок **Клиент Telnet**. Следует также иметь в виду, что брандмауэр Windows блокирует некоторые сетевые порты, которые будут использоваться в примерах программ из этой главы. А для того чтобы разблокировать эти порты, вы должны обладать полномочиями администратора.

Утилитой `telnet` можно пользоваться не только для соединения с удаленным компьютером. С ее помощью можно также взаимодействовать с различными сетевыми службами. Ниже приводится один из примеров необычного использования этой утилиты. Для этого введите в командной строке следующую команду:

```
telnet time-A.timefreq.bldrdoc.gov 13
```

На рис. 3.1 приведен пример ответной реакции сервера, которая в режиме командной строки будет иметь следующий вид:

```
54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *
```

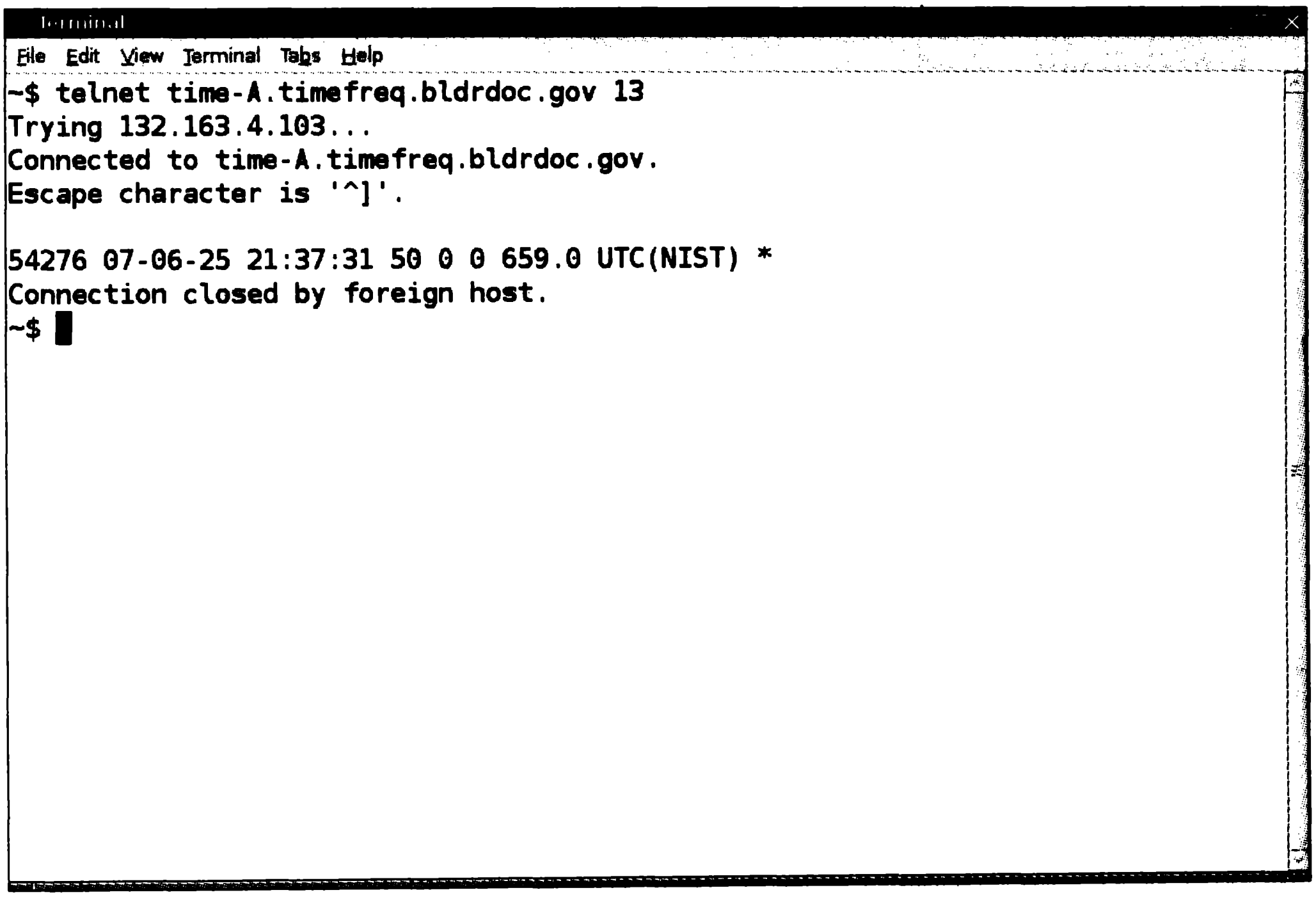


Рис. 3.1. Результат, получаемый из службы учета времени дня

Что же в действительности произошло? Утилита `telnet` подключилась к серверу службы учета времени дня, который работает на большинстве компьютеров под управлением операционной системы UNIX. Указанный в этом примере сервер находится в Национальном институте стандартов и технологий (National Institute of Standards and Technology) в г. Боулдер, штат Колорадо. Его системное время

синхронизировано с цезиевыми атомными часами. (Конечно, полученное значение текущего времени будет не совсем точным из-за задержек, связанных с передачей данных по сети.) По принятым правилам сервер службы времени всегда связан с портом **13**.

НА ЗАМЕТКУ! В сетевой терминологии *порт* — это не какое-то конкретное физическое устройство, а абстрактное понятие, упрощающее представление о соединении сервера с клиентом (рис. 3.2).

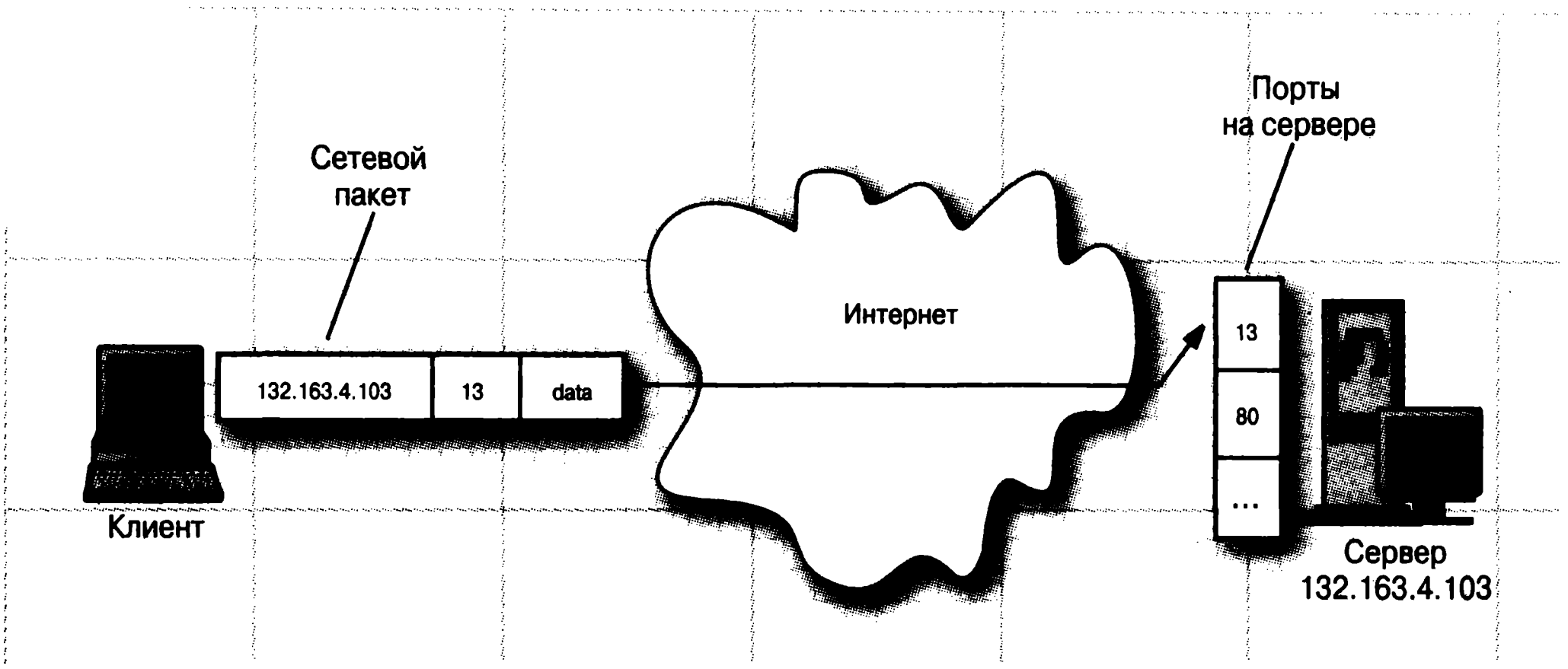


Рис. 3.2. Схема соединения клиента с сервером через конкретный порт

Программное обеспечение сервера постоянно работает на удаленном компьютере и ожидает поступления сетевого трафика через порт **13**. При получении операционной системой на удаленном компьютере сетевого пакета с запросом на подключение к порту **13** на сервере активизируется соответствующий процесс и устанавливается соединение. Такое соединение может быть прервано одним из его участников.

После запуска утилиты `telnet` с параметром `time-A.timefreq.bldrdoc.gov` с целью начать сеанс связи через порт **13** независимое сетевое программное обеспечение преобразует строку `"time-A.timefreq.bldrdoc.gov"` в IP-адрес **132.163.4.104**. Затем посылается запрос на соединение с заданным компьютером через порт **13**. После установления соединения программа на удаленном компьютере передает обратно строку с данными, а затем разрывает соединение. Разумеется, клиенты и серверы могут вести и более сложные диалоги до разрыва соединения.

Проведем еще один, более интересный эксперимент. С этой целью выполните следующие действия.

1. Введите в режиме командной строки команду
2. **`telnet horstmann.com 80`**
3. Затем аккуратно и точно введите следующее, *дважды* нажав клавишу <Enter> в конце:

```
GET / HTTP/1.1
Host: horstmann.com
пустая строка
```

На рис. 3.3 показана ответная реакция сервера в окне утилиты `telnet`. Она имеет уже знакомый вид страницы текста в формате HTML, а именно начальной страницы веб-сайта Кея Хорстманна. Именно так обычный веб-браузер получает искомые веб-страницы. Для запроса веб-страниц на сервере он применяет сетевой протокол HTTP. Разумеется, браузер отображает данные в намного более удобном для чтения виде, чем формат HTML.

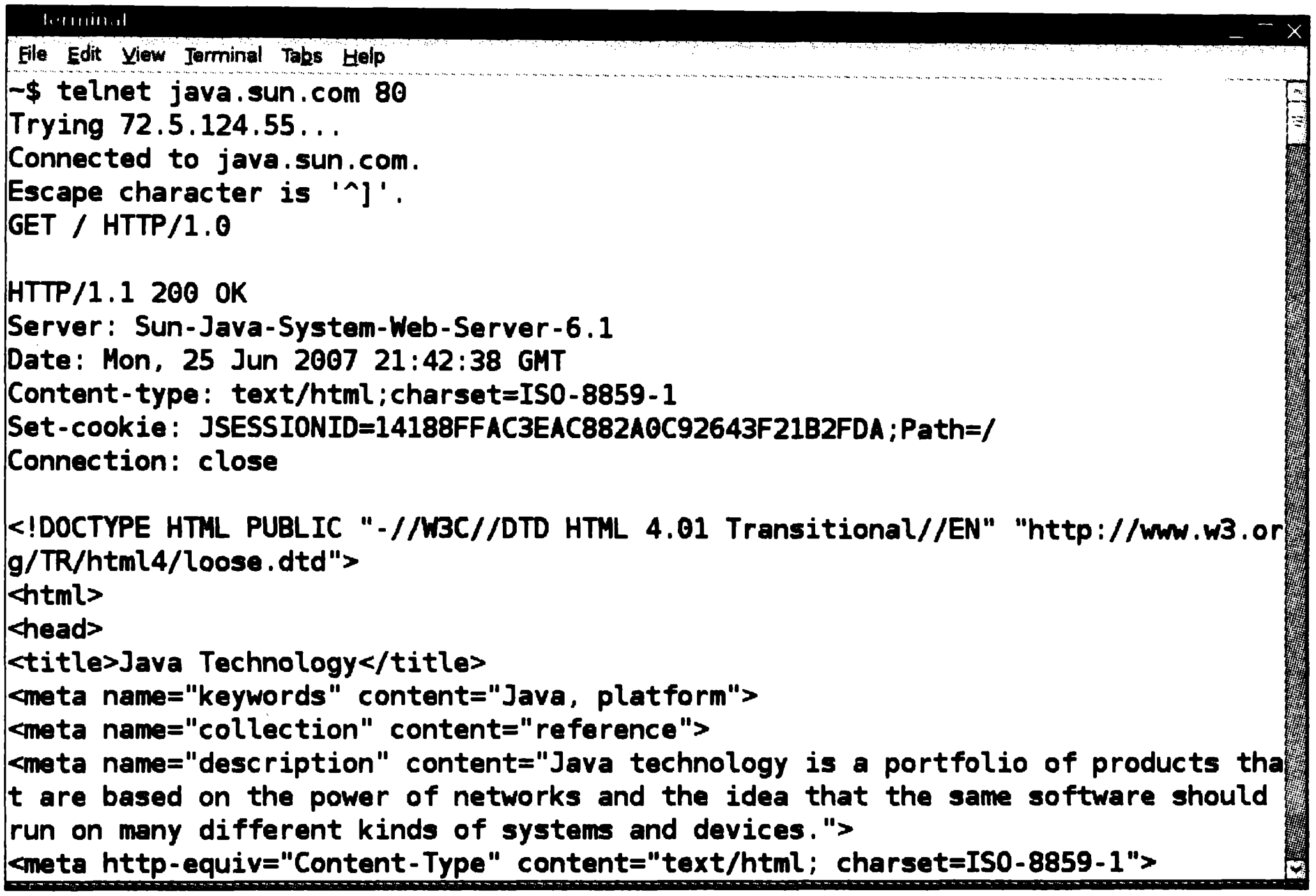


Рис. 3.3. Доступ к HTTP-порту с помощью утилиты `telnet`

НА ЗАМЕТКУ! Пару "ключ-значение" `Host: horstmann.com` требуется указывать для подключения к веб-серверу, на котором под одним и тем же IP-адресом размещаются разные домены. Ее можно не указывать, если на веб-сервере размещается единственный домен.

В первом примере сетевой программы, исходный код которой приведен в листинге 3.1, выполняются те же самые действия, что и при использовании утилиты `telnet`. Она устанавливает соединение с сервером через порт и выводит получаемые в ответ данные.

Листинг 3.1. Исходный код из файла `socket/SocketTest.java`

```
1  package socket;
2
3  import java.io.*;
4  import java.net.*;
5  import java.util.*;
```

```
6
7  /**
8   * В этой программе устанавливается сокетное соединение с атомными часами
9   * в г. Боулдере, шт. Колорадо и выводится время, передаваемое из сервера
10  *
11  * @version 1.20 2004-08-03
12  * @author Cay Horstmann
13 */
14 public class SocketTest
15 {
16   public static void main(String[] args) throws IOException
17   {
18     try (Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13))
19     {
20       InputStream inStream = s.getInputStream();
21       Scanner in = new Scanner(inStream);
22
23       while (in.hasNextLine())
24       {
25         String line = in.nextLine();
26         System.out.println(line);
27       }
28     }
29   }
30 }
```

В данной программе наибольший интерес представляют следующие две строки кода:

```
Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
InputStream inStream = s.getInputStream();
```

В первой строке кода открывается сокет. *Сокет* — это абстрактное понятие, обозначающее возможность для программ устанавливать соединения для обмена данными по сети. Конструктору сокета передается адрес удаленного сервера и номер порта. Если установить соединение не удается, генерируется исключение типа `UnknownHostException`, а при возникновении каких-нибудь других затруднений — исключение типа `IOException`. Класс `UnknownHostException` является подклассом, производным от класса `IOException`, поэтому в данном простом примере обрабатывается только исключение из суперкласса.

После открытия сокета метод `getInputStream()` из класса `java.net.Socket` возвращает объект типа `InputStream`, который можно использовать как любой другой поток ввода. Получив поток ввода, рассматриваемая здесь программа приступает к выводу каждой введенной символьной строки в стандартный поток вывода. Этот процесс продолжается до тех пор, пока не завершится поток ввода или не разорвется соединение с сервером.

Данная программа может взаимодействовать только с очень простыми серверами, например со службой учета текущего времени. В более сложных случаях клиент посылает серверу запрос на получение данных, а сервер может поддерживать установленное соединение в течение некоторого времени после отправки ответа на запрос. Примеры реализации подобного поведения представлены далее в этой главе.

Класс `Socket` очень удобен для работы в сети, поскольку он скрывает все сложности и подробности установления сетевого соединения и передачи данных по сети, реализуемые средствами библиотеки Java. А пакет `java.net`, по существу, предоставляет тот же самый программный интерфейс, который используется для работы с файлами.

НА ЗАМЕТКУ! В этой книге рассматривается только сетевой протокол TCP (Transmission Control Protocol — протокол управления передачей). На платформе Java поддерживается также протокол UDP (User Datagram Protocol — протокол пользовательских дейтаграмм), который может служить для отправки пакетов (называемых иначе *дейтаграммами*) с гораздо меньшими издержками, чем по протоколу TCP. Недостаток такого способа обмена данными по сети заключается в том, что пакеты необязательно доставлять получателю в последовательном порядке, и они вообще могут быть потеряны. Получатель сам должен позаботиться о том, чтобы пакеты были организованы в определенном порядке, а кроме того, он должен сам запрашивать повторно передачу отсутствующих пакетов. Протокол UDP хорошо подходит для тех приложений, которые могут обходиться без отсутствующих пакетов, например, для организации аудио- и видеопотоков или продолжительных измерений.

java.net.Socket 1.0

- `Socket(String host, int port)`

  Создает сокет для соединения с указанным хостом или портом.

- `InputStream getInputStream()`
- `OutputStream getOutputStream()`

  Получают поток ввода для чтения данных из сокета или записи данных в сокет.

Время ожидания для сокетов

Чтение данных из сокета продолжается до тех пор, пока данные доступны. Если хост (т.е. сетевой узел) недоступен, прикладная программа будет ожидать очень долго, и все будет зависеть от того, когда операционная система, под управлением которой работает компьютер, определит момент завершения периода ожидания.

Для конкретной прикладной программы можно самостоятельно определить наиболее подходящую величину времени ожидания для сокета, а затем вызвать метод `setSoTimeout()`, чтобы установить эту величину в миллисекундах. В приведенном ниже фрагменте когда показано, как это делается.

```
Socket s = new Socket(. . .);
s.setSoTimeout(10000); // истечение времени ожидания через 10 секунд
```

Если величина времени ожидания была задана для сокета, то при выполнении всех последующих операций чтения и записи данных будет генерироваться исключение типа `SocketTimeoutException` по истечении времени ожидания до фактического завершения текущей операции. Но это исключение можно перехватить, чтобы отреагировать на данное событие надлежащим образом, как показано ниже.

```
try
{
  InputStream in = s.getInputStream(); // читать данные из потока ввода in
```

```
   . . .
}
catch (InterruptedIOException exception)
{
   отреагировать на истечение времени ожидания
}
```

Что касается времени ожидания для сокетов, то остается еще одно затруднение, которое придется каким-то образом разрешить. Так, приведенный ниже конструктор может установить блокировку в течение неопределенного периода времени до тех пор, пока не будет установлено первоначальное соединение с хостом.

```
Socket(String host, int port)
```

Это затруднение можно преодолеть, если сначала создать несоединяемый сокет, а затем соединиться с заданным временем ожидания:

```
Socket s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

Если же требуется предоставить пользователям возможность прерывать соединение с сокетом в любой момент, то ниже, в разделе "Прерываемые сокеты" поясняется, как этого добиться.

java.net.Socket 1.0

- `Socket()` **1.1**

  Создает сокет, который еще не соединен в данный момент времени.

- `void connect(SocketAddress` *address*`)` **1.4**

  Соединяет данный сокет по указанному адресу.

- `void connect(SocketAddress` *address*`, int` *timeoutInMilliseconds*`)` **1.4**

  Соединяет данный сокет по указанному адресу или осуществляет возврат, если заданный промежуток времени истек.

- `void setSoTimeout(int timeoutInMilliseconds)` **1.1**

  Задает время блокировки для чтения *запросов* в данном сокете. По истечении времени блокировки возникает исключение типа `InterruptedIOException`.

- `boolean isConnected()` **1.4**

  Возвращает логическое значение `true`, если установлено соединение с сокетом.

- `boolean isClosed()` **1.4**

  Возвращает логическое значение `true`, если разорвано соединение с сокетом.

Межсетевые адреса

Как правило, нет особой нужды беспокоиться о межсетевых адресах в Интернете — числовых адресах хостов, состоящих из четырех байт (или из шестнадцати — по протоколу IPv6), как, например, **132.163.4.102**. Тем не менее, если требуется выполнить взаимное преобразование имен хостов и межсетевых адресов, то для этой цели можно воспользоваться классом `InetAddress`.

В пакете `java.net` поддерживаются межсетевые адреса по протоколу IPv6, при условии, что их поддержка обеспечивается и со стороны операционной системы хоста. В частности, статический метод `getByName()` возвращает объект типа `InetAddress` для хоста, как показано в приведенной ниже строке кода. Например, в следующей строке кода возвращается объект типа `InetAddress`, инкапсулирующий последовательность из четырех байт **132.163.4.104**:

```
InetAddress address = InetAddress.getByName("time-A.timefreq.bldrdoc.gov");
```

Получить доступ к этим байтам можно с помощью метода `getAddress()`:

```
byte[] addressBytes = address.getAddress();
```

Имена некоторых хостов с большим объемом трафика соответствуют нескольким межсетевым адресам, что объясняется попыткой сбалансировать нагрузку. Так, на момент написания данной книги имя хоста `google.com` соответствовало двенадцати различным сетевым адресам. Один из них выбирается случайным образом во время доступа к хосту. Получить межсетевые адреса всех хостов можно, вызвав метод `getAllByName()`:

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

И, наконец, иногда требуется адрес локального хоста. Если вы просто запросите адрес локального хоста, указав `localhost`, то неизменно получите в ответ локальный петлевой адрес **127.0.0.1**, которым другие не смогут воспользоваться для подключения к вашему компьютеру. Вместо этого вызовите метод `getLocalHost()`, чтобы получить адрес вашего локального хоста, как показано ниже.

```
InetAddress address = InetAddress.getLocalHost();
```

В листинге 3.2 приведен пример простой программы, выводящей межсетевой адрес локального хоста, если не указать дополнительные параметры в командной строке, или же все межсетевые адреса другого хоста, если указать имя хоста в командной строке, как в следующем примере:

```
java inetAddress/InetAddressTest www.horstmann.com
```

Листинг 3.2. Исходный код из файла `inetAddress/InetAddressTest.java`

```
1  package inetAddress;
2
3  import java.io.*;
4  import java.net.*;
5  /**
6   * В этой программе демонстрируется применение класса InetAddress.
7   * В качестве аргумента в командной строке следует указать имя хоста
8   * или же запустить программу без аргументов, чтобы получить в ответ
9   * адрес локального хоста
10  * @version 1.02 2012-06-05
11  * @author Cay Horstmann
12 */
13 public class InetAddressTest
14 {
15   public static void main(String[] args) throws IOException
16   {
17     if (args.length > 0)
18     {
```

```
19        String host = args[0];
20        InetAddress[] addresses = InetAddress.getAllByName(host);
21        for (InetAddress a : addresses)
22          System.out.println(a);
23      }
24      else
25      {
26        InetAddress localHostAddress = InetAddress.getLocalHost();
27        System.out.println(localHostAddress);
28      }
29    }
30 }
```

java.net.InetAddress 1.0

- `static InetAddress getByName(String host)`
- `static InetAddress[] getAllByName(String host)`

  Конструируют объект типа `InetAddress` или массив всех межсетевых адресов для заданного имени хоста.

- `static InetAddress getLocalHost()`

  Конструирует объект типа `InetAddress` для локального хоста.

- `byte[] getAddress()`

  Возвращает массив байтов, содержащий числовой адрес.

- `String getHostAddress()`

  Возвращает адрес хоста в виде символьной строки с десятичными числами, разделенными точками, например `"132.163.4.102"`.

- `String getHostName()`

  Возвращает имя хоста.

Реализация серверов

Итак, рассмотрев особенности реализации элементарного сетевого клиента, способного получать данные из сети, перейдем к обсуждению реализации простого сервера, способного посылать данные. После запуска серверная программа переходит в режим ожидания от клиентов подключения к портам сервера. Для рассматриваемого здесь примера выбран номер порта **8189**, который не используется ни одним из стандартных устройств. В следующей строке кода создается сервер с контролируемым портом **8189**:

```
ServerSocket s = new ServerSocket(8189);
```

А в приведенной ниже строке кода серверной программе предписывается ожидать подключения клиентов к заданному порту.

```
Socket incoming = s.accept();
```

Как только какой-нибудь клиент подключится к данному порту, отправив по сети запрос на сервер, данный метод возвратит объект типа `Socket`,

представляющий установленное соединение. Этот объект можно использовать для чтения и записи данных в потоки ввода-вывода, как показано в приведенном ниже фрагменте кода.

```
InputStream inStream = incoming.getInputStream();
OutputStream outStream = incoming.getOutputStream();
```

Все данные, направляемые в поток вывода серверной программы, поступают в поток ввода клиентской программы. А все данные, направляемые в поток вывода из клиентской программы, поступают в поток ввода серверной программы. Во всех примерах, приведенных в этой главе, обмен текстовыми данными осуществляется через сокеты. Поэтому соответствующие потоки ввода-вывода через сокет преобразуются в сканирующие и записывающие потоки типа `Scanner` и `Writer` следующим образом:

```
Scanner in = new Scanner(inStream);
PrintWriter out = new PrintWriter(outStream, true); // автоматическая очистка
```

Допустим, клиентская программа посылает следующее приветствие:

```
out.println("Hello! Enter BYE to exit.");
```

Если для подключения к серверной программе через порт **8189** используется утилита `telnet`, это приветствие отображается на экране терминала.

В рассматриваемой здесь простой серверной программе вводимые данные, отправленные клиентской программой, считываются построчно и посылаются обратно клиентской программе в режиме эхопередачи, как показано в приведенном ниже фрагменте кода. Этим наглядно демонстрируется получение данных от клиентской программы. А настоящая серверная программа должна обработать полученные данные и выдать соответствующий ответ.

```
String line = in.nextLine();
out.println("Echo: " + line);
if (line.trim().equals("BYE")) done = true;
```

По завершении сеанса связи открытый сокет закрывается следующим образом:

```
incoming.close();
```

Вот, собственно, и все, что делает данная программа. Любая серверная программа, например, веб-сервер, работающий по протоколу HTTP, выполняет аналогичный цикл следующих действий.

1. Получение из потока ввода входящих данных запроса на конкретную информацию от клиентской программы.
2. Расшифровка клиентского запроса.
3. Сбор информации, запрашиваемой клиентом.
4. Передача обнаруженной информации клиентской программе через поток вывода исходящих данных.

В листинге 3.3 приведен весь исходный код описанного выше примера серверной программы.

Листинг 3.3. Исходный код из файла `server/EchoServer.java`

```
1  package server;
2
3  import java.io.*;
4  import java.net.*;
5  import java.util.*;
6  /**
7   * В этой программе реализуется простой сервер, прослушивающий порт
8   * 8189 и посылающий обратно клиенту все полученные от него данные
9   * @version 1.21 2012-05-19
10 * @author Cay Horstmann
11 */
12 public class EchoServer
13 {
14   public static void main(String[] args) throws IOException
15   {
16     // установить сокет на стороне сервера
17     try (ServerSocket s = new ServerSocket(8189))
18     {
19       // ожидать подключения клиента
20       try (Socket incoming = s.accept())
21       {
22         InputStream inStream = incoming.getInputStream();
23         OutputStream outStream = incoming.getOutputStream();
24
25         try (Scanner in = new Scanner(inStream))
26         {
27           PrintWriter out = new PrintWriter(outStream, true);
28                                       // автоматическая очистка
29           out.println("Hello! Enter BYE to exit.");
30
31           // передать обратно данные, полученные от клиента
32           boolean done = false;
33           while (!done && in.hasNextLine())
34           {
35             String line = in.nextLine();
36             out.println("Echo: " + line);
37             if (line.trim().equals("BYE")) done = true;
38           }
39         }
40       }
41     }
42   }
43 }
```

Для проверки работоспособности данной серверной программы ее нужно скомпилировать и запустить. Затем необходимо подключиться с помощью утилиты `telnet` к локальному серверу `localhost` (или по IP-адресу **127.0.0.1**) через порт **8189**. Если ваш компьютер непосредственно подключен к Интернету, любой пользователь может получить доступ к данной серверной программе, если ему известен IP-адрес и номер порта. При подключении через этот порт будет получено следующее сообщение, как показано на рис. 3.4:

```
Hello! Enter BYE to exit.
(Привет! Введите BYE (Пока), чтобы выйти из программы)
```

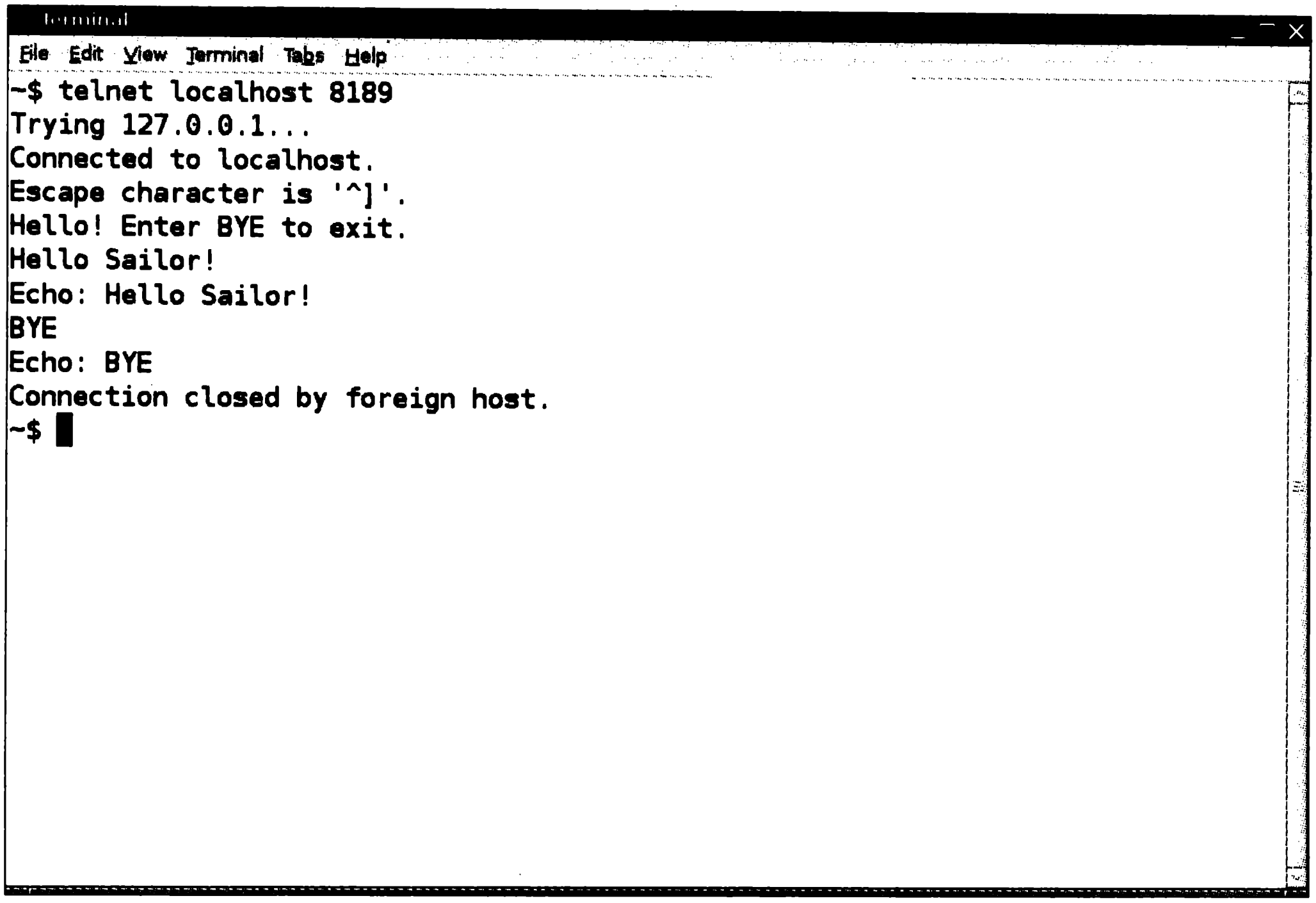


Рис. 3.4. Сеанс связи с сервером, передающим обратно данные, полученные от клиента

Введите любую фразу и понаблюдайте за тем, как она будет получена обратно в том же самом виде. Для отключения от сервера введите **BYE** (все символы в верхнем регистре). В итоге завершится и серверная программа.

java.net.ServerSocket 1.0

- `ServerSocket(int port)`
- Создает сокет на стороне сервера, контролирующего указанный порт.
- `Socket accept()`
- Ожидает соединения. Этот метод блокирует (т.е. переводит в режим ожидания) текущий поток до тех пор, пока не будет установлено соединение. Возвращает объект типа `Socket`, через который программа может взаимодействовать с подключаемым клиентом.
- `void close()`
- Закрывает сокет на стороне сервера.

Обслуживание многих клиентов

В предыдущем простом примере серверной программы не предусмотрена возможность одновременного подключения сразу нескольких клиентских программ. Обычно серверная программа работает на компьютере сервера, а клиентские программы могут одновременно подключаться к ней через Интернет из любой точки мира. Если на сервере не предусмотрена обработка одновременных запросов от

многих клиентов, это может привести к тому, что один клиент может монополизировать доступ к серверной программе в течение длительного времени. Во избежание подобных ситуаций следует прибегнуть к помощи потоков исполнения.

Всякий раз, когда серверная программа устанавливает новое сокетное соединение, т.е. в результате вызова метода `accept()` возвращается сокет, запускается новый поток исполнения для подключения *данного* клиента к серверу. После этого происходит возврат в основную программу, которая переходит в режим ожидания следующего соединения. Для того чтобы все это произошло, в серверной программе следует организовать приведенный ниже основной цикл.

```
while (true)
{
  Socket incoming = s.accept();
  Runnable r = new ThreadedEchoHandler(incoming);

  Thread t = new Thread(r);
  t.start();
}
```

Класс `ThreadedEchoHandler` реализует интерфейс `Runnable` и в своем методе `run()` поддерживает взаимодействие с клиентской программой:

```
class ThreadedEchoHandler implements Runnable
{
  . . .
  public void run()
  {
    try
    {
      InputStream inStream = incoming.getInputStream();
      OutputStream outStream = incoming.getOutputStream();
      обработать полученный запрос и отправить ответ
      incoming.close();
    }
    catch(IOException e)
    {
       обработать исключение
    }
  }
}
```

Когда при каждом соединении запускается новый поток исполнения, несколько клиентских программ могут одновременно подключаться к серверу. Это нетрудно проверить, выполнив следующие действия.

1. Скомпилируйте и запустите на выполнение серверную программу, исходный код которой приведен в листинге 3.4.
2. Откройте несколько окон утилиты `telnet` (рис. 3.5).
3. Переходя из одного окна в другое, введите команды. В итоге каждое отдельное окно утилиты `telnet` будет независимо от других взаимодействовать с серверной программой.
4. Для того чтобы разорвать соединение и закрыть окно утилиты `telnet`, нажмите комбинацию клавиш <Ctrl+C>.

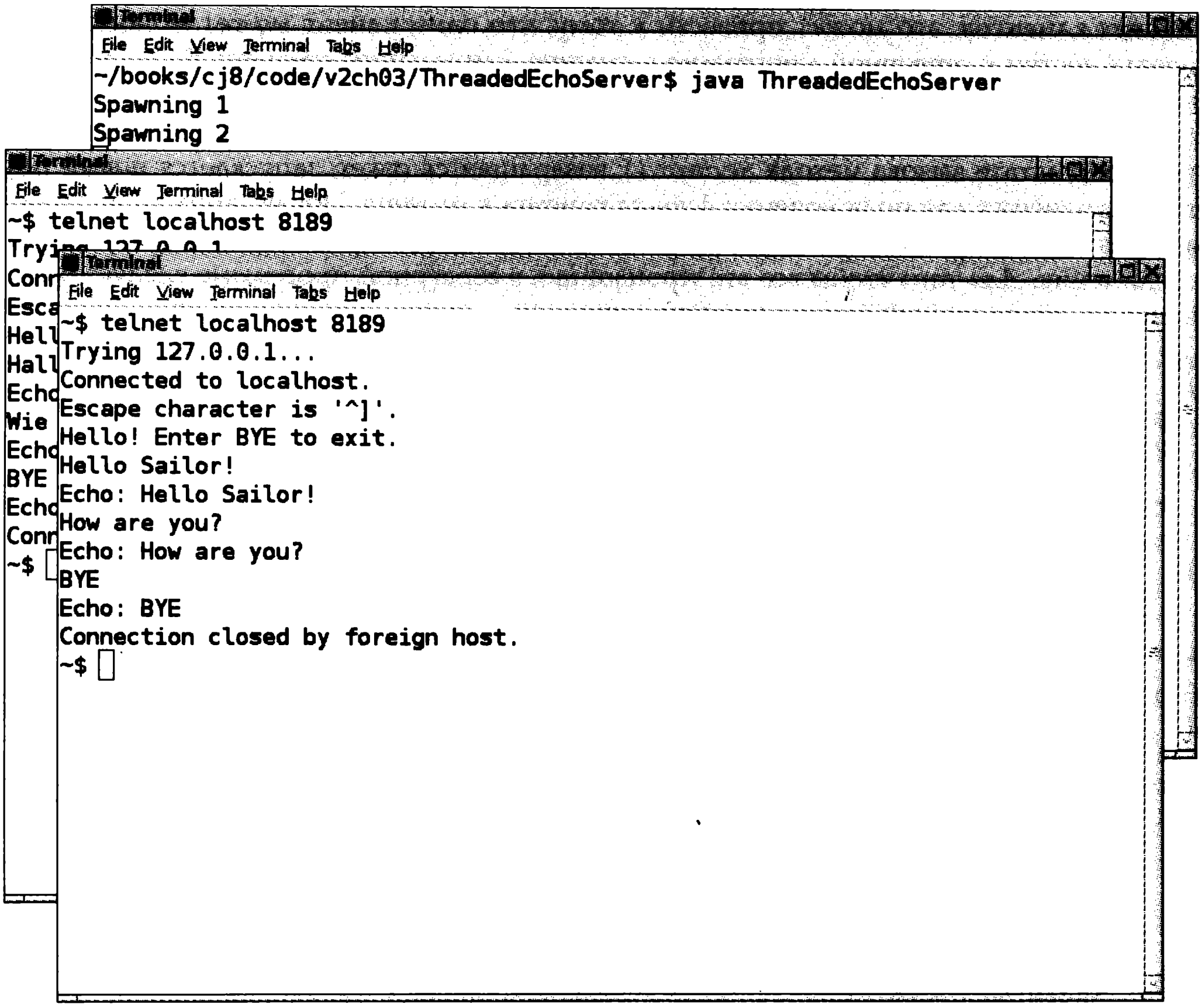


Рис. 3.5. Сеанс одновременной связи нескольких клиентов с сервером

НА ЗАМЕТКУ! В рассматриваемой здесь программе для каждого соединения порождается отдельный поток исполнения. Такой прием не вполне подходит для высокопроизводительного сервера. Более эффективной работы сервера можно добиться, используя средства из пакета `java.nio`. Дополнительные сведения по данному вопросу можно получить, обратившись по адресу https://www.ibm.com/developerworks/java/library/j-javaio/.

Листинг 3.4. Исходный код из файла `threaded/ThreadedEchoServer.java`

```
1  package threaded;
2
3  import java.io.*;
4  import java.net.*;
5  import java.util.*;
6  /**
7   * В этой программе реализуется многопоточный сервер, прослушивающий
8   * порт 8189 и передающий обратно все данные, полученные от клиентов
9   * @author Cay Horstmann
10  * @version 1.21 2012-06-04
11 */
```

```
12 public class ThreadedEchoServer
13 {
14   public static void main(String[] args )
15   {
16     try
17     {
18       int i = 1;
19       ServerSocket s = new ServerSocket(8189);
20       while (true)
21       {
22         Socket incoming = s.accept();
23         System.out.println("Spawning " + i);
24         Runnable r = new ThreadedEchoHandler(incoming);
25         Thread t = new Thread(r);
26         t.start();
27         i++;
28       }
29     }
30     catch (IOException e)
31     {
32       e.printStackTrace();
33     }
34   }
35 }
36
37 /**
38  * Этот класс обрабатывает данные, получаемые сервером от
39  * клиента через одно сокетное соединение
40 */
42 class ThreadedEchoHandler implements Runnable
43 {
44   private Socket incoming;
45
46   /**
47    * Конструирует обработчик
48    * @param i Входящий сокет
49   */
50   public ThreadedEchoHandler(Socket i)
51   {
52     incoming = i;
53   }
54
55   public void run()
56   {
57     try
58     {
59       try
60       {
61         InputStream inStream = incoming.getInputStream();
62         OutputStream outStream = incoming.getOutputStream();
63
64         Scanner in = new Scanner(inStream);
65         PrintWriter out = new PrintWriter(outStream, true);
66                                         // автоматическая очистка
67         out.println( "Hello! Enter BYE to exit." );
68         // передать обратно данные, полученные от клиента
69         boolean done = false;
70         while (!done && in.hasNextLine())
71         {
```

```
72            String line = in.nextLine();
73            out.println("Echo: " + line);
74            if (line.trim().equals("BYE"))
75              done = true;
76          }
77        }
78        finally
79        {
80          incoming.close();
81        }
82      }
83      catch (IOException e)
84      {
85        e.printStackTrace();
86      }
87    }
88 }
```

Полузакрытие

Полузакрытие обеспечивает возможность прервать передачу данных на одной стороне сокетного соединения, продолжая в то же время прием данных от другой стороны. Рассмотрим типичную ситуацию. Допустим, данные направляются на сервер, но заранее неизвестно, какой именно объем данных требуется передать. Если речь идет о файле, то его закрытие, по существу, означает завершение передачи данных. Если же закрыть сокет, то соединение с сервером будет немедленно разорвано.

Преодолеть подобное затруднение призвано полузакрытие. Если закрыть поток вывода через сокет, то для сервера это будет означать завершение передачи данных запроса. При этом поток ввода остается открытым, позволяя получить ответ от сервера. Код, реализующий механизм полузакрытия на стороне клиента, приведен ниже.

```
Socket socket = new Socket(host, port);
Scanner in = new Scanner(socket.getInputStream());
PrintWriter writer = new PrintWriter(socket.getOutputStream());
// передать данные запроса
writer.print(. . .);
writer.flush();
socket.shutdownOutput();
// теперь сокет полузакрыт
// принять данные ответа
while (in.hasNextLine() != null) { String line = in.nextLine(); . . . }
socket.close();
```

Серверная программа просто читает данные из потока ввода до тех пор, пока не закроется поток вывода на другом конце соединения. Очевидно, что такой подход применим только для служб однократного действия по протоколам, подобным HTTP, где клиент устанавливает соединение с сервером, передает запрос, получает ответ, после чего соединение разрывается.

java.net.Socket 1.0

- `void shutdownOutput()` **1.3**

  Устанавливает поток вывода в состояние завершения.

- `void shutdownInput()` **1.3**

  Устанавливает поток ввода в состояние завершения.

- `boolean isOutputShutdown()` **1.4**

  Возвращает логическое значение `true`, если вывод данных был остановлен.

- `boolean isInputShutdown()` **1.4**

  Возвращает логическое значение `true`, если ввод данных был остановлен.

Прерываемые сокеты

При подключении через сокет текущий поток исполнения блокируется до тех пор, пока соединение не будет установлено, или же до истечения времени ожидания. Аналогично, если пытаться передать или принять данные через сокет, текущий поток приостановит свое исполнение до успешного завершения операции или до истечения времени ожидания.

В прикладных программах, работающих в диалоговом режиме, желательно предоставить пользователям возможность прервать слишком затянувшийся процесс установления соединения через сокет. Но если поток исполнения блокирован для нереагирующего сокета, то разблокировать его не удастся, вызвав метод `interrupt()`.

Для прерывания сокетных операций служит класс `SocketChannel`, предоставляемый в пакете `java.nio`. Объект типа `SocketChannel` создается следующим образом:

```
SocketChannel channel = SocketChannel.open(new InetSocketAddress(host,
port));
```

У канала отсутствуют связанные с ним потоки ввода-вывода. Вместо этого в канале предоставляются методы `read()` и `write()`, использующие объекты типа `Buffer`. (Подробнее о буферах см. в главе 1.) Эти методы объявляются в интерфейсах `ReadableByteChannel` и `WritableByteChannel`. Если же нет желания иметь дело с буферами, для чтения из канала типа `SocketChannel` можно воспользоваться объектом типа `Scanner`. Для этой цели в классе `Scanner` предусмотрен следующий конструктор с параметром типа `ReadableByteChannel`:

```
Scanner in = new Scanner(channel);
```

Для того чтобы превратить канал в поток вывода, применяется статический метод `Channels.newOutputStream()`:

```
OutputStream outStream = Channels.newOutputStream(channel);
```

Вот, собственно, и все, что нужно сделать для прерывания сокетной операции. Если же поток исполнения будет прерван в процессе установления соединения, чтения или записи, соответствующая операция завершится и сгенерирует исключение.

В примере программы, исходный код которой приведен в листинге 3.5, демонстрируется применение прерываемых и блокирующих сокетов. Сервер передает числовые данные, имитируя прерывание их передачи после десятого числа. Если щелкнуть на любой кнопке, запустится поток исполнения, устанавливающий соединение с сервером и выводящий на экран передаваемые данные. В первом потоке исполнения используется прерываемый сокет, а во втором — блокирующий. Если щелкнуть

на кнопке Cancel (Отмена) во время вывода первых десяти чисел, то прервется исполнение любого из двух потоков.

А если щелкнуть на кнопке Cancel после передачи первых десяти чисел, то прервется исполнение только первого потока. Блокировка второго потока исполнения будет продолжаться до тех пор, пока сервер не разорвет окончательно соединение (рис. 3.6).

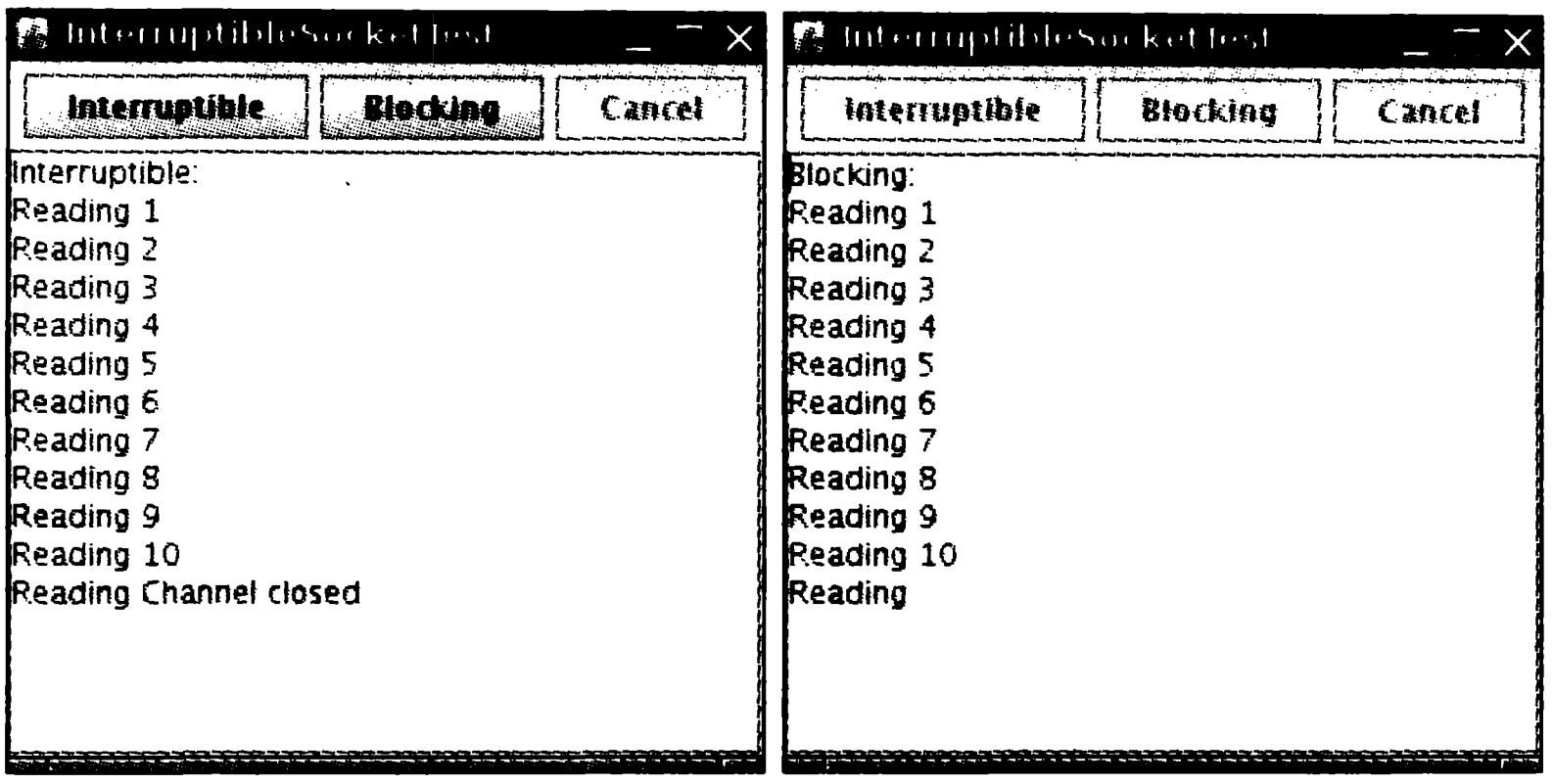


Рис. 3.6. Прерывание сокета

Листинг 3.5. Исходный код из файла `interruptible/InterruptibleSocketTest.java`

```
1   package interruptible;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.util.*;
6   import java.net.*;
7   import java.io.*;
8   import java.nio.channels.*;
9   import javax.swing.*;
10
11  /**
12   * В этой программе демонстрируется прерывание сокета через канал
13   * @author Cay Horstmann
14   * @version 1.03 2012-06-04
15  */
16  public class InterruptibleSocketTest
17  {
18    public static void main(String[] args)
19    {
20    EventQueue.invokeLater(new Runnable()
21      {
22        public void run()
23        {
24          JFrame frame = new InterruptibleSocketFrame();
25          frame.setTitle("InterruptibleSocketTest");
26          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27          frame.setVisible(true);
28        }
```

```
29        });
30     }
31  }
32
33  class InterruptibleSocketFrame extends JFrame
34  {
35     public static final int TEXT_ROWS = 20;
36     public static final int TEXT_COLUMNS = 60;
37
38     private Scanner in;
39     private JButton interruptibleButton;
40     private JButton blockingButton;
41     private JButton cancelButton;
42     private JTextArea messages;
43     private TestServer server;
44     private Thread connectThread;
45
46     public InterruptibleSocketFrame()
47     {
48        JPanel northPanel = new JPanel();
49        add(northPanel, BorderLayout.NORTH);
50
51        messages = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
52        add(new JScrollPane(messages));
53
54        interruptibleButton = new JButton("Interruptible");
55        blockingButton = new JButton("Blocking");
56        northPanel.add(interruptibleButton);
57        northPanel.add(blockingButton);
58
59        interruptibleButton.addActionListener(new ActionListener()
60           {
61              public void actionPerformed(ActionEvent event)
62              {
63                 interruptibleButton.setEnabled(false);
64                 blockingButton.setEnabled(false);
65                 cancelButton.setEnabled(true);
66                 connectThread = new Thread(new Runnable()
67                    {
68                       public void run()
69                       {
70                          try
71                          {
72                             connectInterruptibly();
73                          }
74                          catch (IOException e)
75                          {
76                             messages.append(
77                "\nInterruptibleSocketTest.connectInterruptibly: " + e);
78                          }
79                       }
80                    });
81                 connectThread.start();
82              }
83           });
84
85        blockingButton.addActionListener(new ActionListener()
86           {
```

```
87          public void actionPerformed(ActionEvent event)
88          {
89            interruptibleButton.setEnabled(false);
90            blockingButton.setEnabled(false);
91            cancelButton.setEnabled(true);
92            connectThread = new Thread(new Runnable()
93              {
94                public void run()
95                {
96                  try
97                  {
98                    connectBlocking();
99                  }
100                 catch (IOException e)
101                 {
102                   messages.append(
103                      "\nInterruptibleSocketTest.connectBlocking: " + e);
104                 }
105               }
106             });
107           connectThread.start();
108         }
109       });
110
111     cancelButton = new JButton("Cancel");
112     cancelButton.setEnabled(false);
113     northPanel.add(cancelButton);
114     cancelButton.addActionListener(new ActionListener()
115       {
116         public void actionPerformed(ActionEvent event)
117         {
118           connectThread.interrupt();
119           cancelButton.setEnabled(false);
120         }
121       });
122     server = new TestServer();
123     new Thread(server).start();
124     pack();
125   }
126
127   /**
128    * Соединяет с проверяемым сервером, используя прерываемый ввод-вывод
129   */
130   public void connectInterruptibly() throws IOException
131   {
132     messages.append("Interruptible:\n");
133     try (SocketChannel channel =
134         SocketChannel.open(new InetSocketAddress("localhost", 8189)))
135     {
136       in = new Scanner(channel);
137       while (!Thread.currentThread().isInterrupted())
138       {
139         messages.append("Reading ");
140         if (in.hasNextLine())
141         {
142           String line = in.nextLine();
143           messages.append(line);
144           messages.append("\n");
```

```
145           }
146         }
147       }
148       finally
149       {
150         EventQueue.invokeLater(new Runnable()
151           {
152             public void run()
153             {
154               messages.append("Channel closed\n");
155               interruptibleButton.setEnabled(true);
156               blockingButton.setEnabled(true);
157             }
158           });
159       }
160     }
161
162     /**
163      * Соединяет с проверяемым сервером, используя блокирующий ввод-вывод
164     */
165     public void connectBlocking() throws IOException
166     {
167       messages.append("Blocking:\n");
168       try (Socket sock = new Socket("localhost", 8189))
169       {
170         in = new Scanner(sock.getInputStream());
171         while (!Thread.currentThread().isInterrupted())
172         {
173           messages.append("Reading ");
174           if (in.hasNextLine())
175           {
176             String line = in.nextLine();
177             messages.append(line);
178             messages.append("\n");
179           }
180         }
181       }
182       finally
183       {
184         EventQueue.invokeLater(new Runnable()
185           {
186             public void run()
187             {
188               messages.append("Socket closed\n");
189               interruptibleButton.setEnabled(true);
190               blockingButton.setEnabled(true);
191             }
192           });
193       }
194     }
195
196     /**
197      * Многопоточный сервер, прослушивающий порт 8189 и посылающий
198      * клиентам числа, имитируя зависание после передачи 10 чисел
199     */
200     class TestServer implements Runnable
201     {
202       public void run()
```

```
203     {
204       try
205       {
206         ServerSocket s = new ServerSocket(8189);
207
208         while (true)
209         {
210           Socket incoming = s.accept();
211           Runnable r = new TestServerHandler(incoming);
212           Thread t = new Thread(r);
213           t.start();
214         }
215       }
216       catch (IOException e)
217       {
218         messages.append("\nTestServer.run: " + e);
219       }
220     }
221   }
222
223   /**
224    * Этот класс обрабатывает данные, получаемые сервером
225    * от клиента через одно сокетное соединение
226   */
227   class TestServerHandler implements Runnable
228   {
229     private Socket incoming;
230     private int counter;
231
232     /**
233      * Конструирует обработчик
234      * @param i Входящий сокет
235     */
236     public TestServerHandler(Socket i)
237     {
238       incoming = i;
239     }
240
241     public void run()
242     {
243       try
244       {
245         try
246         {
247           OutputStream outStream = incoming.getOutputStream();
248           PrintWriter out = new PrintWriter(outStream, true);
249           while (counter < 100)
250           {
251             counter++;
252             if (counter <= 10) out.println(counter);
253             Thread.sleep(100);
254           }
255         }
256         finally
257         {
258           incoming.close();
259            messages.append("Closing server\n");
260         }
```

```
261         }
262         catch (Exception e)
263         {
264           messages.append("\nTestServerHandler.run: " + e);
265         }
266       }
267     }
268 }
```

java.net.InetSocketAddress 1.4

- `InetSocketAddress(String` *hostname*`, int` *port*`)`
- Создает объект адреса с указанными именем сетевого узла (хоста) и номером порта, преобразуя имя узла в адрес при установлении соединения. Если преобразовать имя сетевого узла в адрес не удается, устанавливается логическое значение `true` свойства `unresolved`.
- `boolean isUnresolved()`
- Возвращает логическое значение `true`, если для данного объекта не удается преобразовать имя сетевого узла в адрес.

java.nio.channels.SocketChannel 1.4

- `static SocketChannel open(SocketAddress` *address*`)`
- Открывает канал для сокета и связывает его с удаленным сетевым узлом по указанному адресу.

java.nio.channels.Channels 1.4

- `static InputStream newInputStream(ReadableByteChannel` *channel*`)`
- Создает поток ввода для чтения данных из указанного канала.
- `static OutputStream newOutputStream(WritableByteChannel` *channel*`)`
- Создает поток вывода для записи данных в указанный канал.

Получение данных из Интернета

Для того чтобы получить доступ к веб-серверам из программы на Java, требуется более высокий уровень сетевого взаимодействия, чем установление соединения через сокет и выдача HTTP-запросов. В последующих разделах будут рассмотрены классы, предоставляемые для этой цели в библиотеке Java.

URL и URI

Классы `URL` и `URLConnection` инкапсулируют большую часть внутреннего механизма извлечения данных с удаленного веб-сайта. Объект типа `URL` создается следующим образом:

```
URL url = new URL(символьная строка с URL);
```

Если требуется только извлечь содержимое из указанного ресурса, достаточно вызвать метод `openStream()` из класса `URL`. Этот метод возвращает объект типа `InputStream`. Поток ввода данного типа можно использовать обычным образом, например, создать объект типа `Scanner`:

```
InputStream inStream = url.openStream();
Scanner in = new Scanner(inStream);
```

В пакете `java.net` отчетливо различаются унифицированные *указатели* ресурсов (URL) и унифицированные *идентификаторы* ресурсов (URI). В частности, URI — это лишь синтаксическая конструкция, содержащая различные части символьной строки, обозначающей веб-ресурс. А URL — это особая разновидность идентификатора URI с исчерпывающими данными о местоположении ресурса. Имеются и такие URI, как, например, `mailto:cay@hortsmann.com`, которые не являются указателями ресурсов, потому что по ним нельзя обнаружить какие-нибудь данные. Такой URI называется унифицированным *именем* ресурса (URN).

В классе `URI` из библиотеки Java отсутствуют методы доступа к ресурсу по указанному идентификатору, поскольку этот класс предназначен только для синтаксического анализа символьной строки, обозначающей ресурс. В отличие от него, класс `URL` позволяет открыть поток ввода-вывода для данного ресурса. Поэтому в классе `URL` допускается взаимодействие только по тем протоколам и схемам, которые поддерживаются в библиотеке Java, в том числе `http:`, `https:` и `ftp:` — для Интернета, `file:` — для локальной файловой системы, а также `jar:` — для обращения к JAR-файлам.

Синтаксический анализ URI — непростая задача, поскольку идентификаторы ресурсов могут иметь сложную структуру. В качестве примера ниже приведены URI с замысловатой структурой.

```
http://maps.yahoo.com/py/maps.py?csz=Cupertino+CA
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

В обозначении URI задаются правила построения таких идентификаторов. Структура URI выглядит следующим образом:

```
[схема:]специальная_часть_схемы[#фрагмент]
```

где квадратные скобки (`[]`) обозначают необязательную часть, а двоеточие и знак `#` служат в качестве разделителей. Если *схема*: присутствует как составная часть в идентификаторе URI, то он называется *абсолютным*, а иначе — *относительным*. Абсолютный URI называется *непрозрачным*, если *специальная_часть_схемы* не начинается с косой черты (`/`), как, например, показано ниже.

```
mailto:cay@horstmann.com
```

Все абсолютные, прозрачные URI и все относительные URL имеют *иерархическую структуру*. Ниже приведены характерные тому примеры.

```
http://horstmann.com/index.html
../../java/net/Socket.html#Socket()
```

Составляющая *специальная_часть_схемы* иерархического URI имеет следующую структуру:

```
[//полномочия][путь][?запрос]
```

И здесь квадратные скобки (`[]`) обозначают необязательную часть. В URI серверов составляющая *полномочия* имеет приведенную ниже форму, где элемент *порт* должен иметь целочисленное значение.

```
[сведения_о_пользователе@]хост[:порт]
```

В документе RFC 2396, стандартизирующем идентификаторы URI, допускается также механизм указания составляющей *полномочия* в другом формате на основе данных из реестра. Но он не получил широкого распространения.

Одно из назначений класса `URI` состоит в синтаксическом анализе отдельных составляющих идентификатора. Они извлекаются с помощью перечисленных ниже методов.

```
getScheme()
getSchemeSpecificPart()
getAuthority()
getUserInfo()
getHost()
getPort()
getPath()
getQuery()
getFragment()
```

Другое назначение класса `URI` состоит в обработке абсолютных и относительных идентификаторов. Так, если имеются абсолютный и относительный идентификаторы URI:

```
http://docs.mycompany.com/api/java/net/ServerSocket.html
```

и

```
../../java/net/Socket.html#Socket()
```

их можно объединить в абсолютный URI следующим образом:

```
http://docs.mycompany.com/api/java/net/Socket.html#Socket()
```

Такой процесс называется *преобразованием* относительного URI. Обратный процесс называется *преобразованием абсолютных адресов в относительные*. Например, имея *базовый* URI:

```
http://docs.mycompany.com/api
```

можно преобразовать следующий абсолютный URI:

```
http://docs.company.com/api/java/lang/String.html
```

в приведенный ниже относительный URI.

```
java/lang/String.html
```

Для выполнения обоих видов преобразования в классе `URI` предусмотрены два соответствующих метода:

```
relative = base.relativize(combined);
combined = base.resolve(relative);
```

Извлечение данных средствами класса URLConnection

Для получения дополнительных сведений о веб-ресурсе следует воспользоваться классом `URLConnection`, предоставляющим намного больше средств управления доступом к веб-ресурсам, чем более простой класс `URL`. Для работы с объектом типа `URLConnection` необходимо тщательно спланировать и выполнить следующие действия.

1. Вызвать метод `openConnection()` из класса `URL` для получения объекта типа `URLConnection` следующим образом:

   ```
   URLConnection connection = url.openConnection();
   ```

2. Задать свойства запроса с помощью перечисленных ниже методов.

```
setDoInput()
setDoOutput()
setIfModifiedSince()
setUseCaches()
setAllowUserInteraction()
setRequestProperty()
setConnectTimeout()
setReadTimeout()
```

3. Эти методы будут подробно рассматриваться далее.
4. Установить соединение с удаленным ресурсом с помощью метода `connect()`:

```
connection.connect();
```

5. Помимо создания сокета, для установления соединения с веб-сервером этот метод запрашивает также у сервера *данные заголовка*.
6. После подключения к веб-серверу становятся доступными поля заголовка. Обращаться к ним можно с помощью универсальных методов `getHeaderFieldKey()` и `getHeaderField()`. Кроме того, для удобства разработки предусмотрены перечисленные ниже методы обработки стандартных полей запроса.

```
getContentType()
getContentLength()
getContentEncoding()
getDate()
getExpiration()
getLastModified()
```

7. И наконец, для доступа к данным указанного ресурса следует вызвать метод `getInputStream()`, предоставляющий поток ввода для чтения данных. (Это тот же самый поток ввода, который возвращается методом `openStream()` из класса `URL`.) Существует также метод `getContent()`, но он не такой удобный. Для обработки содержимого стандартных типов, например, текста (`text/plain`) или изображений (`image/gif`), придется воспользоваться классами из пакета `com.sun`. Кроме того, можно зарегистрировать собственные обработчики содержимого, но они в данной книге не рассматриваются.

НА ЗАМЕТКУ! Некоторые разработчики, пользующиеся классом `URLConnection`, ошибочно считают, что методы `getInputStream()` и `getOutpuStream()` аналогичны одноименным методам из класса `Socket`. Это не совсем так. Класс `URLConnection` способен выполнять много других функций, в том числе обрабатывать заголовки запросов и ответов. Поэтому рекомендуется строго придерживаться указанной выше последовательности действий.

Рассмотрим методы из класса `URLConnection` более подробно. В нем имеется ряд методов, задающих свойства соединения еще до подключения к веб-серверу. Наиболее важными среди них являются методы `setDoInput()` и `setDoOutput()`. По умолчанию при соединении предоставляется поток ввода для чтения данных с веб-сервера, но не поток вывода для записи данных. Для того чтобы получить поток вывода (например, с целью разместить данные на веб-сервере) нужно сделать следующий вызов:

```
connection.setDoOutput(true);
```

Далее можно установить ряд заголовков запроса и послать их веб-серверу в составе единого запроса. Ниже приведен пример заголовков запроса.

```
GET www.server.com/index.html HTTP/1.0
Referer: http://www.somewhere.com/links.html
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)
Host: www.server.com
Accept: text/html, image/gif, image/jpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: orangemilano=192218887821987
```

Метод `setIfModifiedSince()` служит для уведомления о том, что требуется получить только те данные, которые были изменены после определенной даты.

Методы `setUseCaches()` и `setAllowUserInteraction()` следует вызывать только в аплетах. В частности, метод `setUseCaches()` предписывает браузеру проверить сначала кеш. А метод `setAllowUserInteraction()` позволяет открыть в аплете диалоговое окно с предложением ввести имя и пароль пользователя для защиты доступа к сетевым ресурсам (рис. 3.7).

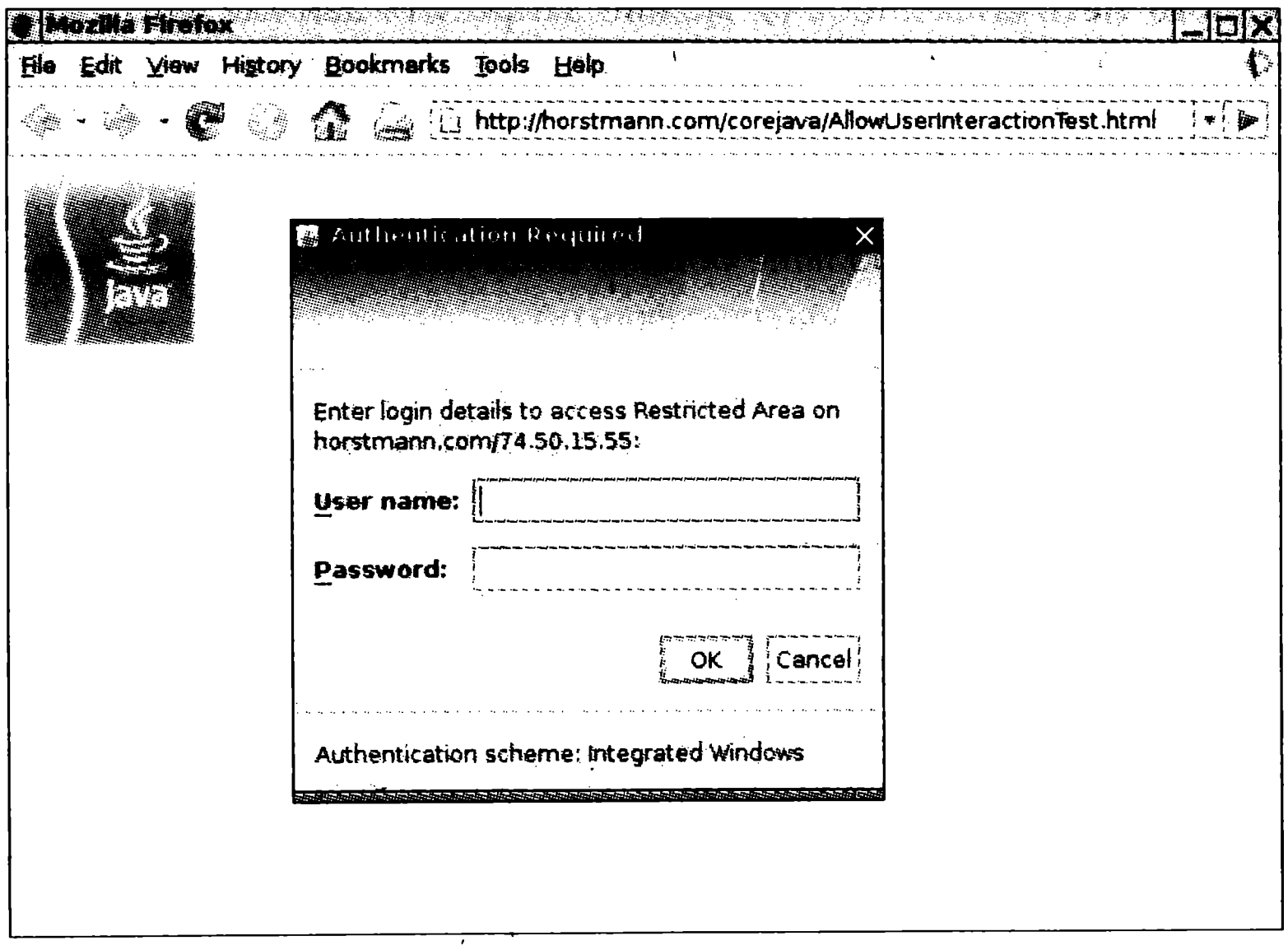


Рис. 3.7. Диалоговое окно для ввода имени и пароля доступа к сетевым ресурсам

И наконец, с помощью метода `setRequestProperty()` можно установить пару "имя–значение", имеющую определенный смысл для конкретного протокола. Формат заголовка запроса по протоколу HTTP описан в документе RFC 2616. Некоторые его параметры не очень хорошо документированы, поэтому за дополнительными разъяснениями зачастую приходится обращаться к другим программистам. Так, для доступа к защищенной паролем веб-странице необходимо выполнить следующие действия.

1. Составьте символьную строку из имени пользователя, двоеточия и пароля:

```
String input = username + ":" + password;
```

2. Перекодируйте полученную в итоге символьную строку по алгоритму кодирования Base64, как показано ниже. (Этот алгоритм преобразует последовательность байтов в последовательность символов в коде ASCII.)

```
String encoding = base64Encode(input);
```

3. Вызовите метод `setRequestProperty()` с именем свойства `"Authorization"` и значением `"Basic " + encoding`, как показано ниже.

```
connection.setRequestProperty("Authorization", "Basic " + encoding);
```

СОВЕТ. Здесь рассматривается способ обращения к защищенной паролем веб-странице. А для доступа к защищенному паролем FTP-файлу применяется совершенно другой подход. В этом случае достаточно сформировать URL следующего вида:

`ftp://`*`имя_пользователя`*`:`*`пароль`*`@ftp.`*`ваш_сервер`*`.com/pub/file.txt`

После вызова метода `connect()` можно запросить данные заголовка из ответа. Рассмотрим сначала способ перечисления всех полей заголовка. Создатели рассматриваемого здесь класса посчитали нужным создать собственный способ перебора полей. Так, при вызове приведенного ниже метода получается *n*-й ключ заголовка, причем нумерация начинается с единицы! В итоге возвращается пустое значение `null`, если *n* равно нулю или больше общего количества полей заголовка.

```
String key = connection.getHeaderFieldKey(n);
```

Но для определения количества полей не предусмотрено никакого другого метода. Чтобы перебрать все поля, приходится просто вызывать метод `getHeaderFieldKey()` до тех пор, пока не будет получено пустое значение `null`. Аналогично при вызове следующего метода возвращается значение из *n*-го поля:

```
String value = connection.getHeaderField(n);
```

Метод `getHeaderFields()` возвращает объект типа `Map` с полями заголовка, как показано ниже.

```
Map<String,List<String>> headerFields = connection.getHeaderFields();
```

В качестве примера ниже приведен ряд полей заголовка из типичного ответа на запрос по протоколу HTTP.

```
Date: Wed, 27 Aug 2008 00:15:48 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```

Для удобства разработки предусмотрены шесть методов, получающих значения из наиболее употребительных полей заголовка и приводящие эти значения к соответствующим числовым типам по мере необходимости. Все эти служебные методы перечислены в табл. 3.1. В методах, возвращающих значения типа `long`, отсчет количества возвращаемых секунд начинается с полуночи 1 января 1970 г.

Таблица 3.1. Служебные методы, получающие значения полей заголовка из ответа на запрос

Имя поля (ключа)	Имя метода	Возвращаемое значение
Date	getDate	long
Expires	getExpiration	long
Last-Modified	getLastModified	long
Content-Length	getContentLength	int
Content-Type	getContentType	String
Content-Encoding	getContentEncoding	String

В примере программы, исходный код которой приведен в листинге 3.6, предоставляется возможность поэкспериментировать с соединениями по URL. Запустив программу, вы можете указать в командной строке конкретный URL, имя пользователя и пароль:

```
java urlConnection.URLConnectionTest http://www.сервер.com пользователь пароль
```

В итоге программа выведет на экран следующее.

- Все ключи и значения из полей заголовка.
- Значения, возвращаемые шестью служебными методами доступа к наиболее употребительным полям заголовка, как показано в табл. 3.1.
- Первые 10 символьных строк из запрашиваемого ресурса.

Это довольно простая программа, за исключением той части, где выполняется кодирование по алгоритму Base64. Вместо используемого в данном примере класса можно применить недокументированный класс `sun.misc.BASE64Encoder`. Для этого достаточно заменить используемый вызов метода `base64Encode()` вызовом аналогичного метода, приведенного ниже. Но в данном примере мы не стали полагаться на недокументированные классы, использовав в ней свой класс.

```
String encoding = new sun.misc.BASE64Encoder().encode(input.getBytes());
```

НА ЗАМЕТКУ! Класс `javax.mail.internet.MimeUtility` из стандартного расширения JavaMail библиотеки Java также содержит метод для кодирования по алгоритму Base64. В состав JDK входит также класс `java.util.prefs.Base64`, который служит той же самой цели, но он не является открытым, и поэтому его нельзя использовать в прикладных программах.

Листинг 3.6. Исходный код из файла `urlConnection/URLConnectionTest.java`

```
1   package urlConnection;
2
3   import java.io.*;
4   import java.net.*;
5   import java.util.*;
6
7   /**
8    * В этой программе устанавливается соединение по заданному URL и
9    * отображаются данные заголовка из получаемого ответа, а также
10   * первые 10 строк запрашиваемых данных. Для этого в командной строке
11   * следует указать конкретный URL и дополнительно имя пользователя и
12   * пароль (для элементарной аутентификации по протоколу HTTP)
```

```
13   * @version 1.11 2007-06-26
14   * @author Cay Horstmann
15  */
16  public class URLConnectionTest
17  {
18    public static void main(String[] args)
19    {
20      try
21      {
22        String urlName;
23        if (args.length > 0) urlName = args[0];
24        else urlName = "http://horstmann.com";
25
26        URL url = new URL(urlName);
27        URLConnection connection = url.openConnection();
28
29        // установить имя пользователя и пароль, если они
30        // указаны в командной строке
31        if (args.length > 2)
32        {
33          String username = args[1];
34          String password = args[2];
35          String input = username + ":" + password;
36          String encoding = base64Encode(input);
37          connection.setRequestProperty(
38              "Authorization", "Basic " + encoding);
39        }
40
41        connection.connect();
42
43        // вывести поля заголовка
44        Map<String, List<String>> headers = connection.getHeaderFields();
45        for (Map.Entry<String, List<String>> entry : headers.entrySet())
46        {
47          String key = entry.getKey();
48          for (String value : entry.getValue())
49            System.out.println(key + ": " + value);
50        }
51
52        // вывести значения полей заголовка, используя служебные методы
53        System.out.println("----------");
54        System.out.println(
55            "getContentType: " + connection.getContentType());
56        System.out.println(
57            "getContentLength: " + connection.getContentLength());
58        System.out.println(
59            "getContentEncoding: " + connection.getContentEncoding());
60        System.out.println("getDate: " + connection.getDate());
61        System.out.println(
62            "getExpiration: " + connection.getExpiration());
63        System.out.println(
64            "getLastModifed: " + connection.getLastModified());
65        System.out.println("----------");
66
67        Scanner in = new Scanner(connection.getInputStream());
68
69        // вывести первые десять строк запрашиваемого содержимого
70        for (int n = 1; in.hasNextLine() && n <= 10; n++)
71          System.out.println(in.nextLine());
```

```
72          if (in.hasNextLine()) System.out.println(". . .");
73        }
74        catch (IOException e)
75        {
76          e.printStackTrace();
77        }
78      }
79
80      /**
81       * Кодирует символьную строку по алгоритму Base64
82       * @param s Символьная строка
83       * @return Возвращает символьную строку s,
85       *         закодированную по алгоритму Base64
86      */
87      public static String base64Encode(String s)
88      {
89        ByteArrayOutputStream bOut = new ByteArrayOutputStream();
90        Base64OutputStream out = new Base64OutputStream(bOut);
91        try
92        {
93          out.write(s.getBytes());
94          out.flush();
95        }
96        catch (IOException e)
97        {
98        }
99        return bOut.toString();
100     }
101 }
103 /**
104  * Этот потоковый фильтр преобразует поток вывода байтов в их
105  * кодировку по алгоритму Base64, где каждые три байта кодируются четырьмя
106  * символами. Каждые шесть битов в кодировке |11111122|22223333|33444444|
107  * кодируются по таблице toBase64. Если количество вводимых байтов не
108  * кратно трем, то последняя группа из четырех символов заполняется
109  * одним или двумя знаками равенства. А каждая выводимая строка состоит
110  * не больше, чем из 76 символов
111 */
112 class Base64OutputStream extends FilterOutputStream
113 {
114   private static char[] toBase64 =
115     { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
116       'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
117       'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
118       'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
119       'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7',
120       '8', '9', '+', '/' };
121
122   private int col = 0;
123   private int i = 0;
124   private int[] inbuf = new int[3];
125
126   /**
127    * Конструирует потоковый фильтр
128    * @param out Потоковый фильтр
129   */
130   public Base64OutputStream(OutputStream out)
131   {
132     super(out);
```

```
133   }
134
135   public void write(int c) throws IOException
136   {
137     inbuf[i] = c;
138     i++;
139     if (i == 3)
140     {
141       if (col >= 76)
142       {
143         super.write('\n');
144         col = 0;
145       }
146       super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
147       super.write(
148          toBase64[((inbuf[0] & 0x03) << 4) | ((inbuf[1] & 0xF0) >> 4)]);
149       super.write(
150          toBase64[((inbuf[1] & 0x0F) << 2) | ((inbuf[2] & 0xC0) >> 6)]);
151       super.write(toBase64[inbuf[2] & 0x3F]);
152       col += 4;
153       i = 0;
154     }
155   }
156   public void flush() throws IOException
157   {
158     if (i > 0 && col >= 76)
159     {
160       super.write('\n');
161       col = 0;
162     }
163     if (i == 1)
164     {
165       super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
166       super.write(toBase64[(inbuf[0] & 0x03) << 4]);
167       super.write('=');
168       super.write('=');
169     }
170     else if (i == 2)
171    {
172       super.write(toBase64[(inbuf[0] & 0xFC) >> 2]);
173       super.write(
174          toBase64[((inbuf[0] & 0x03) << 4) | ((inbuf[1] & 0xF0) >> 4)]);
175       super.write(toBase64[(inbuf[1] & 0x0F) << 2]);
176       super.write('=');
177     }
178   }
179 }
```

java.net.URL 1.0

- `InputStream openStream()`

  Открывает поток ввода для чтения данных из ресурса.

- `URLConnection openConnection();`

  Возвращает объект типа `URLConnection`, управляющий соединением с ресурсом.

java.net.URLConnection 1.0

- `void setDoInput(boolean` *doInput*`)`
- `boolean getDoInput()`

  Если параметр `doInput` принимает логическое значение `true`, пользователь может принимать вводимые данные из текущего объекта типа `URLConnection`.

- `void setDoOutput(boolean` *doOutput*`)`
- `boolean getDoOutput()`

  Если параметр *doOutput* принимает логическое значение `true`, пользователь может передавать выводимые данные в текущий объект типа `URLConnection`.

- `void setIfModifiedSince(long` *time*`)`
- `long getIfModifiedSince()`

  Свойство `ifModifiedSince` настраивает данный объект типа `URLConnection` на извлечение только тех данных, которые были изменены после указанного момента времени. Время задается в секундах, начиная с полуночи 1 января 1970 г. по Гринвичу.

- `void setUseCaches(boolean` *useCaches*`)`
- `boolean getUseCaches()`

  Если параметр `useCaches` принимает логическое значение `true`, данные можно извлечь из локального кеша. Следует, однако, иметь в виду, что кеш не поддерживается самим объектом типа `URLConnection`. Поэтому кеш должен быть предоставлен внешней программой, например, браузером.

- `void setAllowUserInteraction(boolean` *allowUsreInteraction*`)`
- `boolean getAllowUserInteraction()`

  Если параметр *allowUserInteraction* принимает логическое значение, у пользователя может запрашиваться пароль. Следует, однако, иметь в виду, что у самого объекта типа `URLConnection` отсутствуют средства, требующиеся для выполнения подобных запросов. Поэтому запрос пароля должен быть организован во внешней программе, например, в браузере или подключаемом модуле.

- `void setConnectTimeout(int` *timeout*`)` **5.0**
- `int getConnectTimeout()` **5.0**

  Устанавливают или возвращают величину времени ожидания (в миллисекундах) для соединения. Если время ожидания истечет до установления соединения, метод `connect()` из соответствующего потока ввода сгенерирует исключение типа `SocketTimeoutException`.

- `void setReadTimeout(int` *timeout*`)` **5.0**
- `int getReadTimeout()` **5.0**

  Устанавливают или возвращают величину времени ожидания (в миллисекундах) для чтения данных. Если время ожидания истечет до успешного завершения операции чтения, метод `read()` сгенерирует исключение типа `SocketTimeoutException`.

- `void setRequestProperty(String` *key*`, String` *value*`)`

  Устанавливает значение в поле заголовка.

- `Map<String,List<String>> getRequestProperties()` **1.4**

  Возвращает отображение со свойствами запроса. Все свойства одного и того же ключа вносятся в список.

- `void connect()`

  Устанавливает соединение с удаленным ресурсом и получает данные заголовка из ответа.

- `Map<String,List<String>> getHeaderFields()` **1.4**

  Возвращает отображение с полями заголовка из ответа. Все свойства одного и того же ключа вносятся в список.

- `String getHeaderFieldKey(int n)`

  Возвращает ключ *n*-го поля заголовка из ответа или пустое значение `null`, если *n* меньше или равно нулю или превышает количество полей.

- `String getHeaderField(int n)`

  Возвращает значение *n*-го поля заголовка из ответа или пустое значение `null`, если *n* меньше или равно нулю или превышает количество полей.

- `int getContentLength()`

  Возвращает длину доступного содержимого или **-1**, если длина неизвестна.

- `String getContentType()`

  Возвращает тип содержимого, например, `text/plain` или `image/gif`.

- `String getContentEncoding()`

  Возвращает кодировку содержимого, например `gzip`. Применяется редко, потому что используемая по умолчанию кодировка не всегда указывается в поле `identity` заголовка `Content-Encoding`.

- `long getDate()`
- `long getExpiration()`
- `long getLastModifed()`

  Возвращают время создания, последней модификации ресурса или время, когда срок действия ресурса истекает. Время указывается в секундах, начиная с 1 января 1970 г. по Гринвичу.

- `InputStream getInputStream()`
- `OutputStream getOutputStream()`

  Возвращают поток ввода для чтения данных из ресурса или вывода для записи данных в ресурс.

- `Object getContent()`

  Выбирает подходящий обработчик содержимого для чтения данных из ресурса. Этот метод вряд ли полезен для чтения данных стандартного типа, например, `text/plain` или `image/gif`, кроме тех случаев, когда требуется создать собственный обработчик этих типов данных.

Отправка данных формы

В предыдущем разделе описывался способ чтения данных с веб-сервера, а в этом разделе рассматривается способ передачи данных из клиентской программы на веб-сервер, а также другим программам, которые может вызывать веб-сервер. Для передачи данных из браузера на веб-сервер нужно заполнить форму, аналогичную приведенной на рис. 3.8.

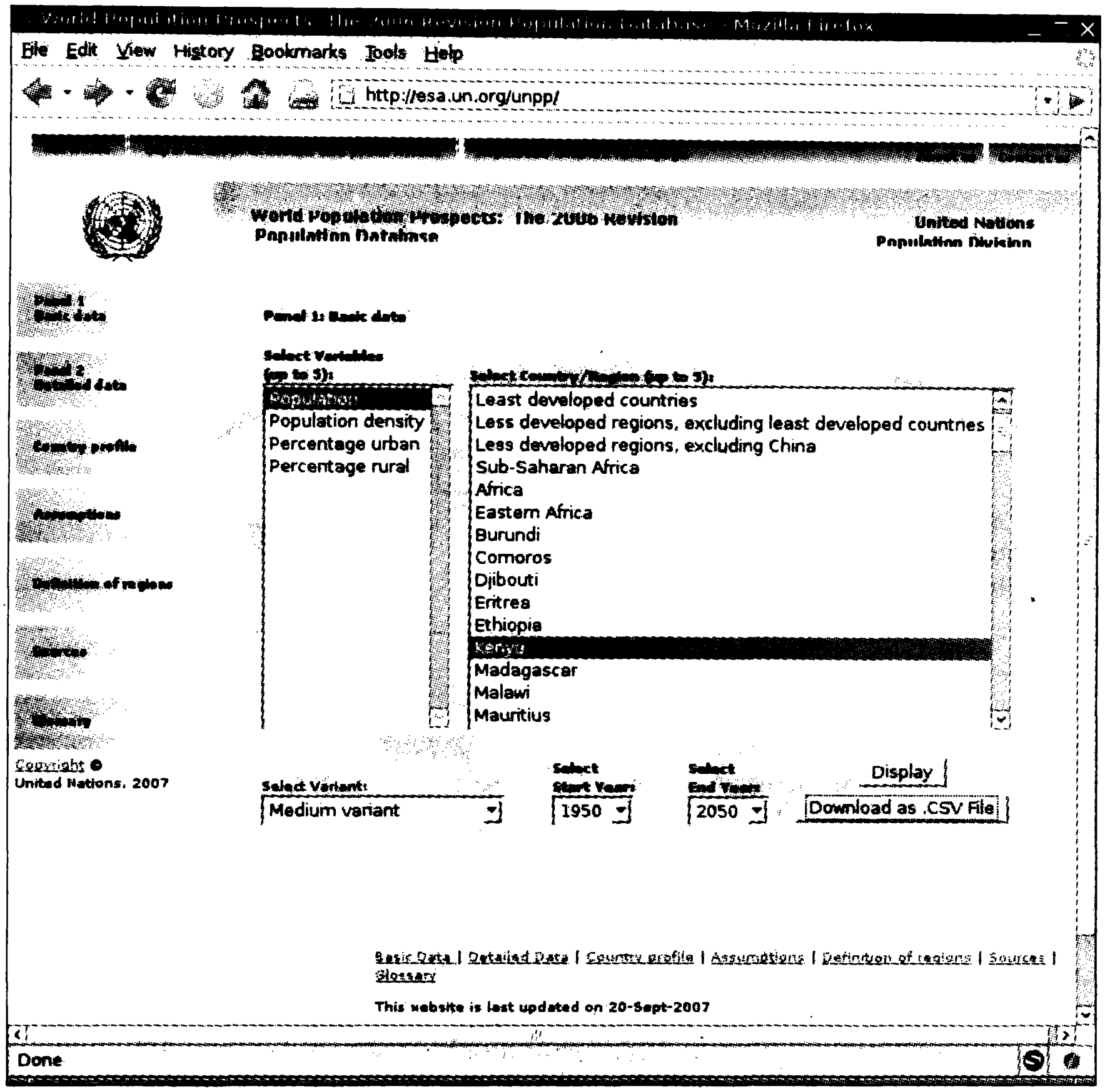


Рис. 3.8. HTML-форма

Когда пользователь щелкает на кнопке Submit (Предъявить), данные, введенные в текстовых полях, а также сведения о состоянии флажков и кнопок-переключателей передаются на веб-сервер. Получив данные, введенные пользователем в форме, веб-сервер вызывает программу для их последующей обработки.

Существует целый ряд технологий, позволяющих веб-серверу вызывать программы для обработки данных. Наиболее часто для этой цели используются сервлеты на Java, JavaServer Faces, Microsoft ASP (Active Server Pages) и сценарии CGI. Ради простоты все программы, выполняющиеся на стороне сервера, будут обозначаться единообразно как *сценарий*, независимо от конкретной применяемой технологии.

Сценарий, выполняющийся на стороне сервера, обрабатывает данные, введенные пользователем в форме, и формирует новую HTML-страницу, которую веб-сервер передает обратно браузеру. Последовательность действий по обработке данных из формы условно показана на рис. 3.9. Страница, сформированная сервером, может содержать новые данные (например, результаты поиска) или только подтверждение о получении введенных данных. Здесь и далее не рассматриваются вопросы реализации сценариев, выполняемых на стороне сервера, а основное внимание уделяется написанию клиентских программ, предназначенных для взаимодействия с готовыми сценариями.

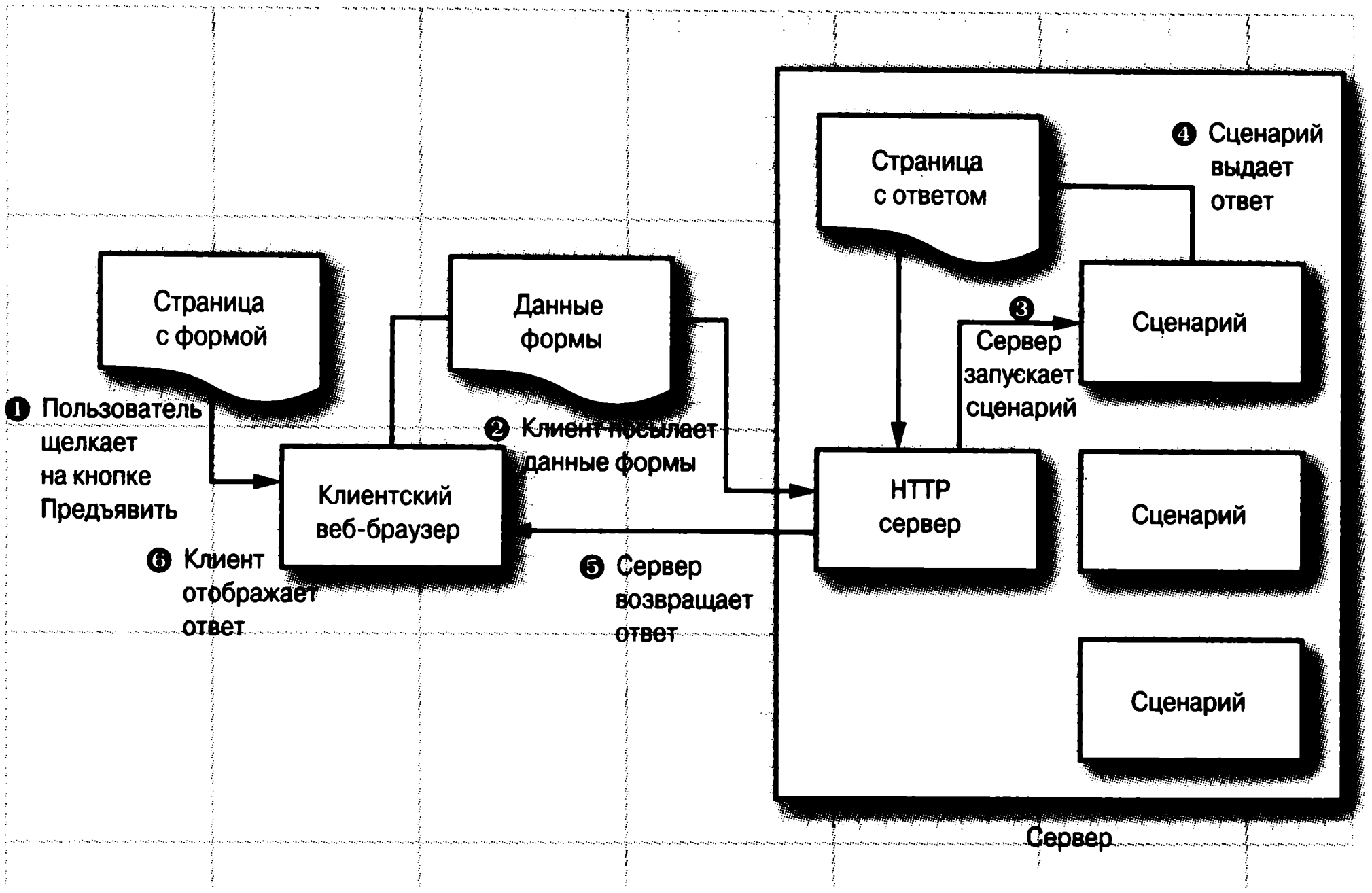


Рис. 3.9. Порядок обработки данных по сценарию на стороне сервера

При передаче данных на веб-сервер не имеет никакого значения, будет ли использован для их интерпретации сценарий CGI, сервлет или программа другого типа. Клиент посылает данные на веб-сервер в стандартном формате, а веб-сервер должен сам найти ту программу, которая выдаст нужный ответ.

Передача данных на веб-сервер может осуществляться по командам `GET` и `POST`. При выдаче команды `GET` параметры просто указываются в конце URL в следующем формате:

```
http://хост/сценарий?параметры
```

Каждый параметр имеет вид "имя–значение". Параметры разделяются знаками **&**. Значения параметров кодируются по схеме кодирования URL, которая подчиняется следующим правилам.

- Символы от **A** до **Z**, от **a** до **z**, от **0** до **9**, а также знаки **.**, **-**, **~** и **_** остаются без изменения.
- Все пробелы заменяются знаками **+**.
- Все остальные символы кодируются в кодировке UTF-8, а каждый байт преобразуется к виду **%UV**, где **UV** — двухзначное шестнадцатеричное число.

Например, название города и штата *New York, NY* передается в закодированном виде как `New+York%2c+NY`. Здесь шестнадцатеричное число **2c** (или десятичное **44**) обозначает запятую в коде ASCII. Благодаря такому способу кодирования промежуточные программы не будут путаться в пробелах и смогут правильно интерпретировать другие символы.

Так, на момент написания этой книги веб-сервер Yahoo! содержал сценарий `py.maps.py` на хосте `maps.yahoo.com`. Для выполнения этого сценария требуются два параметра: `addr` и `csz`. Чтобы получить карту местности по адресу 1 Market Street, San Francisco, CA, нужно сформировать следующий URL:

```
http://maps.yahoo.com/py/maps.py?addr=1+Market+Street&csz=San+Francisco+CA
```

Команда `GET` очень проста в употреблении, но имеет существенный недостаток: в большинстве браузеров количество символов, которые можно включить в запрос по команде `GET`, ограничено.

Параметры команды `POST` не нужно включать в состав URL. Вместо этого нужно получить поток вывода из объекта типа `URLConnection` и записать в него пары "имя–значение". Кроме того, значения, включаемые в URL, необходимо закодировать, разделив их знаком **&**. Рассмотрим этот процесс более подробно. Для передачи данных сценарию сначала создается объект типа `URLConnection` следующим образом:

```
URL url = new URL("http://хост/сценарий");
URLConnection connection = url.openConnection();
```

Затем вызывается метод `setDoOutput()`, чтобы установить соединение для передачи данных:

```
connection.setDoOutput(true);
```

Далее вызывается метод `getOutputStream()`, чтобы получить поток вывода. Для передачи текстовых данных поток вывода удобно инкапсулировать в объект типа `PrintWriter` следующим образом:

```
PrintWriter out = new PrintWriter(connection.getOutputStream());
```

Теперь можно передать данные на сервер, как следует из приведенного ниже фрагмента кода.

```
out.print(name1 + "=" + URLEncoder.encode(value1, "UTF-8") + "&");
out.print(name2 + "=" + URLEncoder.encode(value2, "UTF-8"));
```

После передачи данных выходной поток вывода закрывается следующим образом:

```
out.close();
```

И, наконец, вызывается метод `getInputStream()`, чтобы прочитать ответ с сервера.

Рассмотрим следующий пример. Веб-сайт `http://esa.un.org/unpd/wpp/unpp/panel_population.htm` содержит страницу с формой для ввода запроса на получение статистических сведений о численности и структуре населения (см. рис. 3.8). В HTML-разметке данной формы содержится следующий дескриптор:

```
<form method="post" onSubmit="return checksubmit(. . .)" . . .>
```

Если проанализировать исходный код функции `checksubmit()` сценария на JavaScript, то можно обнаружить, что загрузка значений, разделенных запятыми, вызывает приведенное ниже действие, в котором указывается имя сценария, выполняющего команду `POST`.

```
document.menuForm.action ="p2k0data_script.asp";
```

Далее нужно найти имена полей, которые потребуются указанному сценарию. Внимательно проанализируем элементы разметки формы. У каждого из них имеется свой атрибут `name`, как показано ниже.

```
<select name="Variable">
<option value="12;">Population</option>
  дополнительные параметры . . .
</select>
```

В данном случае поле имеет имя `Variable`. Оно содержит тип заполнения таблиц статистическими данными. Например, таблица типа `"12;"` содержит оценки общей численности населения. Далее в HTML-разметке данной формы можно найти поле `Location` с параметрами, обозначающими сокращенные названия стран, например **900** (весь мир) и **404** (Кения).

Необходимо также установить ряд других полей. Например, чтобы получить оценку численности населения в Кении в период с 1950 по 2050 гг., нужно сформировать следующую символьную строку:

```
Panel=1&Variable=12%3b&Location=404&Varient=2&StartYear=1950&EndYear=2050;
```

Затем эта символьная строка посылается по следующему URL:

```
http://esa.un.org/unpd/wpp/unpp/p2k0data.asp
```

Обработав полученные данные, сценарий пришлет в итоге следующий ответ:

```
"Country","Variable","Variant","Year","Value"
"Kenya","Population (thousands)","Medium variant","1950",6077
"Kenya","Population (thousands)","Medium variant","1955",6984
"Kenya","Population (thousands)","Medium variant","1960",8115
"Kenya","Population (thousands)","Medium variant","1965",9524
. . .
```

Как видите, сценарий отправляет назад файл с данными, разделенными запятыми. Этот сценарий был выбран для данного примера потому, что в нем нетрудно разобраться, не утруждая себя анализом HTML-дескрипторов в более сложных сценариях.

В листинге 3.7 приведен исходный код примера программы, посылающей данные на сервер по команде `POST`. В файл свойств с расширением `.properties` вводятся следующие данные:

```
url=http://esa.un.org/unpd/wpp/unpp/p2k0data_script.asp
Panel=1
Variable=12;
Location=404
Varient=2
StartYear=1950
EndYear=2050
```

Программа удаляет элемент `url`, а все остальные элементы направляет методу `doPost()`. В методе `doPost()` сначала устанавливается соединение, затем вызывается метод `setDoOutput(true)` и открывается поток вывода. Затем перечисляются все ключи и значения. Для каждой пары "ключ–значение" по очереди передаются ключ, знак **=**, значение и разделительный знак **&**:

```
out.print(key);
out.print('=');
out.print(URLEncoder.encode(value, UTF-8"));
if (дополнительные пары "ключ-значение") out.print('&');
```

И наконец, в данной программе читается ответ с сервера. Следует, однако, иметь в виду, что если при выполнении сценария возникнет ошибка, то вызов метода `connection.getInputStream()` приведет к исключению типа `FileNotFoundException`. Тем не менее сервер продолжит передачу данных,

отправив HTML-страницу с сообщением об ошибке. Обычно это сообщение "Error 404-page not found", уведомляющее о том, что данная страница не найдена. Для перехвата этой страницы нужно привести объект типа URLConnection к типу HttpURLConnection и вызвать метод getErrorStream() следующим образом:

```
InputStream err = ((HttpURLConnection) connection).getErrorStream();
```

Любопытно, какие же данные объект типа URLConnection передает на сервер в дополнение к введенным данным? Объект типа URLConnection сначала посылает на сервер заголовок запроса. При отправке данных формы заголовок содержит следующие сведения:

```
Content-Type: application/x-www-form-urlencoded
```

Заголовок для команды POST должен также включать в себя сведения о длине содержимого, например:

```
Content-Length: 124
```

Конец заголовка обозначается пустой строкой. Затем следует порция передаваемых данных. Веб-сервер отделяет заголовок и направляет полученную порцию данных сценарию, выполняемому на стороне сервера. Следует также иметь в виду, что объект типа URLConnection буферизирует все данные, которые направляются в поток вывода, поскольку он должен сначала определить общую длину содержимого.

Прием, применяемый в данной программе, очень удобен для формирования запросов на получение данных с веб-сервера. Для этого следует проанализировать HTML-разметку веб-страницы, формирующей аналогичный запрос, чтобы выяснить требующиеся параметры и удалить из ответа все лишние HTML-дескрипторы и другие ненужные данные.

Листинг 3.7. Исходный код из файла post/PostTest.java

```
1  package post;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.file.*;
6  import java.util.*;
7
8  /**
9   * В этой программе демонстрируется применение класса URLConnection
10  * для формирования запроса по команде POST
11  * @version 1.30 2012-06-04
12  * @author Cay Horstmann
13 */
14 public class PostTest
15 {
16   public static void main(String[] args) throws IOException
17   {
18     Properties props = new Properties();
19     try (InputStream in = Files.newInputStream(Paths.get(args[0])))
20     {
21       props.load(in);
22     }
23     String url = props.remove("url").toString();
24     String result = doPost(url, props);
25     System.out.println(result);
```

```
26    }
27
28    public static String doPost(String urlString,
29          Map<Object,   Object> nameValuePairs) throws IOException
30    {
31       URL url = new URL(urlString);
32       URLConnection connection = url.openConnection();
33       connection.setDoOutput(true);
34
35       try (PrintWriter out =
36             new PrintWriter(connection.getOutputStream()))
37       {
38          boolean first = true;
39          for (Map.Entry<Object, Object> pair : nameValuePairs.entrySet())
40          {
41             if (first) first = false;
42             else out.print('&');
43
44             String name = pair.getKey().toString();
45             String value = pair.getValue().toString();
46             out.print(name);
47             out.print('=');
48             out.print(URLEncoder.encode(value, "UTF-8"));
49          }
50       }
51
52       StringBuilder response = new StringBuilder();
53       try (Scanner in = new Scanner(connection.getInputStream()))
54       {
55          while (in.hasNextLine())
56          {
57             response.append(in.nextLine());
58             response.append("\n");
59          }
60       }
61       catch (IOException e)
62       {
63          if (!(connection instanceof HttpURLConnection)) throw e;
64          InputStream err = ((HttpURLConnection) connection).getErrorStream();
65          if (err == null) throw e;
66          Scanner in = new Scanner(err);
67          response.append(in.nextLine());
68          response.append("\n");
69       }
70
71       return response.toString();
72    }
73 }
```

java.net.HttpURLConnection 1.0

- `InputStream getErrorStream()`

  Возвращает поток для ввода сообщений веб-сервера об ошибках.

java.net.URLEncoder 1.0

- `static String encode(String s, String encoding)` **1.4**

  Возвращает символьную строку `s` в закодированном для URL виде. Кодирование производится по заданной схеме. При кодировании URL символы **A-Z**, **a-z**, **0-9**, **-**, **_**, **.** и **~** остаются без изменения. Пробелы передаются в виде знаков **+**, а все остальные символы — в виде **%UV**, где **UV** — шестнадцатеричное представление байта.

java.net.URLDecoder 1.2

- `static string decode(String s, String encoding)` **1.4**

  Возвращает в декодированном виде символьную строку `s`, закодированную для URL. Декодирование производится по заданной схеме.

Отправка электронной почты

Раньше для написания программы, отправляющей электронную почту, достаточно было установить сокетное соединение через порт **25**, который обычно используется для обмена данными по протоколу SMTP. А сам протокол SMTP (Simple Mail Transport Protocol — простой протокол для передачи почты) описывает формат сообщений электронной почты. Как только будет установлено соединение с сервером, нужно послать заголовок почтового сообщения (его нетрудно составить в формате SMTP), а затем и текст самого сообщения, выполнив следующие действия.

1. Откройте сокет на своем компьютере, подключенном к Интернету, как показано ниже.

```
Socket s = new Socket("mail.yourserver.com", 25); // номер порта 25
                                                  // соответствует протоколу SMTP
PrintWriter out = new PrintWriter(s.getOutputStream());
```

2. Направьте в поток вывода следующие данные:

```
HELO компьютер отправителя
MAIL FROM: адрес отправителя
RCPT TO: адрес получателя
DATA
почтовое сообщение
(любое количество строк)
.
QUIT
```

В спецификации протокола SMTP (документ RFC 821) требуется, чтобы строки завершались комбинациями символов **/r** и **/n**. Первоначально SMTP-серверы исправно направляли электронную почту от любого адресата. Но когда навязчивые сообщения наводнили Интернет, большинство этих серверов были оснащены встроенными проверками, принимая запросы только по тем IP-адресам, которым они доверяют. Аутентификация обычно происходит через безопасные сокетные соединения.

Реализовать алгоритмы подобной аутентификации вручную — дело непростое. Поэтому в этом разделе будет показано, как пользоваться прикладным интерфейсом JavaMail API для отправки сообщений электронной почты из программы на Java. С этой целью загрузите данный прикладной интерфейс по адресу `www.oracle.com/technetwork/java/javamail` и разархивируйте его на жесткий диск своего компьютера.

Для того чтобы воспользоваться прикладным интерфейсом JavaMail API, необходимо установить некоторые свойства, зависящие от конкретного почтового сервера. В качестве примера ниже приведены свойства, устанавливаемые для почтового сервера GMail. Они считываются из файла свойств в рассматриваемом здесь примере программы, исходный код которой приведен в листинге 3.8.

```
mail.transport.protocol=smtps
mail.smtps.auth=true
mail.smtps.host=smtp.gmail.com
mail.smtps.user=cayhorstmann@gmail.com
```

Из соображений безопасности пароль не вводится в файл свойств, но предлагается для ввода вручную. После чтения из файла свойств сеанс почтовой связи устанавливается следующим образом:

```
Session mailSession = Session.getDefaultInstance(props);
```

Затем составляется почтовое сообщение с указанием требуемого отправителя, получателя, темы и текста самого сообщения, как показано ниже.

```
MimeMessage message = new MimeMessage(mailSession);
message.setFrom(new InternetAddress(from));
message.addRecipient(RecipientType.TO, new InternetAddress(to));
message.setSubject(subject);
message.setText(builder.toString());
```

Далее почтовое сообщение отправляется следующим образом:

```
Transport tr = mailSession.getTransport();
tr.connect(null, password);
tr.sendMessage(message, message.getAllRecipients());
tr.close();
```

Рассматриваемая здесь программа читает почтовое сообщение из текстового файла в приведенном ниже формате.

```
Отправитель
Получатель
Тема
Текст сообщения (любое количество строк)
```

Для запуска данной программы на выполнение введите приведенную ниже команду, где `mail.jar` — архивный JAR-файл, входящий в состав распространяемой версии прикладного интерфейса JavaMail API. (При указании пути к классу пользователи Windows должны ввести после параметра **`-classpath`** точку с запятой вместо двоеточия.)

```
java -classpath .:path/to/mail.jar path/to/message.txt
```

На момент написания данной книги почтовый сервер GMail не проверял достоверность получаемой информации, а следовательно, в почтовом сообщении можно было указать любого отправителя. (Это обстоятельство следует иметь в виду при получении от отправителя по адресу `president@whitehouse.gov` очередного приглашения на официальный прием, организуемый на лужайке перед Белым домом.)

СОВЕТ. Если вам не удастся выяснить причину, по которой соединение с почтовым сервером не действует, сделайте следующий вызов и проверьте почтовые сообщения:

```
mailSession.setDebug(true);
```

Кроме того, обратитесь за полезными советами на веб-страницу JavaMail API FAQ (Часто задаваемые вопросы по прикладному интерфейсу JavaMail API FAQ) по адресу http://www.oracle.com/technetwork/java/javamail/faq-135477.html.

Листинг 3.8. Исходный код из файла mail/MailTest.java

```
1  package mail;
2
3  import java.io.*;
4  import java.nio.charset.*;
5  import java.nio.file.*;
6  import java.util.*;
7  import javax.mail.*;
8  import javax.mail.internet.*;
9  import javax.mail.internet.MimeMessage.RecipientType;
10
11 /**
12  * В этой программе демонстрируется применение прикладного интерфейса
13  * JavaMail API для отправки сообщений по электронной почте
14  * @author Cay Horstmann
15  * @version 1.00 2012-06-04
16 */
17 public class MailTest
18 {
19   public static void main(String[] args)
20         throws MessagingException, IOException
21   {
22     Properties props = new Properties();
23     try (InputStream in =
24          Files.newInputStream(Paths.get("mail", "mail.properties")))
25     {
26       props.load(in);
27     }
28     List<String> lines =
29       Files.readAllLines(Paths.get(args[0]), Charset.forName("UTF-8"));
30
31     String from = lines.get(0);
32     String to = lines.get(1);
33     String subject = lines.get(2);
34
35     StringBuilder builder = new StringBuilder();
36     for (int i = 3; i < lines.size(); i++)
37     {
38       builder.append(lines.get(i));
39       builder.append("\n");
40     }
41
42     Console console = System.console();
43     String password = new String(console.readPassword("Password: "));
44
45     Session mailSession = Session.getDefaultInstance(props);
```

```
46      // mailSession.setDebug(true);
47      MimeMessage message = new MimeMessage(mailSession);
48      message.setFrom(new InternetAddress(from));
49      message.addRecipient(RecipientType.TO, new InternetAddress(to));
50      message.setSubject(subject);
51      message.setText(builder.toString());
52      Transport tr = mailSession.getTransport();
53      try
54      {
55        tr.connect(null, password);
56        tr.sendMessage(message, message.getAllRecipients());
57      }
58      finally
59      {
60        tr.close();
61      }
62    }
63  }
```

В этой главе было показано, каким образом на Java пишется исходный код программ для сетевых клиентов и серверов и как организуется сбор данных с веб-серверов. А в следующей главе речь пойдет о взаимодействии с базами данных. Из нее вы узнаете, как работать с реляционными базами данных в программах на Java, используя прикладной интерфейс JDBC API. В следующей главе дается также краткое введение в управление соединениями с базами данных, а также прикладной интерфейс JNDI API.

ГЛАВА 4

Работа с базами данных

В этой главе...

- Структура JDBC
- Язык SQL
- Конфигурирование JDBC
- Выполнение команд SQL
- Выполнение запросов
- Прокручиваемые и обновляемые результирующие наборы
- Наборы строк
- Метаданные
- Транзакции
- Управление соединениями с базами данных в веб-приложениях и производственных приложениях

В 1996 году компания Sun Microsystems выпустила первую версию прикладного интерфейса API для организации доступа из программ на Java к базам данных (JDBC). Этот прикладной интерфейс позволяет соединяться с базой данных, запрашивать и обновлять данные с помощью языка структурированных запросов (Structured Query Language — SQL). Язык SQL фактически стал стандартным средством взаимодействия с реляционными базами данных. С тех пор JDBC стал одним из наиболее употребительных прикладных интерфейсов API в библиотеке Java.

Средства JDBC неоднократно обновлялись. В состав комплекта JDK 1.2, выпущенного в 1998 году, была включена версия JDBC 2. А версия JDBC 3 была включена в состав Java SE 1.4 и 5.0. На момент написания данной книги последней была версия JDBC 4.1, входившая в состав Java SE 7.

В этой главе рассматриваются принципы, положенные в основу прикладного интерфейса JDBC. Из нее вы узнаете (а возможно, лишь вспомните) о языке SQL, который является стандартным средством доступа к реляционным базам данных. В ней будут также рассмотрены примеры применения интерфейса JDBC, демонстрирующие наиболее распространенные приемы обращения с базами данных в прикладных программах.

НА ЗАМЕТКУ! Как заявляют в компании Oracle, JDBC — это торговая марка, а не сокращение Java Database Connectivity. Она была придумана по аналогии с обозначением ODBC стандартного прикладного интерфейса для работы с базами данных, который был первоначально предложен корпорацией Microsoft и затем внедрен в стандарт SQL.

Структура JDBC

Создатели Java с самого начала осознавали потенциальные преимущества данного языка для работы с базами данных. С 1995 года они начали работать над расширением стандартной библиотеки Java для организации доступа к базам данных средствами SQL. Сначала они попробовали создать такие расширения Java, которые позволили бы осуществлять доступ к произвольной базе данных *только* средствами Java. Но очень скоро они убедились в бесперспективности такого подхода, поскольку для доступа к базам данных применялись самые разные протоколы. Кроме того, поставщики программного обеспечения баз данных были весьма заинтересованы в разработке на Java стандартного сетевого протокола для доступа к базам данных, но при условии, что за основу будет принят их *собственный* сетевой протокол.

В конечном итоге поставщики баз данных и инструментальных средств для доступа к ним сошлись на том, что лучше предоставить прикладной интерфейс API только на Java для доступа к базам данных средствами SQL, а также диспетчер драйверов, который позволил бы подключать к базам драйверы независимых производителей. Такой подход позволял поставщикам баз данных создавать собственные драйверы, которые подключались бы с помощью данного диспетчера. Предполагалось, что это будет простой механизм регистрации драйверов.

Подобная организация прикладного интерфейса JDBC основана на весьма успешной модели интерфейса ODBC, разработанного в корпорации Microsoft. В основу JDBC и ODBC положен общий принцип: программы, написанные в соответствии с требованиями прикладного интерфейса API, способны взаимодействовать с диспетчером драйверов JDBC, который, в свою очередь, использует подключаемые драйверы для обращения к базе данных. Это означает, что для работы с базами данных в прикладных программах достаточно пользоваться средствами JDBC API.

Типы драйверов JDBC

Каждый драйвер JDBC принадлежит к одному из перечисленных ниже типов.

- **Драйвер типа 1.** Преобразует JDBC в ODBC и для взаимодействия с базой данных использует драйвер ODBC. Один такой драйвер был включен в первые версии Java под названием *мост JDBC/ODBC*. Но для его применения требуется установить и настроить соответствующим образом драйвер ODBC. В первом выпуске JDBC этот мост предполагалось использовать только для тестирования, а не для применения в рабочих программах. В настоящее время уже имеется достаточное количество более удачных драйверов, поэтому пользоваться мостом JDBC/ODBC не рекомендуется.

- **Драйвер типа 2.** Разрабатывается преимущественно на Java и частично на собственном языке программирования, который используется для взаимодействия с клиентским прикладным интерфейсом API базы данных. Для применения такого драйвера, помимо библиотеки Java, на стороне клиента нужно установить специфический для данной платформы код.
- **Драйвер типа 3.** Разрабатывается только на основе клиентской библиотеки Java, в которой используется независимый от базы данных протокол передачи запросов базы данных на сервер. Этот протокол приводит запросы базы данных в соответствие с характерным для нее протоколом. Развертывание прикладных программ значительно упрощается благодаря тому, что код, зависящий от конкретной платформы, находится только на сервере.
- **Драйвер типа 4.** Представляет собой библиотеку, написанную только на Java, для приведения запросов JDBC в соответствие с протоколом конкретной базы данных.

Большинство поставщиков баз данных предоставляют драйверы типа 3 или 4. Кроме того, целый ряд сторонних производителей специализируется на создании драйверов, которые позволяют добиться более полного соответствия принятым стандартам, поддерживают большее количество платформ, обладают более высокой производительностью или надежностью, чем драйверы, предлагаемые поставщиками баз данных.

Основные цели прикладного интерфейса JDBC можно сформулировать следующим образом.

- Разработчики пишут программы на Java, пользуясь для доступа к базам данных стандартными средствами языка SQL (или его специализированными расширениями), но следуя только принятым в Java соглашениям.
- Поставщики баз данных и инструментальных средств к ним предоставляют драйверы только низкого уровня. Это дает им возможность оптимизировать драйверы под свою конкретную продукцию.

На заметку! На конференции JavaOne в мае 1996 года представители компании Sun Microsystems указали на ряд следующих причин отказа от модели ODBC.

- Трудна в освоении.
- Имеет всего лишь несколько команд с большим количеством параметров, тогда как стиль программирования на Java основан на применении большого количества простых и интуитивно понятных методов.
- Основана на использовании указателей типа `void*` и других элементов языка C, отсутствующих в Java.
- Менее безопасна и труднее для развертывания, чем решение, получаемое только на Java.

Типичные примеры применения JDBC

Согласно традиционной модели "клиент–сервер" графический пользовательский интерфейс (ГПИ) реализуется на стороне клиента, а база данных располагается на стороне сервера (рис. 4.1). В этом случае драйвер JDBC развертывается на стороне клиента.

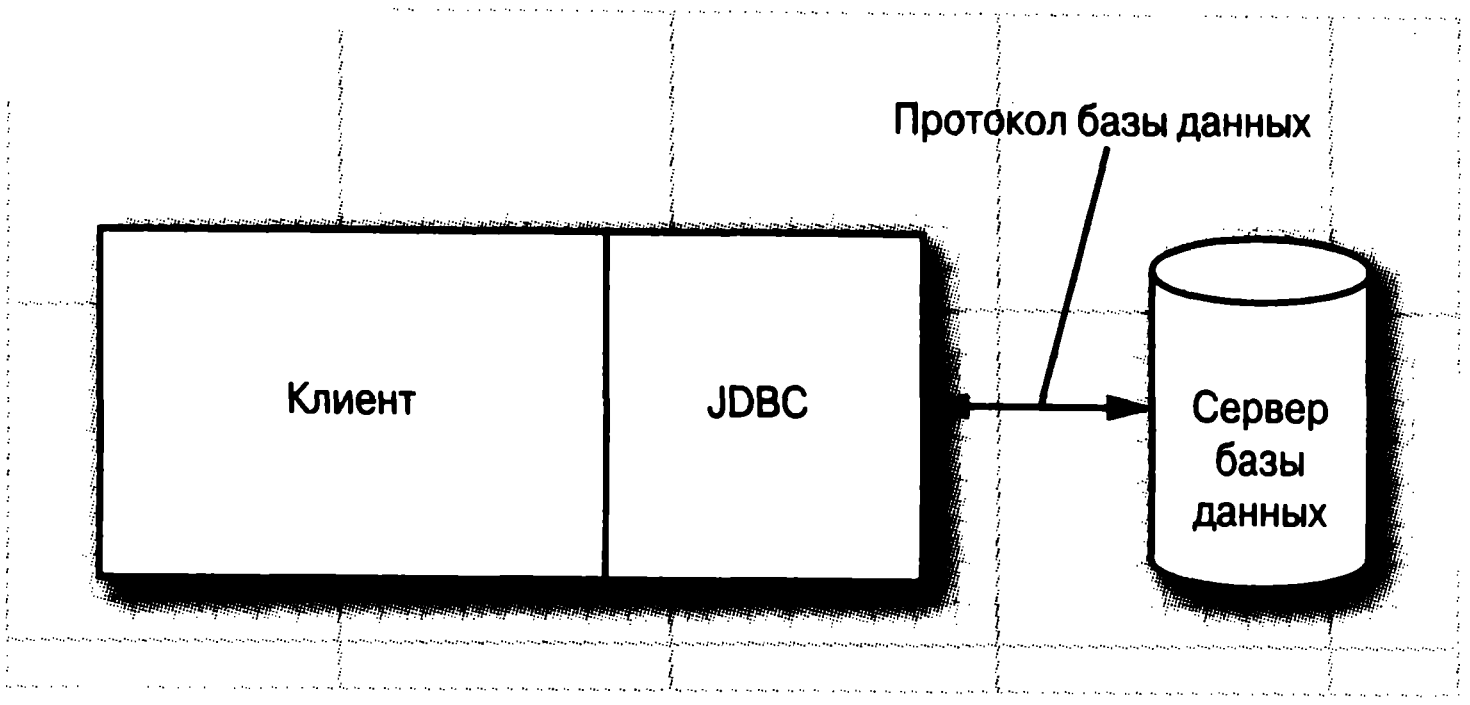


Рис. 4.1. Традиционная структура приложений "клиент-сервер"

Но в настоящее время существует явная тенденция к переходу от архитектуры "клиент–сервер" к трехуровневой модели или даже более совершенной *n*-уровневой модели. В трехуровневой модели клиент не формирует обращения к базе данных. Вместо этого он обращается к средствам промежуточного уровня на сервере, который, в свою очередь, выполняет запросы к базе данных. Трехуровневая модель обладает двумя преимуществами: отделяет *визуальное представление* (на клиентском компьютере) от *бизнес-логики* (на промежуточном уровне) и *исходных данных* (хранящихся в базе данных). Таким образом, становится возможным доступ к тем же самым данным по таким же бизнес-правилам со стороны разнотипных клиентов, в том числе прикладных программ на Java, аплетов или веб-форм.

Взаимодействие между клиентом и промежуточным уровнем может быть реализовано по протоколу HTTP (при использовании веб-браузера в качестве клиента) или с помощью какого-нибудь другого механизма, например вызова удаленных методов (Remote Method Invocation — RMI; подробнее об этом — в главе 11). А прикладной интерфейс JDBC используется для управления взаимодействием между промежуточным уровнем и серверной базой данных. На рис. 4.2 схематически показана основная архитектура трехуровневой модели. Следует, однако, иметь в виду, что у этой модели существует немало вариантов. Так, в версии Java Enterprise Edition определена структура *серверов приложений*, которые управляют компонентами Enterprise JavaBeans и предоставляют такие ценные услуги, как балансировка нагрузки, кеширование запросов, обеспечение безопасности и объектно-реляционное преобразование. В этой архитектуре прикладной интерфейс JDBC играет важную роль, направляя сложные запросы к базе данных. (Более подробные сведения по данному вопросу можно найти, обратившись по адресу `www.oracle.com/technetwork/java/javaee/overview`.)

НА ЗАМЕТКУ! Средства JDBC можно также применять в аплетах и приложениях Web Start, но на практике вряд ли стоит делать это. По умолчанию диспетчер безопасности разрешает устанавливать соединение только с тем сервером, из которого был скопирован аплет. Это означает, что сервер базы данных должен быть расположен на том же компьютере, где и веб-сервер (или компонент ретрансляции драйвера типа 3), что, как правило, не делается. В качестве выхода из данного положения приходится использовать цифровую подпись.

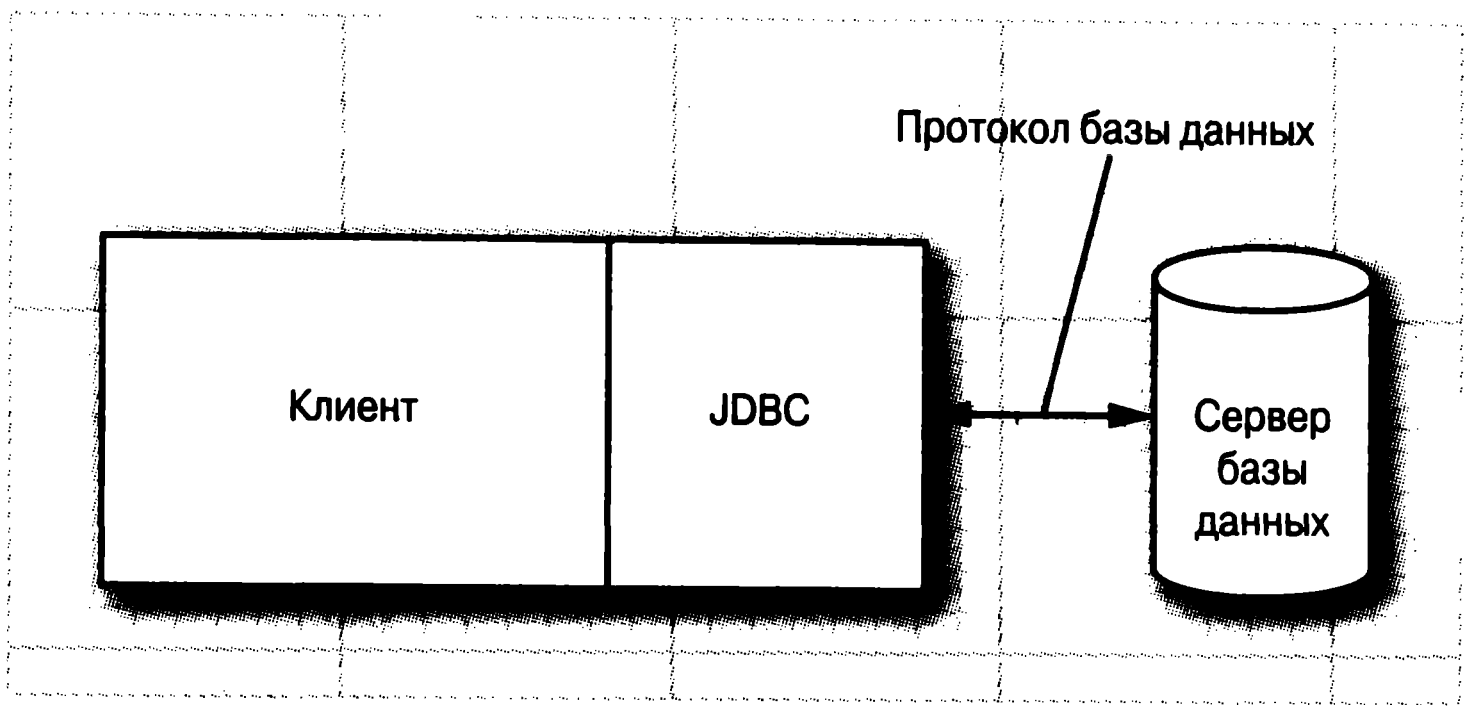


Рис. 4.2. Структура приложений на основе трехуровневой модели

Язык SQL

Прикладной интерфейс JDBC позволяет взаимодействовать с базами данных с помощью языка SQL, который, в свою очередь, образует интерфейс для большинства современных реляционных баз данных. Настольные базы данных предоставляют графический интерфейс, который дает пользователям возможность непосредственно манипулировать данными, но доступ к серверным базам данных возможен только с помощью SQL.

Пакет JDBC можно рассматривать лишь как прикладной интерфейс API для взаимодействия с командами и операторами языка SQL для доступа к базам данных. В этом разделе приводится краткое описание языка SQL. Если вам не приходилось раньше иметь дело с SQL, сведений, представленных в этом разделе, может оказаться недостаточно. Для более досконального изучения основ SQL можно порекомендовать книгу *Learning SQL* Алана Билю (Alan Beaulieu; издательство O'Reilly, 2005 г.) или классическое издание *A Guide to the SQL Standard* К. Дж. Дейта и Хью Дарвена (C. J. Date & Hugh Darwen; издательство Addison-Wesley, 1997 г.).

База данных представляет собой набор именованных таблиц со строками и столбцами. Каждый столбец имеет свое *имя*, а данные хранятся в строках. В качестве примера базы данных здесь и далее рассматривается ряд таблиц с описаниями библиотеки классических книг по вычислительной технике (табл. 4.1–4.4).

Таблица 4.1. Таблица `Authors`

Author_ID	Name	Fname
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
...	...	...

Таблица 4.2. Таблица `Books`

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
...	...	...	

Таблица 4.3. Таблица BooksAuthors

ISBN	Author_ID	Seq_No
0-201-96426-0	DATE	1
0-201-96426-0	DARW	2
0-19-501919-9	ALEX	1
...	...	...

Таблица 4.4. Таблица Publishers

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com
...	...	...

На рис. 4.3 представлена таблица Books, а на рис. 4.4 — результат *соединения* таблиц Books и Publishers. Обе таблицы содержат идентификатор издателя. При соединении таблиц по этому коду получается *результат запроса* в виде таблицы, содержащей данные из обеих исходных таблиц. В каждой строке этой таблицы содержатся сведения о книге, название и адрес веб-сайта издательства. Обратите внимание на то, что данные с названием книги и адресом веб-сайта неоднократно дублируются, поскольку в результирующей таблице оказывается несколько строк, относящихся к одному и тому же издательству.

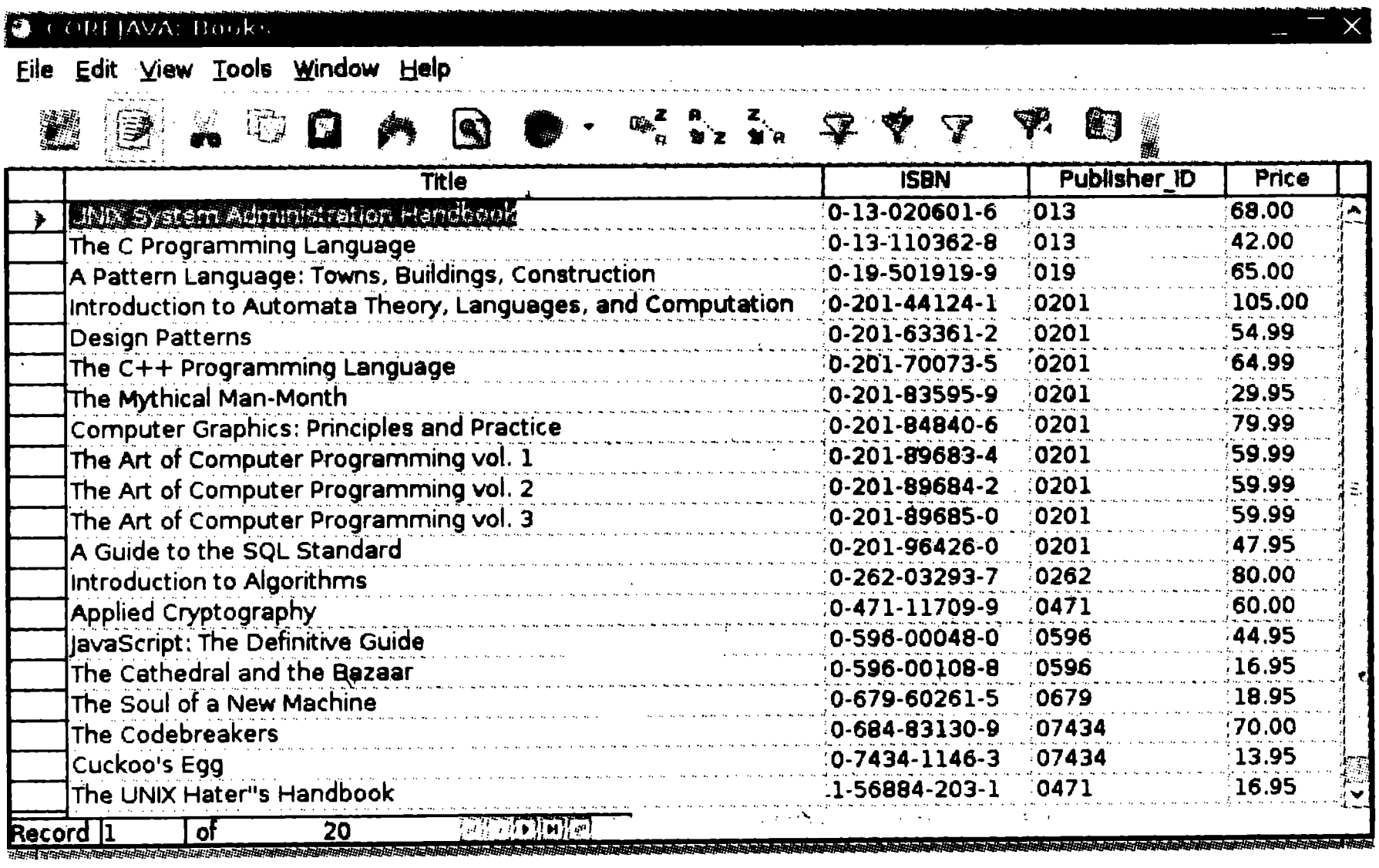


Рис. 4.3. Образец таблицы с данными о книгах

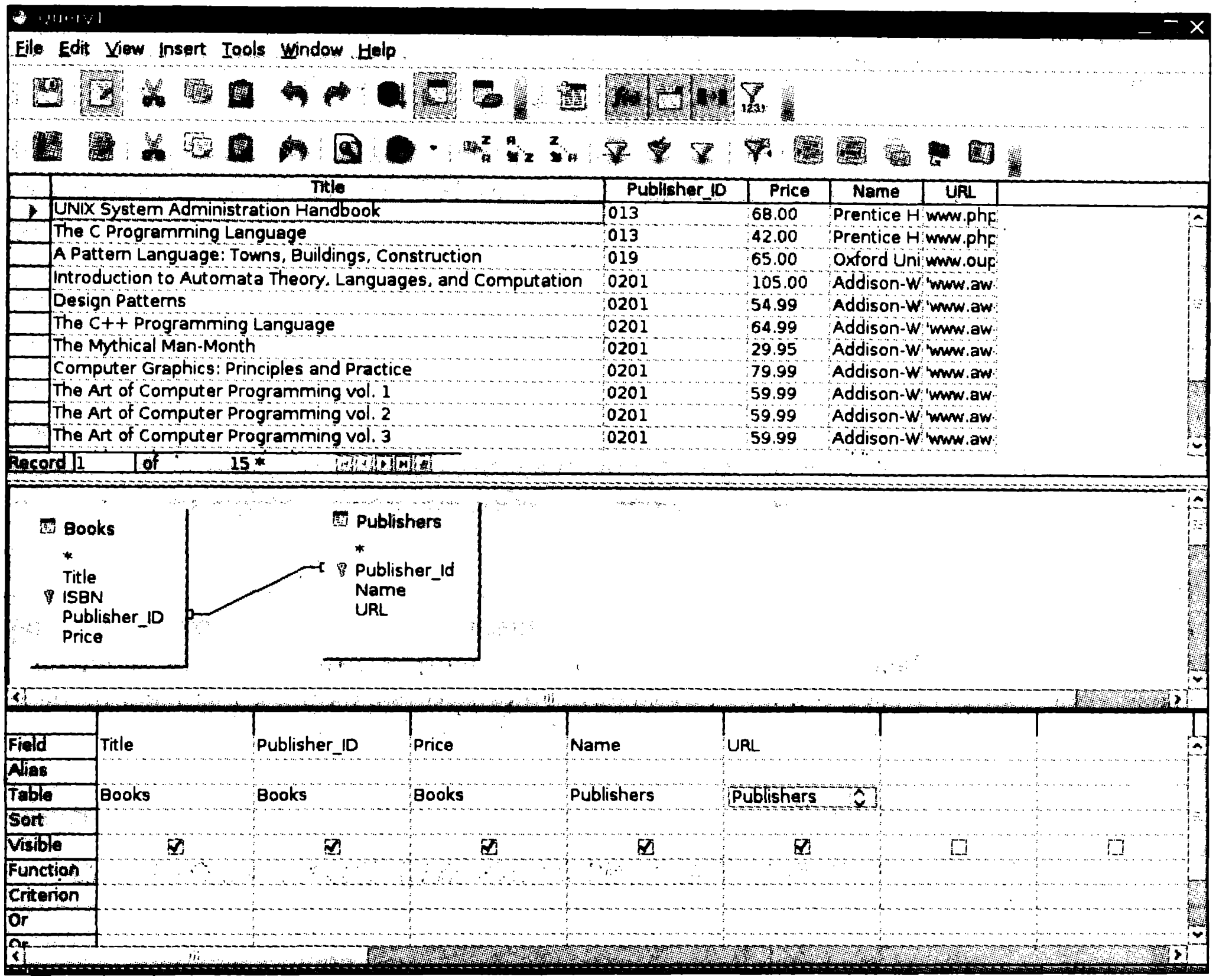


Рис. 4.4. Результат соединения двух таблиц

Преимущество соединения таблиц заключается в том, что в этом случае удается избежать нежелательного дублирования данных. Например, в простейшей структуре базы данных таблица Books может содержать столбцы с названием и адресом веб-сайта издательства. Но в таком случае данные будут дублироваться уже не только в результате запроса, но и в самой базе данных.

При изменении адреса веб-сайта придется также изменить эти данные во *всех* записях в базе данных. Очевидно, что при выполнении столь трудоемкой задачи могут легко возникнуть ошибки. В реляционной модели данные распределяются среди нескольких таблиц таким образом, чтобы они не дублировались без особой надобности. Например, адрес веб-сайта каждого издательства хранится в единственном экземпляре в таблице с данными об издательствах. А при необходимости данные из разных таблиц нетрудно соединить в результат запроса.

На рис. 4.3 и 4.4 показано инструментальное средство с ГПИ, предназначенное для просмотра и связывания таблиц. Многие поставщики программного обеспечения предлагают разнообразные диалоговые инструментальные средства для создания запросов путем манипулирования столбцами и ввода данных в готовые формы. Они называются инструментальными средствами составления *запросов по образцу* (QBE). А при использовании SQL запрос создается в текстовом виде в строгом соответствии с синтаксисом этого языка, как показано ниже.

```
SELECT Books.Title, Books.Publisher_Id, Books.Price,
       Publishers.Name, Publishers.URL
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

В оставшейся части этого раздела описываются основные способы создания подобных запросов базы данных. Читатели, знакомые с SQL, могут пропустить этот материал. Ключевые слова SQL принято вводить прописными буквами, хотя это правило не является обязательным.

Команда SELECT может применяться в самых разных целях, например, для выбора всех элементов из таблицы Books по следующему запросу:

```
SELECT * FROM Books
```

Оператор FROM обязательно указывается в каждой команде SELECT. Он сообщает базе данных о тех таблицах, в которых требуется выполнить поиск данных. В команде SELECT можно указать любые требующиеся столбцы следующим образом:

```
SELECT ISBN, Price, Title
FROM Books
```

Выбор строк можно ограничить с помощью условия, указываемого в операторе WHERE, как показано ниже.

```
SELECT ISBN, Price, Title
FROM Books
WHERE Price <= 29.95
```

Следует особенно подчеркнуть, что для сравнения в SQL используются операции = и <>, а не == или !=, как при программировании на Java.

НА ЗАМЕТКУ! Некоторые поставщики баз данных используют операцию != для обозначения сравнения, но учтите, что такое обозначение не соответствует стандарту SQL, и поэтому пользоваться им не рекомендуется.

В операторе WHERE может присутствовать оператор LIKE для сопоставления с заданным шаблоном. Но вместо обычных символов подстановки * и ? в данном операторе используется знак %, обозначающий любое количество символов, а знак _ — один символ. Ниже приведен пример запроса на выборку книг, в названиях которых отсутствуют такие слова, как UNIX или Linux.

```
SELECT ISBN, Price, Title
FROM Books
WHERE Title NOT LIKE '%n_x%'
```

Обратите внимание на то, что в запросах базы данных символьные строки заключаются в одинарные, а не в двойные кавычки. Одинарная кавычка в символьной строке обозначается парой одинарных кавычек, как в приведенном ниже примере запроса на поиск всех книг, в названиях которых содержится одинарная кавычка.

```
SELECT Title
FROM Books
WHERE Title LIKE '%''%'
```

Для того чтобы выбрать данные из нескольких таблиц, их нужно перечислить в следующем порядке:

```
SELECT * FROM Books, Publishers
```

Но без оператора WHERE такой запрос не представляет большого интереса, поскольку по нему получаются все сочетания строк из обеих таблиц. В данном случае таблица Books содержит 20 строк, а таблица Publishers — 8 строк. Поэтому результат выполнения такого запроса будет содержать 20×8 строк с большим количеством дублирующихся данных. Допустим, нужно найти только те книги, которые выпущены издательствами, перечисленными в таблице Publishers. Для поиска такого соответствия книг издательствам можно составить приведенный ниже запрос. Результат выполнения этого запроса содержит 20 строк, т.е. по одной строке на каждую книгу, поскольку на каждую книгу в таблице Publishers приходится лишь одно издательство.

```
SELECT * FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

Если в запросе указано несколько таблиц, то в двух разных местах может упоминаться одно и то же имя столбца, как в показанном выше примере (столбец Publisher_Id из таблицы Books и аналогичный столбец Publisher_Id из таблицы Publishers). Во избежание неоднозначной интерпретации имен столбцов их следует предварять префиксом с именем таблицы, например Books.Publisher_Id.

Языковыми средствами SQL можно пользоваться и для изменения данных в базе данных. Допустим, требуется снизить на 5 долларов текущую цену всех книг, в названиях которых содержится подстрока "C++". С этой целью можно составить следующий запрос:

```
UPDATE Books
SET Price = Price - 5.00
WHERE Title LIKE '%C++%'
```

Аналогично для удаления всех книг по C++ понадобится команда DELETE, как показано в приведенном ниже примере запроса.

```
DELETE FROM Books
WHERE Title LIKE '%C++%'
```

Кроме того, в SQL предусмотрены встроенные функции для вычисления средних значений, поиска максимальных и минимальных значений в столбце и выполнения многих других действий. Более подробные сведения о встроенных в SQL функциях можно найти по адресу http://sqlzoo.net. (На этом сайте имеются также отличные материалы по SQL для обучения в диалоговом режиме.)

Для ввода новых данных в таблицу обычно используется команда INSERT:

```
INSERT INTO Books
VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201', 47.95)
```

Для ввода каждой строки в таблицу приходится выполнять отдельную команду INSERT. Но прежде чем составлять запросы, изменять и вводить данные, необходимо предоставить место для их хранения, т.е. создать таблицу. Для создания новой таблицы служит команда CREATE TABLE, в которой указывается имя и тип данных каждого столбца, как показано в приведенном ниже примере.

```
CREATE TABLE Books
(
  Title CHAR(60),
  ISBN CHAR(13),
  Publisher_Id CHAR(6),
  Price DECIMAL(10,2)
)
```

В табл. 4.5 перечислены наиболее распространенные типы данных в SQL. Дополнительные операторы, задающие ключи и ограничения, употребляемые в команде CREATE TABLE, в этой книге не рассматриваются.

Таблица 4.5. Типы данных SQL

Тип данных	Описание
INTEGER или INT	Обычно 32-разрядное целое значение
SMALLINT	Обычно 16-разрядное целое значение
NUMERIC(*m*, *n*), DECIMAL(*m*, *n*) или DEC(*m*, *n*)	Десятичное числовое значение с фиксированной точкой, содержащее *m* цифр, в том числе *n* знаков после точки
FLOAT(n)	Числовое значение с плавающей точкой и точностью до *n* знаков
REAL	Обычно 32-разрядное числовое значение с плавающей точкой
DOUBLE	Обычно 64-разрядное числовое значение с плавающей точкой
CHARACTER(*n*) или CHAR(*n*)	Строка фиксированной длины *n* символов
VARCHAR(*n*)	Строка переменной длины максимум *n* символов
BOOLEAN	Логическое значение
DATE	Календарная дата (зависит от реализации)
TIME	Время (зависит от реализации)
TIMESTAMP	Дата и время (зависят от реализации)
BLOB	Большой двоичный объект
CLOB	Большой символьный объект

Конфигурирование JDBC

Разумеется, для работы с базой данных потребуется система управления базой данных (СУБД), для которой в прикладном интерфейсе JDBC имеется подходящий драйвер. Среди имеющихся СУБД можно выбрать следующие: IBM DB2, Microsoft SQL Server, MySQL, Oracle или PostgreSQL.

Далее необходимо создать экспериментальную базу данных, например, под названием COREJAVA. Создайте новую базу данных сами или попросите сделать это администратора баз данных, а также наделить вас правами для создания, обновления и удаления таблиц.

Если вам не приходилось раньше устанавливать базу данных с архитектурой "клиент–сервер", то процесс ее установки, конечно, покажется вам очень сложным, а обнаружить причину возможной неудачи будет совсем не просто. Поэтому в таких случаях рекомендуется обратиться к услугам опытных специалистов.

Если у вас отсутствует опыт работы с базами данных, рекомендуется установить сначала базу данных Apache Derby, которая входит в состав большинства версий JDK 7. (Если же Apache Derby не входит в состав используемой вами версии JDK, загрузите эту базу данных по адресу http://db.apache.org/derby.)

НА ЗАМЕТКУ! Версия Apache Derby, входящая в состав JDK, официально называется JavaDB. Во избежание недоразумений в этой главе она будет называться Derby.

Прежде чем вы сможете написать свою первую программу для работы с базой данных, вам придется изучить ряд других вопросов, рассматриваемых в последующих разделах.

URL базы данных

Для установления соединения с базой данных необходимо указать ряд параметров, имеющих отношение к базам данных. К их числу могут относиться имена хостов, номера портов, а также имена баз данных. В JDBC используется синтаксис описания источника данных, подобный обычным URL. Ниже приведены некоторые примеры такого синтаксиса.

```
jdbc:derby://localhost:1527/COREJAVA;create=true
jdbc:postgresql:COREJAVA
```

Эти URL определяют в JDBC базы данных Derby и PostgreSQL по имени `COREJAVA`. Ниже приведена общая синтаксическая форма записи URL в JDBC, где *подчиненный_протокол* обозначает специальный драйвер для соединения с базой данных, а *другие_сведения* имеют формат, который зависит от применяемого подчиненного протокола. По поводу выбора конкретного формата следует обращаться к документации на базу данных.

```
jdbc:подчиненный_протокол:другие_сведения
```

JAR-файлы драйверов

Вам нужно будет также получить JAR-файл, в котором находится драйвер для вашей базы данных. Так, если вы пользуетесь базой данных Derby, вам понадобится файл `derbyclient.jar`. Если же это другая база данных, вам придется найти для нее подходящий драйвер. В частности, драйверы для базы данных PostgreSQL доступны для загрузки по адресу `http://jdbc.postgresql.org`.

При запуске программы, обращающейся к базе данных, в командной строке нужно указать JAR-файл драйвера после параметра **`-classpath`**. (Для компиляции самой программы JAR-файл драйвера не нужен.) Для запуска подобных программ из командной строки воспользуйтесь приведенной ниже командой. В Windows текущий каталог, обозначаемый знаком `.`, отделяется от местонахождения JAR-файла драйвера точкой с запятой.

```
java -classpath путь_к_файлу_драйвера:. имя_программы
```

Запуск базы данных

Прежде чем подключиться к серверу базы данных, его нужно запустить. Подробности этого процесса зависят от конкретной базы данных. В частности, для запуска базы данных Derby выполните следующие действия.

1. Откройте командную оболочку и перейдите в каталог, где будут находиться файлы базы данных.
2. Найдите файл `derbyrun.jar`. В одних версиях JDK он может находиться в каталоге `jdk/db/lib`, а в других — в отдельном установочном каталоге JavaDB. Каталог, содержащий файл `lib/derbyrun.jar`, обозначается здесь и далее как *derby*.
3. Выполните следующую команду:

   ```
   java -jar derby/lib/derbyrun.jar server start
   ```

4. Еще раз проверьте, работает ли база данных должным образом. Создайте файл `ij.properties`, введя в него следующие строки:

```
ij.driver=org.apache.derby.jdbc.ClientDriver
ij.protocol=jdbc:derby://localhost:1527/
ij.database=COREJAVA;create=true
```

5. Запустите в другой копии командной оболочки диалоговое инструментальное средство для написания сценариев базы данных Derby (оно называется `ij`), выполнив следующую команду:

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

6. Теперь вы можете выдать команды SQL, например, следующие:

```
CREATE TABLE Greetings (Message CHAR(20));
INSERT INTO Greetings VALUES ('Hello, World!');
SELECT * FROM Greetings;
DROP TABLE Greetings;
```

7. Обратите внимание на то, что каждая команда должна завершаться точкой с запятой. Чтобы выйти из режима ввода команд SQL, введите команду `EXIT;`.

8. Завершив работу с базой данных, остановите ее сервер, выполнив следующую команду:

```
java -jar derby/lib/derbyrun.jar server shutdown
```

Если вы пользуетесь другой базой данных, вам нужно найти в документации на нее сведения о запуске и остановке сервера базы данных, а также о том, как подключаться к нему и выполнять команды SQL.

Регистрация класса драйвера

Многие JAR-файлы прикладного интерфейса JDBC (например, драйвер базы данных Derby, входящий в состав версии Java SE 7) автоматически регистрируют класс драйвера. В этом случае вы можете пропустить этап ручной регистрации, рассматриваемый в этом разделе. JAR-файл может автоматически зарегистрировать класс драйвера, если он содержит файл `META-INF/services/java.sql.Driver`. Чтобы убедиться в этом, достаточно распаковать JAR-файл драйвера.

НА ЗАМЕТКУ! В механизме регистрации используется малоизвестная часть спецификации JAR (подробнее об этом см. по адресу `http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html#Service%20Provider`). Для драйвера, совместимого с версией JDBC4, автоматическая регистрация обязательна.

Если же JAR-файл драйвера не поддерживает автоматическую регистрацию, вам придется выяснить имена классов драйверов JDBC, используемых поставщиком вашей базы данных. Типичными именами классов драйверов являются следующие:

```
org.apache.derby.jdbc.ClientDriver
org.postgresql.Driver
```

Зарегистрировать драйвер с помощью класса `DriverManager` можно двумя способами. Один способ состоит в том, чтобы загрузить класс драйвера в программу на Java, как показано в приведенной ниже строке кода, где выполняется статический инициализатор, который и осуществляет регистрацию загружаемого драйвера.

```
Class.forName("org.postgresql.Driver"); // принудительно загрузить
                                        // класс драйвера
```

Другой способ состоит в том, чтобы задать свойство `jdbc.drivers`. Это свойство можно указать в качестве аргумента непосредственно в командной строке:

```
java -Djdbc.drivers=org.postgresql.Driver имя_программы
```

С другой стороны, вы можете установить системное свойство в своей прикладной программе, сделав следующий вызов:

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

При необходимости можно также указать несколько разных драйверов, разделив их двоеточием:

```
org.postgresql.Driver:org.apache.derby.jdbc.ClientDriver
```

Соединение с базой данных

Установить соединение с базой данных в своей прикладной программе на Java вы можете так, как показано в приведенном ниже фрагменте кода.

```
String url = "jdbc:postgresql:COREJAVA";
String username = "dbuser";
String password = "secret";
Connection conn = DriverManager.getConnection(url, username, password);
```

Диспетчер перебирает все зарегистрированные драйверы, пытаясь найти тот, который соответствует подчиненному протоколу, указанному в URL базы данных. Метод `getConnection()` возвращает объект типа `Connection`, который используется для выполнения команд SQL. Чтобы соединиться с базой данных, необходимо знать имя пользователя базы данных и пароль.

НА ЗАМЕТКУ! По умолчанию база данных Derby допускает соединение под любым именем пользователя, не проверяя пароль. Для каждого пользователя в этой базе данных формируется отдельный ряд таблиц. Именем пользователя по умолчанию является **app**.

Все сказанное выше о работе с базами данных демонстрируется на примере тестовой программы, исходный код которой приведен в листинге 4.1. Эта программа загружает из файла `database.properties` параметры соединения с базой данных и затем устанавливает его. Файл `database.properties`, предоставляемый вместе с примером кода, содержит сведения о соединении с базой данных Derby. Если вы пользуетесь другой базой данных, введите этот в файл соответствующие сведения о соединении с конкретной базой данных. Ниже в качестве примера приведены параметры соединения с базой данных PostgreSQL.

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

После соединения с базой данных рассматриваемая здесь тестовая программа выполняет следующие команды SQL:

```
CREATE TABLE Greetings (Message CHAR(20))
INSERT INTO Greetings VALUES ('Hello, World!')
SELECT * FROM Greetings
```

В результате выполнения команды SELECT выводится приведенная ниже символьная строка.

```
Hello, World!
```

После этого таблица, созданная в базе данных, удаляется из нее по следующей команде SQL:

```
DROP TABLE Greetings
```

Чтобы выполнить данную тестовую программу, запустите сначала базу данных, как описано выше, а затем саму программу, введя приведенную ниже команду. (Как всегда, пользователям Windows следует ввести точку с запятой (;) вместо двоеточия (:) для разделения составляющих пути к файлу.)

```
java -classpath .:driverJAR test.TestDB
```

СОВЕТ! Для устранения неполадок в JDBC можно активизировать трассировку JDBC. С этой целью вызовите метод `DriverManager.setLogWriter()`, чтобы направить сообщения трассировки в записывающий поток типа `PrintWriter`. Вывод трассировки содержит подробный перечень действий JDBC. В большинстве реализаций драйвера JDBC предоставляются дополнительные механизмы трассировки. Например, для базы данных Derby добавьте параметр `traceFile` в URL прикладного интерфейса JDBC следующим образом:

```
jdbc:derby://localhost:1527/COREJAVA;create=true;traceFile=trace.out
```

Листинг 4.1. Исходный код из файла `test/TestDB.java`

```
1  package test;
2
3  import java.nio.file.*;
4  import java.sql.*;
5  import java.io.*;
6  import java.util.*;
7
8  /**
9   * В этой программе проверяется правильность конфигурирования
10  * базы данных и драйвера JDBC
11  * @version 1.02 2012-06-05
12  * @author Cay Horstmann
13 */
14 public class TestDB
15 {
16   public static void main(String args[]) throws IOException
17   {
18     try
19     {
20       runTest();
21     }
22     catch (SQLException ex)
23     {
24       for (Throwable t : ex)
25       t.printStackTrace();
26     }
27   }
28   /**
29    * Выполняет тест, создавая таблицу, вводя в нее значение,
```

```
30     * отображая содержимое таблицы и, наконец, удаляя ее
31    */
32    public static void runTest() throws SQLException, IOException
33    {
34      try (Connection conn = getConnection())
35      {
36        Statement stat = conn.createStatement();
37
38        stat.executeUpdate("CREATE TABLE Greetings (Message CHAR(20))");
39        stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, World!')");
40
41        try (ResultSet result = stat.executeQuery("SELECT * FROM Greetings"))
42        {
43          if (result.next())
44          System.out.println(result.getString(1));
45        }
46        stat.executeUpdate("DROP TABLE Greetings");
47      }
48    }
49
50    /**
51     * Получает соединение с базой данных из свойств,
52     * определенных в файле database.properties
53     * @return Возвращает соединение с базой данных
54    */
55    public static Connection getConnection() throws SQLException, IOException
56    {
57      Properties props = new Properties();
58      try (InputStream in =
59           Files.newInputStream(Paths.get("database.properties")))
60      {
61        props.load(in);
62      }
63      String drivers = props.getProperty("jdbc.drivers");
64      if (drivers != null) System.setProperty("jdbc.drivers", drivers);
65      String url = props.getProperty("jdbc.url");
66      String username = props.getProperty("jdbc.username");
67      String password = props.getProperty("jdbc.password");
68
69      return DriverManager.getConnection(url, username, password);
70    }
71  }
```

java.sql.DriverManager 1.1

- `static Connection getConnection(String` *url*`, String` *user*`, String` *password*`)`

  Устанавливает соединение с указанной базой данных и возвращает объект типа `Connection`.

Выполнение команд SQL

Для выполнения команды SQL сначала создается объект типа `Statement`. Для этой цели используется объект типа `Connection`, который можно получить, вызвав метод `DriverManager.getConnection()` следующим образом:

```
Statement stat = conn.createStatement();
```

Затем формируется символьная строка с требующейся командой SQL, как показано в приведенном ниже примере кода.

```
String command = "UPDATE Books"
    + " SET Price = Price - 5.00"
    + " WHERE Title NOT LIKE '%Introduction%'";
```

Далее вызывается метод `executeUpdate()` из класса `Statement`:

```
stat.executeUpdate(command);
```

Метод `executeUpdate()` возвращает количество строк, полученных из таблицы базы данных в результате выполнения команды SQL, или же нуль строк для команд, которые не возвращают количество строк из таблицы. Так, в результате приведенного выше вызова метода `executeUpdate()` возвращается количество книг, цена которых снижена на 5 долларов.

Вызывая метод `executeUpdate()`, можно выполнять команды `INSERT`, `UPDATE` и `DELETE`, а также команды определения данных, в том числе `CREATE TABLE` и `DROP TABLE`. Но для выполнения команды `SELECT` необходимо вызвать другой метод, а именно `executeQuery()`. Имеется также универсальный метод `execute()`, с помощью которого можно выполнять произвольные команды SQL, но он применяется в основном для составления запросов в диалоговом режиме.

Если вы составляете запрос базы данных, вас, конечно, интересует результат его обработки. Метод `executeQuery()` возвращает объект типа `ResultSet`, как показано ниже. Этот объект можно использовать для построчного просмотра результатов выполнения запроса.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books")
```

Для анализа результирующего набора организуется приведенный ниже цикл.

```
while (rs.next())
{
  проанализировать строку из результирующего набора
}
```

ВНИМАНИЕ! Порядок последовательной обработки строк в классе `ResultSet` организован несколько иначе, чем в интерфейсе `java.util.Iterator`. В классе `ResultSet` итератор устанавливается на позиции *перед* первой строкой из результирующего набора. Поэтому для его перемещения к первой строке нужно вызвать метод `next()`. Кроме того, в данном классе отсутствует метод `hasNext()`, а следовательно, метод `next()` придется вызывать до тех пор, пока не будет возвращено логическое значение `false`.

Строки располагаются в результирующем наборе в совершено произвольном порядке. Если порядок их следования важен, его необходимо установить с помощью оператора `ORDER BY`. При обработке отдельной строки обычно требуется получить содержимое отдельных полей (или столбцов). Для этой цели имеется целый ряд методов доступа к полям (или столбцам). Ниже приведены некоторые примеры вызова подобных методов доступа.

```
String isbn = rs.getString(1);
double price = rs.getDouble("Price");
```

Для каждого *типа данных* Java предусмотрен отдельный метод доступа, например `getString()` или `getDouble()`. И каждый из них реализован в двух формах:

один — с числовым параметром, другой — со строковым параметром. Если метод доступа вызывается с числовым параметром, данные извлекаются из столбца с указанным номером. Например, в результате вызова метода `rs.getString(1)` возвращается значение из первого столбца текущей строки таблицы.

ВНИМАНИЕ! В отличие от массивов, нумерация столбцов таблиц в базе данных начинается с **1**.

Если же метод доступа вызывается со строковым параметром, данные извлекаются из столбца с указанным именем. Например, в результате вызова метода `rs.getDouble("Price")` возвращается значение из столбца с именем `Price`. Первая форма методов доступа с числовым параметром более эффективна, но строковые параметры улучшают восприятие и упрощают сопровождение кода. Если указанный тип не соответствует фактическому типу, метод доступа автоматически выполняет необходимое преобразование данных. Например, при вызове метода `rs.getString("Price")` числовое значение с плавающей точкой, извлекаемое из столбца `Price`, преобразуется в символьную строку.

java.sql.Connection 1.1

- `Statement createStatement()`

  Создает объект типа `Statement`, который может использоваться для выполнения команд SQL без параметров.

- `void close()`

  Немедленно разрывает текущее соединение и освобождает созданные для него ресурсы JDBC.

java.sql.Statement 1.1

- `ResultSet executeQuery(String` *`sqlQuery`*`)`

  Выполняет команду SQL из указанной символьной строки и возвращает объект типа `ResultSet` с результатами выполнения этой команды.

- `int executeUpdate(String` *`sqlStatement`*`)`

  Выполняет команды SQL типа `INSERT`, `UPDATE` и `DELETE` из указанной символьной строки, а также команды определения данных `CREATE TABLE` и `DROP TABLE`. Возвращает количество строк, обработанных при выполнении данной команды SQL, или нулевое значение, если для данной команды SQL не установлен подсчет обновлений.

- `boolean execute(String` *`sqlStatement`*`)`

  Выполняет команду SQL из указанной символьной строки и возвращает логическое значение `true`, если эта команда предоставляет результирующий набор, а иначе — логическое значение `false`. Может сформировать множество результирующих наборов и подсчетов обновлений. Для доступа к данным, полученным в результате выполнения данной команды SQL, следует вызвать метод `getResultSet()` или `getUpdateCount()`. Вопросы обработки множественных результатов рассматриваются далее в разделе "Множественные результаты".

- `ResultSet getResultSet()`

  Возвращает объект типа `ResultSet` с результатами выполнения предыдущей команды SQL или пустое значение `null`, если выполнение предыдущей команды не дало никаких результатов. Этот метод следует вызывать только один раз для каждой выполняемой команды SQL.

- `int getUpdateCount()`

  Возвращает количество строк, обработанных при выполнении предыдущей команды обновления, или значение **-1**, если для данной команды SQL не установлен подсчет обновлений. Этот метод следует вызывать только один раз для каждой выполняемой команды SQL.

- `void close()`

  Закрывает данную команду SQL и связанные с ней результирующие наборы.

- `boolean isClosed()` **6**

  Возвращает логическое значение `true` при закрытии данной команды SQL.

- `void closeOnCompletion()` **7**

  Закрывает данную команду SQL, как только будут закрыты все связанные с ней результирующие наборы.

java.sql.ResultSet 1.1

- `boolean next()`

  Перемещает указатель текущей строки в результирующем наборе на одну позицию вперед. После прохождения последней строки возвращает логическое значение `false`. Следует иметь в виду, что данный метод нужно непременно вызвать для перемещения указателя к первой строке в результирующем наборе.

- *Xxx* `get`*Xxx*`(int` *columnNumber*`)`
- *Xxx* `get`*Xxx*`(String` *columnLabel*`)`

  (`Xxx` обозначает тип данных, например `int`, `double`, `String`, `Date` и т.д.)

- `<T> T getObject(int` *columnNumber*`, Class<T>` *type*`)` **7**
- `<T> T getObject(String` *columnLabel*`, Class<T>` *type*`)` **7**

  Возвращает значение столбца, задаваемого номером или меткой, с преобразованием в указанный тип данных. Следует иметь в виду, что допускаются не все варианты преобразования типов. Метка столбца — это метка, указываемая в операторе `AS` команды SQL, или имя столбца, если оператор `AS` не используется.

- `int findColumn(String` *columnName*`)`

  Возвращает номер столбца с указанным именем.

- `void close()`

  Немедленно закрывает текущий результирующий набор.

- `boolean isClosed()` **6**

  Возвращает логическое значение `true` при закрытии данной команды.

Управление соединениями, командами и результирующими наборами

Каждый объект типа `Connection` может создать один или несколько объектов типа `Statement`. Один и тот же объект типа `Statement` можно использовать для выполнения нескольких не связанных между собой команд и запросов. Но для такого объекта допускается наличие *не более одного* открытого результирующего набора. Если же требуется выполнить несколько команд и одновременно проанализировать их результаты, то для этого понадобится несколько объектов типа `Statement`.

Не следует, однако, забывать, что, по крайней мере, одна широко употребляемая база данных (Microsoft SQL Server) взаимодействует с драйвером JDBC, допускающим одновременную активизацию только одного объекта типа `Statement`. Количество открытых объектов типа `Statement`, одновременно поддерживаемых драйвером JDBC, можно выяснить, вызвав метод `getMaxStatements()` из класса `DatabaseMetaData`.

На первый взгляд, подобное ограничение кажется излишним, но на практике дело редко доходит до одновременной обработки нескольких результирующих наборов. Если же результирующие наборы связаны друг с другом, то можно всегда сформировать составной запрос и проанализировать единственный результирующий набор. Намного выгоднее предоставить базе данных возможность самостоятельно объединить запросы, чем циклически обрабатывать несколько результирующих наборов в программе на Java.

Покончив дело с объектом типа `ResultSet`, `Statement` или `Connection`, следует как можно скорее вызвать метод `close()`. Ведь эти объекты используют крупные структуры данных и истощаемые ресурсы на сервере базы данных. Метод `close()` из класса `Statement` автоматически закрывает результирующий набор, связанный с объектом данного класса, если, конечно, этот набор открыт при выполнении соответствующей команды. Аналогично метод `close()` из класса `Connection` закрывает все объекты данного класса, открытые для соединения с базой данных. С другой стороны, в версии Java SE 7 появилась возможность вызвать метод `closeOnCompletion()` из класса `Statement`, чтобы автоматически закрыть объект данного класса, как только будут закрыты все связанные с ним результирующие наборы.

Если соединение установлено кратковременно, то можно и не беспокоиться о закрытии объектов типа `Statement` и связанных с ними результирующих наборов. А для абсолютной гарантии, что объект соединения с базой данных не останется открытым, можно воспользоваться оператором `try` с ресурсами следующим образом:

```
try (Connection conn = . . .)
{
  Statement stat = conn.createStatement();
  ResultSet result = stat.executeQuery(queryString);
  обработать результат запроса
}
```

СОВЕТ. Оператор `try` с ресурсами рекомендуется применять только для разрыва соединения с базой данных, а для обработки исключений — отдельный блок `try/catch`. Благодаря такому разделению блоков исходный код легче читать и сопровождать.

Анализ исключений SQL

У каждого исключения SQL имеется цепочка объектов типа `SQLException`, которые извлекаются методом `getNextException()`. Эта цепочка исключений является

дополнением "причинной" цепочки объектов типа Throwable, имеющихся в каждом исключении. (Подробнее об исключениях в Java см. в главе 11 первого тома данной книги.) Чтобы полностью перечислить все исключения, пришлось бы организовать два вложенных цикла. К счастью, в версии Java SE 6 был усовершенствован класс SQLException для реализации интерфейса Iterable<Throwable>. В частности, метод iterator() производит объект типа Iterator<Throwable>, который осуществляет перебор в обеих цепочках, сначала проходя по "причинной" цепочке первого объекта типа SQLException, а затем переходя к следующему объекту типа SQLException и т.д. Для этой цели можно организовать усовершенствованный цикл for следующим образом:

```
for (Throwable t : sqlException)
{
  сделать что-нибудь с объектом t
}
```

Чтобы продолжить анализ объекта типа SQLException, для него можно вызвать методы getSQLState() и getErrorCode(). Первый метод выдает символьную строку по стандарту X/Open или SQL:2003. (Для того чтобы выяснить, какому именно стандарту соответствует применяемый драйвер, достаточно вызвать метод getSQLStateType() из интерфейса DatabaseMetadata.) Что же касается кода ошибки, то у различных поставщиков баз данных он разный.

Исключения SQL организованы в виде древовидной структуры наследования, приведенной на рис. 4.5. Благодаря этому имеется возможность перехватывать отдельные типы ошибок независимо от предпочтений поставщиков баз данных.

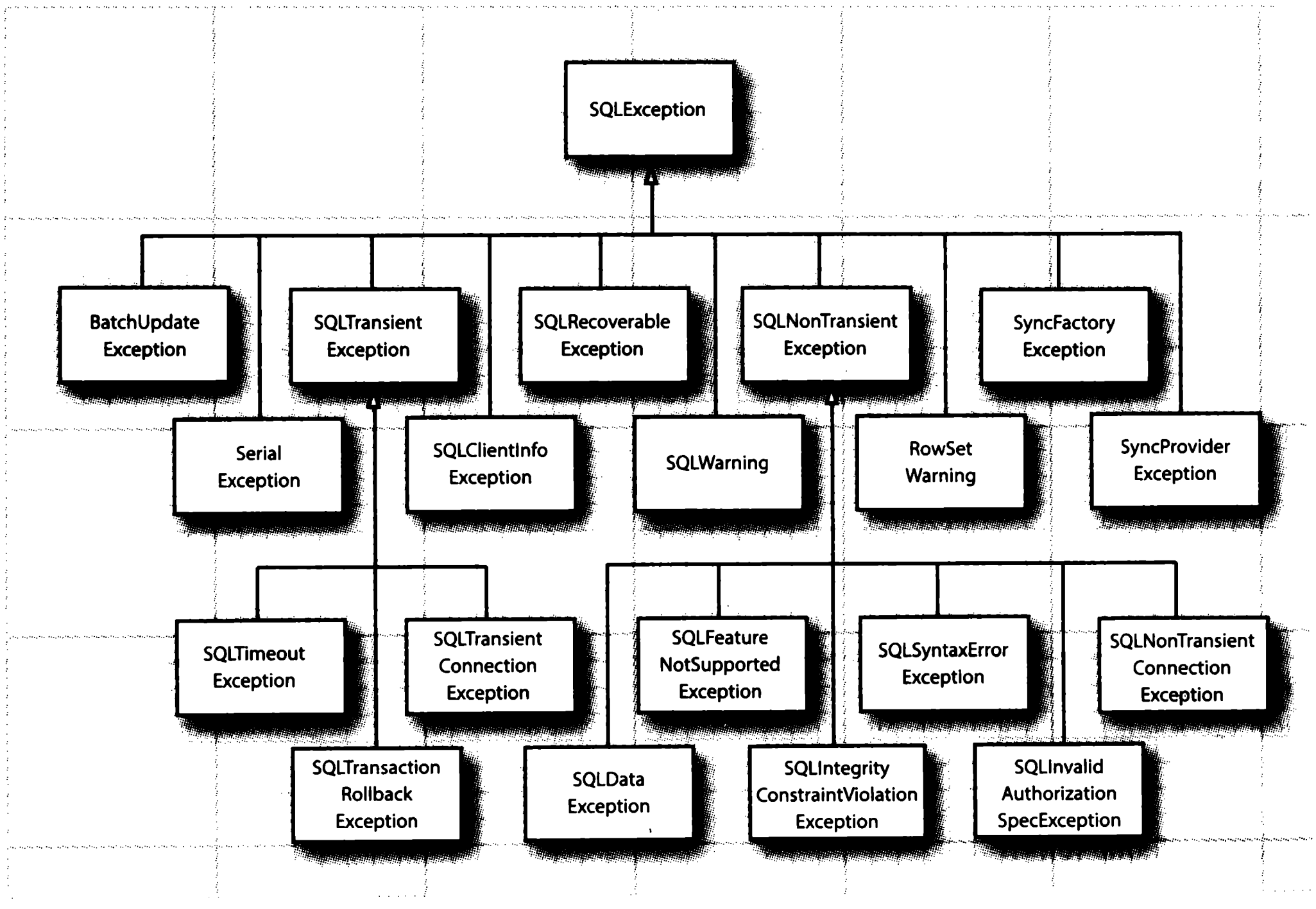


Рис. 4.5. Типы исключений SQL

Кроме того, драйвер базы данных может сообщать о некритичных ситуациях в виде предупреждений. Подобные предупреждения можно получать от соединений, команд и результирующих наборов. Класс `SQLWarning` является подклассом, производным от класса `SQLException`, несмотря на то, что объект типа `SQLWarning` не генерируется в виде исключения. Вызвав методы `getSQLState()` и `getErrorCode()`, можно получить дополнительные сведения о предупреждениях. Подобно исключениям SQL, предупреждения организуются в цепочку. И чтобы получить все предупреждения, потребуется следующий цикл:

```
SQLWarning w = stat.getWarning();
while (w != null)
{
  сделать что-нибудь с объектом w
  w = w.nextWarning();
}
```

Подкласс `DataTruncation`, производный от класса `SQLWarning`, используется в тех случаях, когда данные считываются из базы данных и в этот момент происходит неожиданное их усечение. Если усечение данных произошло при выполнении команды обновления, то объект типа `DataTruncation` генерируется в виде исключения.

java.sql.SQLException 1.1

- `SQLException getNextException()`
- Получает исключение SQL, следующее по цепочке за данным, или пустое значение `null`, если достигнут конец цепочки.
- `Iterator<Throwable> iterator()` **6**
- Получает итератор, перебирающий по цепочке исключения SQL и их причины.
- `String getSQLState()`
- Получает стандартный код ошибки, обозначающий состояние SQL.
- `int getErrorCode()`
- Получает код ошибки, характерный для поставщика используемой базы данных.

java.sql.SQLWarning 1.1

- `SQLWarning getNextWarning()`
- Получает предупреждение, следующее по цепочке за данным, или пустое значение `null`, если достигнут конец цепочки.

***java.sql.Connection* 1.1**
***java.sql.Statement* 1.1**
***java.sql.ResultSet* 1.1**

- `QLWarning getWarnings()`
- `SQLWarning getWarnings()`
- Возвращают первое ожидающее предупреждение или пустое значение `null`, если ожидающие предупреждения отсутствуют.

java.sql.DataTruncation 1.1

- `boolean getParameter()`
- Возвращает логическое значение `true`, если усечение данных применяется к параметру, или логическое значение `false`, если оно применяется к столбцу.
- `int getIndex()`
- Возвращает индекс усеченного параметра или столбца.
- `int getDataSize()`
- Возвращает количество байт, которые необходимо передать, или значение **-1**, если извлекаемое значение неизвестно.
- `int getTransferSize()`
- Возвращает количество байт, которые были фактически переданы, или значение **-1**, если извлекаемое значение неизвестно.

Заполнение базы данных

Попробуем теперь написать первую программу, используя прикладной интерфейс JDBC. Конечно, было бы неплохо, если бы в этой программе можно было выполнить некоторые из рассмотренных выше запросов. Но, к сожалению, это невозможно, потому что база данных пуста. Сначала ее нужно заполнить данными, хотя сделать это нетрудно с помощью команд SQL для создания таблиц и ввода в них данных. Большинство СУБД способны обрабатывать команды SQL из текстового файла, но в этом случае проявляются досадные отличия в завершающих символах операторов и другие синтаксические особенности реализации SQL на разных платформах.

В силу этих причин воспользуемся прикладным интерфейсом JDBC, чтобы создать простую программу для построчного чтения команд SQL из файла и последующего их выполнения. В частности, рассматриваемая здесь программа должна читать данные из текстового файла в следующем формате:

```
CREATE TABLE Publishers (Publisher_Id CHAR(6), Name CHAR(30), URL CHAR(80));
INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley', 'www.aw-bc.com');
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons', 'www.wiley.com');
. . .
```

В листинге 4.2 приведен исходный код программы `ExecSQL`, считывающей команды SQL из текстового файла и затем выполняющей их. Для применения этой программы совсем не обязательно разбираться в ее исходном коде. Самое главное, что она позволяет заполнить базу данных и выполнить примеры программ в оставшейся части этой главы.

Прежде всего убедитесь в том, что сервер базы данных работает нормально, и запустите программу `ExecSQL` на выполнение, введя следующие команды:

```
java -classpath путь_к_драйверу:. exec.ExecSQL Books.sql
java -classpath путь_к_драйверу:. exec.ExecSQL Authors.sql
java -classpath путь_к_драйверу:. exec.ExecSQL Publishers.sql
java -classpath путь_к_драйверу:. exec.ExecSQL BooksAuthors.sql
```

Перед запуском данной программы непременно проверьте содержимое файла свойств `database.properties` на соответствие вашей среде исполнения (см. раздел "Соединение с базой данных" ранее в этой главе).

НА ЗАМЕТКУ! В состав вашей базы данных может входить утилита для непосредственного чтения файлов SQL. Например, в базе данных Derby можно выполнить приведенную ниже команду для чтения файла свойств `ij.properties`, описанного ранее в разделе "Запуск базы данных".

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties Books.sql
```

В формате данных для программы `ExecSQL` допускается вставка точки с запятой в конце каждой строки, поскольку такой формат предполагается в большинстве утилит баз данных.

Ниже вкратце описываются основные этапы выполнения программы `ExecSQL`.

1. Устанавливается соединение с базой данных. Метод `getConnection()` считывает содержимое файла свойств `database.properties` и вводит свойство `jdbc.drivers` в список системных свойств. Диспетчер драйверов использует свойство `jdbc.drivers` для загрузки соответствующего драйвера базы данных. Для установления соединения с базой данных в методе `getConnection()` применяются свойства `jdbc.url`, `jdbc.username` и `jdbc.password`.
2. Открывается текстовый файл с командами SQL. Если такой файл отсутствует, пользователю предлагается ввести команды вручную с консоли.
3. Все заданные команды SQL выполняются с помощью универсального метода `execute()`. При получении результирующего набора этот метод возвращает логическое значение `true`. В конце всех четырех текстовых файлов с командами SQL содержится команда `SELECT *`. Это дает возможность убедиться, что данные успешно введены в базу данных.
4. При наличии результирующего набора полученные результаты выводятся на экран. А поскольку это обобщенный результирующий набор, то для определения количества столбцов в нем потребуются метаданные. Более подробно метаданные рассматриваются в соответствующем разделе далее в этой главе.
5. Если при выполнении команд SQL возникает какое-нибудь исключение, то выводятся сведения об этом исключении, а также обо всех остальных исключениях, которые могут следовать по цепочке.
6. По завершении всех заданных команд SQL соединение с базой данных разрывается.

Листинг 4.2. Исходный код из файла `exec/ExecSQL.java`

```
1   package exec;
2
3   import java.io.*;
4   import java.nio.file.*;
5   import java.util.*;
6   import java.sql.*;
7
8   /**
9    * Эта программа выполняет все команды SQL из текстового файла.
10   * Она вызывается следующим образом:
11   * java -classpath путь_к_драйверу:. ExecSQL commandFile
12   * @version 1.31 2012-06-05
13   * @author Cay Horstmann
14  */
```

```
15  class ExecSQL
16  {
17    public static void main(String args[]) throws IOException
18    {
19      try
20      {
21        Scanner in = args.length == 0 ? new Scanner(System.in) :
22            new Scanner(Paths.get(args[0]));
23        try (Connection conn = getConnection())
24        {
25          Statement stat = conn.createStatement();
26
27          while (true)
28          {
29            if (args.length == 0)
30                System.out.println("Enter command or EXIT to exit:");
31            if (!in.hasNextLine()) return;
32            String line = in.nextLine();
33            if (line.equalsIgnoreCase("EXIT")) return;
34            if (line.trim().endsWith(";")) // удалить точку с запятой
35                                           // в конце строки
36            {
37              line = line.trim();
38              line = line.substring(0, line.length() - 1);
39            }
40            try
41            {
42              boolean isResult = stat.execute(line);
43              if (isResult)
44              {
45                ResultSet rs = stat.getResultSet();
46                showResultSet(rs);
47              }
48              else
49              {
50                int updateCount = stat.getUpdateCount();
51                System.out.println(updateCount + " rows updated");
52              }
53            }
54            catch (SQLException ex)
55            {
56              for (Throwable e : ex)
57                e.printStackTrace();
58            }
59          }
60        }
61      }
62      catch (SQLException e)
63      {
64        for (Throwable t : e)
65        t.printStackTrace();
66      }
67    }
68
69    /**
70     * Получает соединение с базой данных из свойств,
71     * определенных в файле database.properties
72     * @return Возвращает соединение с базой данных
73    */
```

```
74    public static Connection getConnection()
75          throws SQLException, IOException
76    {
77      Properties props = new Properties();
78      try (InputStream in =
79           Files.newInputStream(Paths.get("database.properties")))
80      {
81        props.load(in);
82      }
83
84      String drivers = props.getProperty("jdbc.drivers");
85      if (drivers != null) System.setProperty("jdbc.drivers", drivers);
86
87      String url = props.getProperty("jdbc.url");
88      String username = props.getProperty("jdbc.username");
89      String password = props.getProperty("jdbc.password");
90
91      return DriverManager.getConnection(url, username, password);
92    }
93
94    /**
95     * Выводит результирующий набор
96     * @param result Выводимый результирующий набор
97    */
98    public static void showResultSet(ResultSet result) throws SQLException
99    {
100     ResultSetMetaData metaData = result.getMetaData();
101     int columnCount = metaData.getColumnCount();
102
103     for (int i = 1; i <= columnCount; i++)
104     {
105       if (i > 1) System.out.print(", ");
106       System.out.print(metaData.getColumnLabel(i));
107     }
108     System.out.println();
109
110     while (result.next())
111     {
112       for (int i = 1; i <= columnCount; i++)
113       {
114         if (i > 1) System.out.print(", ");
115         System.out.print(result.getString(i));
116       }
117       System.out.println();
118     }
119   }
120 }
```

Выполнение запросов

В этом разделе рассматривается пример программы, способной выполнять запросы базы данных COREJAVA. Для нормальной работы этой программы в базе данных нужно создать таблицы, описанные в предыдущем разделе. При составлении запроса базы данных можно выбрать автора книги и издательство или же оставить критерий отбора книги независимо от автора или издательства.

Рассматриваемая здесь программа позволяет также вносить изменения в содержимое базы данных. Для этого достаточно выбрать издательство и ввести сумму. Все цены на книги данного издательства автоматически откорректируются по введенной сумме, а программа отобразит количество измененных строк в таблице. После подобной коррекции цен на книги можно выполнить запрос, чтобы проверить новые цены.

Подготовленные операторы и запросы

В рассматриваемой здесь программе используется новое средство: *подготовленные операторы*. Рассмотрим следующий запрос SQL на выборку всех книг отдельного издательства независимо от их авторов:

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = название издательства, выбираемое из списка
```

Вместо создания отдельной команды SQL для каждого пользовательского запроса следует предварительно *подготовить* запрос с главной переменной и многократно использовать его, меняя только значение этой переменной. Такая возможность существенно повышает эффективность работы программы. Перед обработкой каждого запроса СУБД вырабатывает план его эффективного исполнения. Подготавливая запрос для последующего многократного применения, можно исключить повторное планирование его выполнения.

Каждая главная переменная в запросе обозначается знаком вопроса (?). Если в запросе используются несколько главных переменных, нужно внимательно следить за их расстановкой с помощью знаков вопроса, чтобы правильно устанавливать их конкретные значения. Ниже показано, как выглядит предварительно подготовленный запрос в исходном коде рассматриваемой здесь программы.

```
String publisherQuery =
  "SELECT Books.Price, Books.Title" +
  " FROM Books, Publishers" +
  " WHERE Books.Publisher_Id = Publishers.Publisher_Id AND Publishers.Name = ?";
PreparedStatement stat = conn.prepareStatement(publisherQuery);
```

Перед выполнением подготовленного оператора запроса необходимо связать главные переменные с их конкретными значениями, вызвав метод `set()`. Подобно разным формам метода `get()` из класса `ResultSet`, для разных типов данных предусмотрены разные формы метода `set()`. В качестве примера ниже показано, каким образом задается строковое значение с названием издательства.

```
stat.setString(1, publisher);
```

Первый аргумент этого метода обозначает номер позиции главной переменной, обозначаемой знаком вопроса в подготовленном операторе, а второй аргумент — ее конкретное значение.

При повторном использовании подготовленного запроса с несколькими главными переменными все их привязки к конкретным значениям остаются в силе, если только они не изменены с помощью метода `set()` или `clearParameters()`. Это означает, что метод `setXxx()`, где *Xxx* — тип данных, нужно вызывать только для тех главных переменных, которые изменяются в последующих запросах.

После привязки главных переменных к их конкретным значениям можно приступить к выполнению подготовленного оператора следующим образом:

```
ResultSet rs = stat.executeQuery();
```

СОВЕТ. Составление запроса вручную путем сцепления символьных строк — довольно трудоемкое, чреватое ошибками и небезопасное занятие. Ведь в этом случае нужно позаботиться об обозначении специальных символов (например, кавычек). А если при составлении запроса предполагается ввод пользователем данных, то нужно принять меры защиты от внесения злонамеренного кода для совершения атаки на сервер с базой данных. В этом отношении подготовленные операторы оказываются намного более удобными, и поэтому их рекомендуется применять всякий раз, когда в запрос включаются переменные.

Обновление цены осуществляется по команде `UPDATE`. Обратите внимание на то, что для этого вызывается метод `executeUpdate()`, а не `executeQuery()`. Дело в том, что команда `UPDATE` не возвращает результирующий набор, который в данном случае не нужен. Метод `executeUpdate()` возвращает лишь подсчет количества измененных строк в таблице, как показано ниже.

```
int r = stat.executeUpdate();
System.out.println(r + " rows updated");
```

НА ЗАМЕТКУ! Объект типа `PreparedStatement` становится недействительным после того, как связанный с ним объект типа `Connection` закрывается. Но многие драйверы баз данных автоматически *кешируют* подготовленные операторы. Если один и тот же запрос подготавливается дважды, то в СУБД просто еще раз используется план его выполнения. Поэтому, вызывая метод `prepareStatement()`, можно не особенно беспокоиться об издержках на выполнение подготовленных операторов.

Ниже вкратце описываются действия, выполняемые в рассматриваемом здесь примере программы.

- Списочные массивы заполняются именами авторов и названиями издательств с помощью двух запросов, возвращающих все имена авторов и названия издательств, сохраняемые в базе данных.
- Запросы по имени автора имеют более сложную структуру. Ведь у одной книги может быть несколько авторов, и поэтому в таблице `BooksAuthors` сохраняется соответствие авторов и книг. Допустим, у книги с ISBN 0-201-96426-0 два автора с кодами `DATE` и `DARW`. Для отражения этого факта таблица `BooksAuthors` должна содержать следующие строки:

  ```
  0-201-96426-0, DATE, 1
  0-201-96426-0, DARW, 2
  ```

- В третьем столбце указаны порядковые номера авторов. (Для этой цели нельзя использовать сведения о расположении строк в таблице, поскольку в реляционной базе данных порядок следования записей не фиксирован.) Таким образом, в составляемом запросе следует сначала соединить таблицы `Books`, `BooksAuthors` и `Authors`, а затем сравнить в них имя автора с тем, что указано пользователем, как показано ниже.

  ```
  SELECT Books.Price, Books.Title FROM Books, BooksAuthors, Authors,
  Publishers
  WHERE Authors.Author_Id =
  BooksAuthors.Author_Id AND BooksAuthors.ISBN = Books.ISBN
  AND Books.Publisher_Id =
  Publishers.Publisher_Id AND Authors.Name = ? AND Publishers.Name = ?
  ```

НА ЗАМЕТКУ! Некоторые программирующие на Java стараются избегать составления столь сложных запросов SQL. Как ни странно, они выбирают обходной, но неэффективный путь, предполагающий написание немалого объема кода на Java для последовательной обработки нескольких результирующих наборов. Следует, однако, иметь в виду, что СУБД выполняет запросы *намного* эффективнее, чем программа на Java, поскольку СУБД именно для этого и предназначена. В этой связи рекомендуется взять на вооружение следующее эмпирическое правило: то, что можно сделать средствами SQL, нецелесообразно делать средствами Java.

- Метод `changePrices()` выполняет команду `UPDATE`. В этой команде оператору `WHERE` требуется *код* издательства, а известно лишь его *название*. Это затруднение разрешается с помощью вложенного запроса, как выделено ниже полужирным.

```
UPDATE Books
SET Price = Price + ?
WHERE Books.Publisher_Id =
   (SELECT Publisher_Id FROM Publishers WHERE Name = ?)
```

Весь исходный код данной программы приведен в листинге 4.3.

Листинг 4.3. Исходный код из файла `query/QueryTest.java`

```
1   package query;
2
3   import java.io.*;
4   import java.nio.file.*;
5   import java.sql.*;
6   import java.util.*;
7
8   /**
9    * В этой программе демонстрируется ряд сложных запросов базы данных
10   * @version 1.30 2012-06-05
11   * @author Cay Horstmann
12  */
13  public class QueryTest
14  {
15     private static final String allQuery =
16        "SELECT Books.Price, Books.Title FROM Books";
17
18     private static final String authorPublisherQuery =
19        "SELECT Books.Price, Books.Title"
20        + " FROM Books, BooksAuthors, Authors, Publishers"
21        + " WHERE Authors.Author_Id = BooksAuthors.Author_Id
22        AND BooksAuthors.ISBN = Books.ISBN"
23        + " AND Books.Publisher_Id = Publishers.Publisher_Id
24        AND Authors.Name = ?" + " AND Publishers.Name = ?";
25
26     private static final String authorQuery =
27        "SELECT Books.Price, Books.Title FROM Books, BooksAuthors, Authors"
28        + " WHERE Authors.Author_Id = BooksAuthors.Author_Id
29        AND BooksAuthors.ISBN = Books.ISBN"
30        + " AND Authors.Name = ?";
31
32     private static final String publisherQuery =
```

```
33          "SELECT Books.Price, Books.Title FROM Books, Publishers"
35          + " WHERE Books.Publisher_Id = Publishers.Publisher_Id
36          AND Publishers.Name = ?";
37
38      private static final String priceUpdate =
39          "UPDATE Books " + "SET Price = Price + ? "
40           + " WHERE Books.Publisher_Id =
41           (SELECT Publisher_Id FROM Publishers WHERE Name = ?)";
42
43      private static Scanner in;
44      private static Connection conn;
45      private static ArrayList<String> authors = new ArrayList<>();
46      private static ArrayList<String> publishers = new ArrayList<>();
47
48      public static void main(String[] args) throws IOException
49      {
50        try
51        {
52          conn = getConnection();
53          in = new Scanner(System.in);
54          authors.add("Any");
55          publishers.add("Any");
56          try (Statement stat = conn.createStatement())
57          {
58            // заполнить списочный массив именами авторов книг
59            String query = "SELECT Name FROM Authors";
60            try (ResultSet rs = stat.executeQuery(query))
61            {
62              while (rs.next())
63                authors.add(rs.getString(1));
64            }
65            // заполнить списочный массив названиями издательств
66            query = "SELECT Name FROM Publishers";
67            try (ResultSet rs = stat.executeQuery(query))
68            {
69              while (rs.next())
70                publishers.add(rs.getString(1));
71            }
72          }
73          boolean done = false;
74          while (!done)
75          {
76            System.out.print("Q)uery C)hange prices E)xit: ");
77            String input = in.next().toUpperCase();
78            if (input.equals("Q"))
79              executeQuery();
80            else if (input.equals("C"))
81              changePrices();
82            else
83              done = true;
84          }
85        }
86        catch (SQLException e)
87        {
88          for (Throwable t : e)
89            System.out.println(t.getMessage());
```

```
90        }
91     }
92
93     /**
94      * Выполняет выбранный запрос
95     */
96     private static void executeQuery() throws SQLException
97     {
98        String author = select("Authors:", authors);
99        String publisher = select("Publishers:", publishers);
100       PreparedStatement stat;
101       if (!author.equals("Any") && !publisher.equals("Any"))
102       {
103          stat = conn.prepareStatement(authorPublisherQuery);
104          stat.setString(1, author);
105          stat.setString(2, publisher);
106       }
107       else if (!author.equals("Any") && publisher.equals("Any"))
108       {
109          stat = conn.prepareStatement(authorQuery);
110          stat.setString(1, author);
111       }
112       else if (author.equals("Any") && !publisher.equals("Any"))
113       {
114          stat = conn.prepareStatement(publisherQuery);
115          stat.setString(1, publisher);
116       }
117       else
118          stat = conn.prepareStatement(allQuery);
119
120       try (ResultSet rs = stat.executeQuery())
121       {
122          while (rs.next())
123             System.out.println(rs.getString(1) + ", " + rs.getString(2));
124       }
125    }
126
127    /**
128     * Выполняет команду обновления с целью изменить цены на книги
129    */
130    public static void changePrices() throws SQLException
131    {
132       String publisher =
133          select("Publishers:", publishers.subList(1, publishers.size()));
134       System.out.print("Change prices by: ");
135       double priceChange = in.nextDouble();
136       PreparedStatement stat = conn.prepareStatement(priceUpdate);
137       stat.setDouble(1, priceChange);
138       stat.setString(2, publisher);
139       int r = stat.executeUpdate();
140       System.out.println(r + " records updated.");
141    }
142
143    /**
144     * Предлагает пользователю выбрать символьную строку
145     * @param prompt Отображаемое приглашение
```

```
146    * @param options Варианты выбора, предлагаемые пользователю
147    * @return Возвращает выбранный пользователем вариант
148   */
149   public static String select(String prompt, List<String> options)
150   {
151     while (true)
152     {
153       System.out.println(prompt);
154       for (int i = 0; i < options.size(); i++)
155         System.out.printf("%2d) %s%n", i + 1, options.get(i));
156       int sel = in.nextInt();
157       if (sel > 0 && sel <= options.size())
158         return options.get(sel - 1);
159     }
160   }
161
162   /**
163    * Получает соединение с базой данных из свойств,
164    * определенных в файле database.properties
165    * @return Возвращает соединение с базой данных
166   */
167   public static Connection getConnection()
168         throws SQLException, IOException
169   {
170     Properties props = new Properties();
171     try (InputStream in =
172         Files.newInputStream(Paths.get("database.properties")))
173     {
174       props.load(in);
175     }
176
177     String drivers = props.getProperty("jdbc.drivers");
178     if (drivers != null) System.setProperty("jdbc.drivers", drivers);
179     String url = props.getProperty("jdbc.url");
180     String username = props.getProperty("jdbc.username");
181     String password = props.getProperty("jdbc.password");
182
183     return DriverManager.getConnection(url, username, password);
184   }
185 }
```

java.sql.Connection 1.1

- `PreparedStatement prepareStatement(String sql)`

  Возвращает объект типа `PreparedStatement`, содержащий подготовленный оператор. Строка *sql* содержит оператор SQL с одной или несколькими главными переменными, обозначенными вопросительными знаками.

java.sql.PreparedStatement 1.1

- `void setXxx(int n, Xxx x)`

  (`Xxx` обозначает тип данных, например `int`, `double`, `String`, `Date` и т.д.)

  Задает значение `x` для *n*-го параметра.

- `void clearParameters()`

  Очищает все текущие параметры в подготовленном операторе.

- `ResultSet executeQuery()`

  Выполняет подготовленный запрос SQL и возвращает объект типа `ResultSet`.

- `int executeUpdate()`

  Выполняет команды `INSERT`, `UPDATE` или `DELETE`, представленные в объекте типа `PreparedStatement` в виде подготовленных операторов SQL. Возвращает количество обработанных строк или нулевое значение для таких команд типа DDL, как `CREATE TABLE`.

Чтение и запись больших объектов

Помимо чисел, символьных строк и дат, во многих базах данных можно сохранять *большие объекты* (LOB), к числу которых относятся изображения и другие данные. В языке SQL понятие больших объектов разделяется на категории больших двоичных объектов (BLOB) и больших символьных объектов (CLOB).

Чтобы прочитать большой объект, нужно сначала выполнить команду `SELECT`, а затем вызвать метод `getBlob()` или `getClob()` из класса `ResultSet`. В результате будет получен объект типа `Blob` или `Clob`. А для того чтобы получить двоичные данные из объекта типа `Blob`, следует вызвать метод `getBytes()` или `getInputStream()`. Так, если имеется таблица с изображениями на книжных обложках, то такое изображение можно получить следующим образом:

```
PreparedStatement stat =
    conn.prepareStatement("SELECT Cover FROM BookCovers WHERE ISBN=?");
stat.set(1, isbn);
ResultSet result = stat.executeQuery();
if (result.next())
{
  Blob coverBlob = result.getBlob(1);
  Image coverImage = ImageIO.read(coverBlob.getBinaryStream());
}
```

Аналогично, если извлечь объект типа `Clob`, то из него можно получить символьные данные, вызвав метод `String()` или `getCharacterStream()`.

Чтобы разместить большой объект в базе данных, следует вызвать метод `createLOB()` или `createClob()` для объекта типа `Connection`, получить поток вывода или поток записи для большого объекта, записать данные и сохранить этот объект в базе данных. В качестве примера ниже показано, как сохранить изображение в базе данных.

```
Blob coverBlob = connection.createBlob();
int offset = 0;
OutputStream out = coverBlob.setBinaryStream(offset);
ImageIO.write(coverImage, "PNG", out);
PreparedStatement stat =
    conn.prepareStatement("INSERT INTO Cover VALUES (?, ?)");
```

```
stat.set(1, isbn);
stat.set(2, coverBlob);
stat.executeUpdate();
```

java.sql.ResultSet 1.1

- `Blob getBlob(int columnIndex)` **1.2**
- `Blob getBlob(String columnLabel)` **1.2**
- `Clob getClob(int columnIndex)` **1.2**
- `Clob getClob(String columnLabel)` **1.2**

  Получают большой двоичный объект (BLOB) или большой символьный объект (CLOB) из заданного столбца таблицы.

java.sql.Blob 1.2

- `long length()`

  Получает длину данного большого двоичного объекта.

- `byte[] getBytes(long startPosition, long length)`

  Получает данные в указанных пределах из текущего большого двоичного объекта.

- `InputStream getBinaryStream()`
- `InputStream getBinaryStream(long startPosition, long length)`

  Возвращают поток ввода для чтения данных из текущего большого двоичного объекта полностью или в указанных пределах.

- `OutputStream setBinaryStream(long startPosition)` **1.4**

  Возвращает поток вывода для записи данных в текущий большой двоичный объект, начиная с указанной позиции.

java.sql.Clob 1.4

- `long length()`

  Получает количество символов в данном большом символьном объекте.

- `String getSubString(long startPosition, long length)`

  Получает символы из текущего большого двоичного объекта в указанных пределах.

- `Reader getCharacterStream()`
- `Reader getCharacterStream(long startPosition, long length)`

  Возвращают считывающий поток (а не поток ввода) для чтения символов из данного большого символьного объекта в указанных пределах.

- `Writer setCharacterStream(long startPosition)` **1.4**

  Возвращает поток записи (а не поток вывода) для записи символов в данный большой символьный объект, начиная с указанной позиции.

java.sql.Connection 1.1

- `Blob createBlob()` **6**
- `Clob createClob()` **6**

  Создает пустой большой двоичный объект (BLOB) или большой символьный объект (CLOB).

Синтаксис переходов в SQL

Синтаксис переходов предоставляет средства, которые обычно поддерживаются базами данных, но в разных вариантах в зависимости от конкретного синтаксиса базы данных. А в задачу драйвера JDBC входит преобразование синтаксиса переходов в синтаксис конкретной базы данных.

Переходы предусмотрены для следующих средств.

- Литералы времени и даты.
- Вызовы скалярных функций.
- Вызовы хранимых процедур.
- Внешние соединения.
- Символы перехода в операторах `LIKE`.

Литералы даты и времени сильно отличаются в разных базах данных. Чтобы вставить литерал даты или времени, нужно определить его значение в формате ISO 8601 (`http://www.cl.cam.ac.uk/~mgk25/iso-time.html`). После этого драйвер преобразует литерал в собственный формат базы данных. Для значений типа `DATE`, `TIME` или `TIMESTAMP` используются литералы `d`, `t` и `ts` следующим образом:

```
{d '2008-01-24'}
{t '23:59:59'}
{ts '2008-01-24 23:59:59.999'}
```

Скалярной называется такая функция, которая возвращает одно значение. В базах данных применяется немало скалярных функций, но под разными именами. В спецификации JDBC указаны стандартные имена, преобразуемые в имена, специфические для баз данных. Чтобы вызвать скалярную функцию, следует вставить ее стандартное имя и аргументы, как показано ниже. Полный список поддерживаемых имен скалярных функций можно найти в спецификации JDBC.

```
{fn left(?, 20)}
{fn user()}
```

Хранимой называется такая процедура, которая выполняется в базе данных и написана на языке, специфичном для конкретной базы данных. Для вызова хранимой процедуры служит переход `call`. Если у процедуры отсутствуют параметры, то указывать скобки не нужно. А для фиксации возвращаемого значения служит знак равенства. Ниже показано, каким образом вызываются хранимые процедуры.

```
{call PROC1(?, ?)}
{call PROC2}
{call ? = PROC3(?)}
```

Внешнее соединение двух таблиц не требует, чтобы строки из каждой таблицы совпадали по условию соединения. Например, в приведенном ниже запросе

указаны книги, для которых столбец Publisher_Id не имеет совпадений в таблице Publishers, причем пустые значения NULL обозначают отсутствие совпадений.

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers ON Books.Publisher_Id =
    Publisher.Publisher_Id}
```

Чтобы включить в запрос издательства без совпадающих книг, может потребоваться оператор RIGHT OUTER JOIN, а чтобы возвратить по запросу и то и другое, — оператор FULL OUTER JOIN. Синтаксис переходов требуется именно потому, что не во всех базах данных используется стандартное обозначение внешних соединений.

И наконец, знаки _ и % имеют специальное значение в операторе LIKE, обозначая совпадение с одним символом или последовательностью символов. Стандартного способа их буквального использования не существует. Так, для сопоставления всех символьных строк, содержащих знак _, можно воспользоваться приведенной ниже конструкцией, где знак ! определен как символ перехода, а комбинация знаков !_ буквально обозначает знак подчеркивания.

```
... WHERE ? LIKE %!_% {escape '!'}
```

Множественные результаты

По запросу могут быть возвращены множественные результаты. Это может произойти при выполнении хранимой процедуры или в базах данных, которые допускают также выполнение многих команд SELECT в одном запросе. Ниже поясняется, как получить все результирующие наборы.

1. Вызовите метод execute() для выполнения команды SQL.
2. Получите первый результат или подсчет обновлений.
3. Повторите вызов метода getMoreResults(), чтобы перейти к следующему результирующему набору.
4. Завершите процедуру, если больше не остается результирующих наборов или подсчетов обновлений.

Методы execute() и getMoreResults() возвращают логическое значение true, если следующим звеном в цепочке оказывается результирующий набор. А метод getUpdateCount() возвращает значение **-1**, если следующим звеном в цепочке не оказывается подсчет обновлений. В следующем цикле осуществляется последовательный обход всех полученных результатов:

```
boolean isResult = stat.execute(command);
boolean done = false;
while (!done)
{
  if (isResult)
  {
    ResultSet result = stat.getResultSet();
    сделать что-нибудь с полученным результатом (result)
  }
  else
  {
    int updateCount = stat.getUpdateCount();
    if (updateCount >= 0)
      сделать что-нибудь с подсчетом обновлений (updateCount)
    else
```

```
        done = true;
    }
    if (!done) isResult = stat.getMoreResults();
}
```

java.sql.Statement 1.1

- `boolean getMoreResults()`
- `boolean getMoreResults(int current)` **6**

  Получают следующий результат по данной команде SQL. Параметр *current* принимает значение одной из следующих констант: `CLOSE_CURRENT_RESULT` (по умолчанию), `KEEP_CURRENT_RESULT` или `CLOSE_ALL_RESULTS`. Возвращает логическое значение `true`, если следующий результат существует и представляет собой результирующий набор.

Извлечение автоматически генерируемых ключей

В большинстве баз данных поддерживается механизм автоматической нумерации строк в таблице. К сожалению, у разных поставщиков баз данных эти механизмы заметно отличаются. Автоматически присваиваемые номера часто используются в качестве первичных ключей. Несмотря на то что в JDBC не предлагается независимое от особенностей разных баз данных решение для генерирования подобных ключей, тем не менее в этом прикладном интерфейсе предоставляется эффективный способ их извлечения. Если при вводе новой строки в таблицу автоматически генерируется ключ, его можно получить с помощью следующего кода:

```
stmt.executeUpdate(insertStatement, Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next())
{
  int key = rs.getInt(1);
  . . .
}
```

java.sql.Statement 1.1

- `boolean execute(String statement, int autogenerated)` **1.4**
- `int executeUpdate(String statement, int autogenerated)` **1.4**

  Выполняют указанный оператор SQL, как пояснялось выше. Если параметр *autogenerated* принимает значение `Statement.RETURN_GENERATED_KEYS` и указана команда `INSERT`, то первый столбец таблицы содержит автоматически сгенерированный ключ.

Прокручиваемые и обновляемые результирующие наборы

Как пояснялось ранее, метод `next()` из класса `ResultSet` позволяет последовательно перебирать строки в результирующем наборе, получаемом по запросу базы данных. Его очень удобно использовать для анализа полученных данных. Но нередко требуется предоставить пользователю возможность для просмотра результатов выполнения запроса с переходом к предыдущей и следующей строке, как, например,

показано ранее на рис. 4.4. В *прокручиваемом* результирующем наборе можно свободно перемещаться не только к предыдущим и последующим записям, но и на произвольную позицию.

Кроме того, при просмотре результатов выполнения запроса у пользователей часто возникает потребность исправить какие-нибудь данные. В *обновляемом* результирующем наборе можно видоизменять записи программно, чтобы автоматически обновить их в базе данных. Все эти возможности обращения с результирующими наборами обсуждаются в последующих разделах.

Прокручиваемые результирующие наборы

По умолчанию результирующие наборы не являются прокручиваемыми или обновляемыми. Для организации прокрутки результатов выполнения запроса необходимо получить объект типа `Statement` следующим образом:

```
Statement stat = conn.createStatement(type, concurrency);
```

А для подготовленного оператора потребуется следующий вызов:

```
PreparedStatement stat = conn.prepareStatement(command, type, concurrency);
```

Допустимые значения параметров `type` и `concurrency` перечислены в табл. 4.6 и 4.7. Выбирая эти значения, придется найти ответы на следующие вопросы.

- Требуется ли сделать результирующий набор прокручиваемым? Если этого не требуется, выберите значение `ResultSet.TYPE_FORWARD_ONLY`.
- Если все же требуется сделать результирующий набор прокручиваемым, то должен ли он отражать те данные, которые были изменены в базе данных после выполнения запроса? (Здесь и далее предполагается, что установлен параметр `ResultSet.TYPE_SCROLL_INSENSITIVE`, т.е. результирующий набор не "реагирует" на те изменения, которые произошли в базе данных после выполнения запроса.)
- Требуется ли отредактировать результирующий набор и обновить базу данных? (Более подробно этот вопрос рассматривается в следующем разделе.)

Так, если требуется только прокрутка результирующего набора, но не редактирование его данных, это можно организовать следующим образом:

```
Statement stat = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

Теперь можно прокручивать все результирующие наборы, возвращаемые при вызовах приведенного ниже метода. Получаемый в итоге результирующий набор содержит *курсор*, устанавливаемый на текущей позиции.

```
ResultSet rs = stat.executeQuery(query)
```

Таблица 4.6. Значения типа `ResultSet`

Значение	Описание
`TYPE_FORWARD_ONLY`	Без прокрутки (по умолчанию)
`TYPE_SCROLL_INSENSITIVE`	С прокруткой, но без учета изменений в базе данных
`TYPE_SCROLL_SENSITIVE`	С прокруткой и с учетом изменений в базе данных

Таблица 4.7. Параллельные значения `ResultSet`

Значение	Описание
CONCUR_READ_ONLY	Без редактирования и обновления базы данных (по умолчанию)
CONCUR_UPDATABLE	С редактированием и обновлением базы данных

НА ЗАМЕТКУ! Не все драйверы баз данных поддерживают прокручиваемые или обновляемые результирующие наборы. (Методы `supportsResultSetType()` и `supportsResultSetConcurrency()` из интерфейса `DatabaseMetaData` сообщают о типах и режимах параллельной обработки, которые поддерживаются в конкретной базе данных с помощью определенного драйвера.) Но даже если в базе данных поддерживаются результирующие наборы во всех описанных режимах, в некоторых запросах нельзя получить результирующий набор со всеми запрашиваемыми свойствами. (Например, результат выполнения сложного запроса может оказаться необновляемым.)

В этом случае метод `executeQuery()` возвращает объект типа `ResultSet` с меньшими возможностями, вводя предупреждение типа `SQLWarning` в объект соединения. (Способ извлечения предупреждений представлен в разделе "Анализ исключений SQL" ранее в этой главе.) С другой стороны, для выявления конкретного режима работы результирующего набора можно вызвать методы `getType()` и `getConcurrency()` из класса `ResultSet`. Если не выяснить конкретный режим работы результирующего набора и попытаться выполнить неподдерживаемую в нем операцию, например, вызвать метод `previous()` для непрокручиваемого результирующего набора, это неизбежно приведет к исключению типа `SQLException`.

Прокрутка организуется очень просто. Например, для перехода к предыдущим записям в результирующем наборе служит приведенная ниже конструкция. Метод `previous()` возвращает логическое значение `true`, если курсор находится на конкретной строке в результирующем наборе, и логическое значение `false`, если курсор находится перед первой строкой.

```
if (rs.previous()) . . .
```

Для перемещения курсора на *n* строк вперед или назад вызывается следующий метод:

```
rs.relative(n);
```

При положительных значениях параметра *n* курсор перемещается вперед, а при отрицательных — назад (нулевое значение параметра *n* не приводит ни к каким перемещениям). Если попытаться переместить курсор за пределы текущего ряда строк, он расположится за последней строкой или же перед первой строкой в зависимости от знака параметра *n*. После этого метод `relative()` возвращает логическое значение `false`, а перемещение курсора прекращается. Данный метод возвращает логическое значение `true` только в том случае, если курсор устанавливается на конкретной строке.

С другой стороны, курсор можно установить на конкретной строке под номером *n*, вызвав следующий метод:

```
s.absolute(n);
```

А получить текущий номер строки *n* можно следующим образом:

```
int currentRow = rs.getRow();
```

Первая строка в результирующем наборе имеет номер **1**. Если возвращаемое значение равно нулю, значит, курсор находится не на конкретной строке, а за последней

или перед первой строкой. Для перемещения курсора на первой или последней строке результирующего набора, установки курсора перед первой или за последней строкой предусмотрены служебные методы `first()`, `last()`, `beforeFirst()` и `afterLast()` соответственно, а для проверки расположения курсора на одной из этих позиций — служебные методы `isFirst()`, `isLast()`, `isBeforeFirst()` и `isAfterLast()`.

Как видите, обращаться с прокручиваемыми результирующими наборами совсем не трудно. Все рутинные операции, связанные с кешированием данных, получаемых по запросу, выполняются драйвером базы данных.

Обновляемые результирующие наборы

Если требуется отредактировать данные, полученные по запросу в результирующем наборе, а также автоматически обновить базу данных, такой набор нужно сделать обновляемым. Обновляемые результирующие наборы совсем не обязательно должны быть прокручиваемыми. Но если требуется предоставить пользователю возможность редактировать данные, то они должны допускать и прокрутку.

Для получения обновляемого результирующего набора служит приведенный ниже код, где это обстоятельство выделяется полужирным. Результирующие наборы, возвращаемые методом `executeQuery()`, становятся в итоге обновляемыми.

```
Statement stat = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

НА ЗАМЕТКУ! Не по всем запросам возвращается обновляемый результирующий набор. Так, если в запросе предполагается соединение нескольких таблиц, его результат не всегда может быть обновляемым. Но если в запросе предполагается обращение к одной таблице или соединение нескольких таблиц по их первичным ключам, то следует ожидать, что получаемый в итоге результирующий набор окажется обновляемым. Для того чтобы выяснить, является ли результирующий набор обновляемым, следует вызвать метод `getConcurrency()` из класса `ResultSet`.

Допустим, требуется повысить цену на некоторые книги, но отсутствует единый критерий, который можно было бы использовать для этого в команде `UPDATE`. В таком случае придется перебрать в цикле все книги и изменить цены по произвольным условиям, как показано ниже.

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next())
{
  if (. . .)
  {
    double increase = . . .
    double price = rs.getDouble("Price");
    rs.updateDouble("Price", price + increase);
    rs.updateRow(); // непременно вызвать метод updateRow()
                    // после обновления полей в таблице
  }
}
```

Для всех типов данных SQL предусмотрены соответствующие формы метода `updateXxx()`, например `updateDouble()`, `updateString()` и т.д. Как и при вызове метода `getXxx()`, в качестве параметров данного метода могут указываться номер или имя столбца, а затем новое значение для поля.

НА ЗАМЕТКУ! Применяя метод `updateXxx()`, следует иметь в виду, что первый параметр обозначает номер столбца в результирующем наборе, где он может отличаться от номера столбца в базе данных.

Метод `updateXxx()` изменяет только значения полей в текущей строке результирующего набора, а не в самой базе данных. Для обновления всех полей отредактированной строки в базе данных следует вызвать метод `updateRow()`. Если же переместить курсор к следующей строке, не вызывая метод `updateRow()`, все обновления предыдущей строки в результирующем наборе будут отменены, поскольку они не были переданы базе данных. Для отмены обновлений текущей строки в базе данных следует вызвать метод `cancelRowUpdates()`.

В предыдущем примере был продемонстрирован порядок внесения изменений в существующей строке. Для создания новой строки в базе данных нужно сначала вызвать метод `moveToInsertRow()`, переместив тем самым курсор на специальную позицию, называемую *строкой вставки*. Затем новая строка создается на данной позиции с помощью метода `updateXxx()`. А для передачи новой вставляемой строки в базу данных вызывается метод `insertRow()`. По окончании вставки вызывается метод `moveToCurrentRow()`, чтобы переместить курсор назад на ту позицию, которую он занимал до вызова метода `moveToInsertRow()`. В приведенном ниже примере показано, каким образом весь этот процесс реализуется непосредственно в коде.

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

Следует, однако, иметь в виду, что конкретное *расположение* новых данных в результирующем наборе или базе данных *не* поддается непосредственному управлению из прикладного кода. Если не указать конкретное значение для столбца в строке вставки, на этом месте окажется пустое значение `NULL`. Но если на столбец наложено ограничение `NOT NULL`, то сгенерируется исключение и строка не будет вставлена. И наконец, для удаления той строки, на которой установлен курсор, вызывается приведенный ниже метод. Этот метод немедленно удаляет строку как из результирующего набора, так и из базы данных.

```
rs.deleteRow();
```

Таким образом, методы `updateRow()`, `insertRow()` и `deleteRow()` из класса `ResultSet` предоставляют те же самые возможности, что и команды `UPDATE`, `INSERT` и `DELETE` языка SQL. Для программирующих на Java вызов методов более привычен, и поэтому они предпочитают данный подход составлению запросов из команд SQL.

ВНИМАНИЕ! При неаккуратном обращении с обновляемыми результирующими наборами можно получить совершенно неэффективный код. Нередко выполнение команды `UPDATE` оказывается *намного* более эффективным, чем составление запроса и просмотр результирующего набора. Обработку обновляемых результирующих наборов имеет смысл организовывать в диалоговых прикладных программах, где пользователь может вносить произвольные изменения. А для внесения заранее программируемых изменений более подходящей оказывается команда `UPDATE`.

НА ЗАМЕТКУ! В версии JDBC 2 были внедрены дополнительные усовершенствования в результирующие наборы, в том числе возможность обновлять результирующие наборы самыми последними данными, если они были изменены при другом, параллельном соединении с базой данных. А в версии JDBC 3 было внедрено еще одно усовершенствование, определяющее режим работы результирующих наборов при фиксации транзакции. Но эти дополнительные возможности здесь не рассматриваются, поскольку они выходят за рамки введения в базы данных. За дополнительными сведениями о них отсылаем читателей к книге *JDBC™ API Tutorial and Reference, Third Edition* Мэйдена Фишера, Джона Эллиса и Джонатана Брюса (Maydene Fisher, Jon Ellis, Jonathan Bruce; издательство Addison-Wesley, 2003 г.), а также к документации, описывающей спецификацию прикладного интерфейса JDBC и доступной по адресу `www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html`.

java.sql.Connection 1.1

- `Statement createStatement(int type, int concurrency)` **1.2**
- `PreparedStatement prepareStatement(String command, int type, int concurrency)` **1.2**

  Создает обычный или подготовленный оператор SQL и возвращает результирующий набор.

  Параметры: *command* Оператор, подготавливаемый в виде команды SQL

  type Одна из констант (`TYPE_FORWARD_ONLY`, `CONCUR_UPDATABLE` или `TYPE_SCROLL_SENSITIVE`), определяемых в интерфейсе `ResultSet`

  concurrency Одна из констант (`CONCUR_READ_ONLY` или `CONCUR_UPDATABLE`), определяемых в интерфейсе `ResultSet`

java.sql.ResultSet 1.1

- `int getType()` **1.2**

  Возвращает одну из констант (`TYPE_FORWARD_ONLY`, `CONCUR_UPDATABLE` или `TYPE_SCROLL_SENSITIVE`), обозначающих тип результирующего набора.

- `int getConcurrency()` **1.2**

  Возвращает константу (`CONCUR_READ_ONLY` или `CONCUR_UPDATABLE`), обозначающую способ параллельного обращения к результирующему набору (только для чтения или для обновления).

- `boolean previous()` **1.2**

  Перемещает курсор к предыдущей строке. Возвращает логическое значение `true`, если курсор устанавливается на строке, или логическое значение `false`, если он устанавливается перед первой строкой.

- `int getRow()` **1.2**

  Получает номер текущей строки. Нумерация строк начинается с **1**.

- `boolean absolute(int r)` **1.2**

  Перемещает курсор к строке с номером `r`. Возвращает логическое значение `true`, если курсор устанавливается на строке.

- `boolean relative(int d)` **1.2**

  Перемещает курсор на `d` строк. Если значение параметра `d` меньше нуля, то перемещение происходит в обратном направлении. Возвращает логическое значение `true`, если курсор устанавливается на строке.

- `boolean first()` **1.2**
- `boolean last()` **1.2**

  Перемещают курсор к первой или к последней строке. Возвращают логическое значение `true`, если курсор устанавливается на строке.

- `void beforeFirst()` **1.2**
- `void afterLast()` **1.2**

  Устанавливают курсор перед первой или за последней строкой.

- `boolean isFirst()` **1.2**
- `boolean isLast()` **1.2**

  Проверяют, находится ли курсор на первой или на последней строке.

- `boolean isBeforeFirst()` **1.2**
- `boolean isAfterLast()` **1.2**

  Проверяют, находится ли курсор перед первой или за последней строкой.

- `void moveToInsertRow()` **1.2**

  Перемещает курсор на строку вставки. Строка вставки — это специальная строка, которая служит для вставки новых данных с помощью методов `updateXxx()` и `innsertRow()`.

- `void moveToCurrentRow()` **1.2**

  Перемещает курсор из строки вставки на строку, где он находился до вызова метода `moveToInsertRow()`.

- `void insertRow()` **1.2**

  Вводит содержимое строки вставки в базу данных и результирующий набор.

- `void deleteRow()` **1.2**

  Удаляет текущую строку из базы данных и результирующего набора.

- `void updateXxx(int column, Xxx data)` **1.2**
- `void updateXxx(String columnName, Xxx data)` **1.2**

  (*Xxx* обозначает тип данных, например `int`, `double`, `String`, `Date` и т.д.)

  Обновляют содержимое указанного столбца текущей строки в результирующем наборе.

- `void updateRow()` **1.2**

  Передает обновления текущей строки в базу данных.

- `void cancelRowUpdates ()` **1.2**

  Отменяет обновления текущей строки.

java.sql.DatabaseMetaData 1.1

- `boolean supportsResultSetType(int` *type*`)` **1.2**

  Возвращает логическое значение `true`, если база данных способна поддерживать заданный тип результирующего набора.

  Параметры: *type* Одна из констант (`TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE` или `TYPE_SCROLL_SENSITIVE`), определенных в интерфейсе `ResultSet` и обозначающих способ прокрутки

- `boolean supportsResultSetConcurrency(int` *type*`, int` *concurrency*`)` **1.2**

  Возвращает логическое значение `true`, если база данных способна поддерживать заданный способ прокрутки и параллельного обращения к результирующему набору.

  Параметры: *type* Одна из констант (`TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE` или `TYPE_SCROLL_SENSITIVE`), определенных в интерфейсе `ResultSet` и обозначающих способ прокрутки

  concurrency Одна из констант (`CONCUR_READ_ONLY` или `CONCUR_UPDATABLE`), определенных в интерфейсе `ResultSet` и обозначающих способ параллельного обращения к результирующему набору (только для чтения или для обновления)

Наборы строк

Прокручиваемые результирующие наборы предлагают богатые возможности, но они не свободны от недостатков. В течение всего периода взаимодействия с пользователем соединение с базой данных должно быть установлено. Но ведь пользователь может отлучиться на длительное время, а между тем установленное соединение будет напрасно потреблять сетевые ресурсы. В подобной ситуации целесообразно использовать *набор строк*. Интерфейс `RowSet` расширяет интерфейс `ResultSet`, но набор строк не должен быть непременно привязан к соединению с базой данных.

Наборы строк применяются и в том случае, если требуется перенести результаты выполнения запроса на другой уровень сложного приложения или на другое устройство, например на мобильный телефон. Перенести результирующий набор нельзя, поскольку он связан с соединением, а кроме того, структура данных может иметь довольно крупные размеры.

Построение наборов строк

Ниже перечислены интерфейсы, входящие в пакет `javax.sql.rowset` и расширяющие интерфейс `RowSet`.

- Интерфейс `CachedRowSet` позволяет выполнять некоторые операции при отсутствии соединения. Кешируемые наборы строк рассматриваются в следующем разделе.
- Интерфейс `WebRowSet` представляет собой кешируемый набор строк, который может быть сохранен в XML-файле. А сам XML-файл может быть передан другому компоненту приложения и открыт с помощью другого объекта типа `WebRowSet`.
- Интерфейсы `FilteredRowSet` и `JoinRowSet` поддерживают легковесные операции с наборами строк, равнозначные таким командам SQL, как `SELECT` и `JOIN`. Эти операции выполняются только над данными, содержащимися в наборе строк, для чего не требуется устанавливать соединение с базой данных.
- Интерфейс `JdbcRowSet` является оболочкой для интерфейса `ResultSet`. Он определяет методы установки и получения, превращая результирующий набор в компонент JavaBeans. (Более подробно компоненты JavaBeans рассматриваются в главе 8.)

В версии Java 7 появился стандартный способ получения набора строк с помощью приведенных ниже методов. Аналогичные методы имеются для получения наборов строк других типов.

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

До версии Java 7 применялись методы создания наборов строк, специфические для разных баз данных. Кроме того, в пакете `com.sun.rowset`, входящем в состав JDK, предоставляются базовые реализации, позволяющие применять наборы строк даже в том случае, если в конкретной базе данных они не поддерживаются. Имена классов в этих реализациях оканчиваются на `Impl`, например `CachedRowSetImpl`. Как показано ниже, к этим классам можно прибегнуть, если, например, нельзя воспользоваться классом `RowSetProvider`.

```
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
```

Кешируемые наборы строк

Кешируемый набор строк содержит данные из результирующего набора. Интерфейс `CachedRowSet` расширяет интерфейс `ResultSet`, а следовательно, кешируемым набором строк можно пользоваться точно так же, как и результирующим набором. Но наборы строк имеют существенное преимущество, позволяющее разорвать соединение с базой данных и продолжать работать с набором строк. Как демонстрируется в примере программы, исходный код которой представлен далее в листинге 4.4, такая возможность существенно упрощает создание диалоговых приложений. При получении команды от пользователя устанавливается соединение, выполняется запрос, результаты размещаются в наборе строк, после чего соединение с базой данных разрывается.

Кешируемый набор строк позволяет даже видоизменить содержащиеся в нем данные. Разумеется, результаты подобных изменений не отражаются в базе данных немедленно. Для того чтобы принять накопленные изменения, необходимо выполнить явный запрос. В этом случае объект типа `CachedRowSet` повторно устанавливает соединение и выдает команды SQL для записи изменений в базу данных. Объект типа `CachedRowSet` заполняется данными из результирующего набора следующим образом:

```
ResultSet result = . . .;
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
  // или воспользоваться реализацией для конкретной базы данных
crs.populate(result);
conn.close(); // теперь можно разорвать соединение с базой данных
```

С другой стороны, можно предоставить объекту типа `CachedRowSet` возможность автоматически установить соединение. Для этого сначала задаются следующие параметры базы данных:

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

Затем составляется оператор запроса с любыми параметрами, как показано ниже.

```
crs.setCommand("SELECT * FROM Books WHERE PUBLISHER = ?");
crs.setString(1, publisherName);
```

И, наконец, набор строк заполняется результатами запроса. В результате приведенного ниже вызова устанавливается соединение с базой данных, выполняется запрос, заполняется набор строк и затем разрывается соединение.

```
crs.execute();
```

Если полученный результат запроса имеет слишком большой объем, его можно и не полностью вводить в набор строк. В конце концов, пользователи прикладной программы, скорее всего, просмотрят лишь некоторые строки. В таком случае нужно определить размеры страницы следующим образом:

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20);
. . .
crs.execute();
```

В итоге будет доступно только **20** строк. А для того чтобы получить следующую порцию строк, достаточно вызвать приведенный ниже метод.

```
crs.nextPage();
```

Для просмотра и видоизменения набора строк служат те же самые методы, что и для обращения с результирующим набором. Так, если изменить содержимое набора строк, то для записи изменений в базу данных необходимо сделать один из следующих вызовов:

```
crs.acceptChanges(conn);
```

или

```
crs.acceptChanges();
```

Второй вариант вызова метода `acceptChanges()` действует только в том случае, если предоставить для набора строк все сведения, необходимые для соединения с базой данных (URL, имя пользователя и пароль).

Как упоминалось ранее в разделе "Обновляемые результирующие наборы", не все результирующие наборы являются обновляемыми. Аналогично наборы строк, содержащие результаты сложных запросов, не позволяют записывать изменения в базу данных. Если же набор строк содержит данные только из одной таблицы, то никаких затруднений при их записи в базу данных не возникает.

ВНИМАНИЕ! Если заполнить набор строк данными из результирующего набора, то набору строк не будет известно имя обновляемой таблицы. В этом случае нужно специально указать имя таблицы, вызвав метод `setTable()`.

Если данные в базе изменились с того момента, как набор строк был заполнен ими, то возникают дополнительные затруднения, связанные с несоответствием данных. В базовой реализации проверяется, совпадают ли исходные значения из набора строк (т.е. значения перед редактированием) с текущими значениями в базе данных. Если проверка дает положительный результат, то содержимое базы данных заменяется видоизмененными данными. В противном случае генерируется исключение типа `SyncProviderException` и внесенные изменения не записываются. В других реализациях могут применяться иные способы синхронизации данных.

javax.sql.RowSet 1.4

- `String getURL()`
- `void setURL(String url)`

  Получают или устанавливают URL базы данных.
- `String getUsername()`
- `void setUsername(String username)`

  Получают или устанавливают имя пользователя для соединения с базой данных.
- `String getPassword()`
- `void setPassword(String password)`

  Получают или устанавливают пароль для соединения с базой данных.
- `String getCommand()`

- `void setCommand(String` *command*`)`

  Получают или устанавливают команду, при выполнении которой набор строк заполняется данными.

- `void execute()`

  Заполняет данную строку по команде, установленной с помощью метода `setCommand()`. Для того чтобы диспетчер драйверов мог установить соединение, должны быть заданы URL, имя пользователя и пароль.

javax.sql.rowset.CachedRowSet 5.0

- `void execute(Connection` *conn*`)`

  Заполняет набор строк по команде, установленной с помощью метода `setCommand()`. Использует указанное соединение с базой данных, а затем *разрывает* его.

- `void populate(ResultSet` *result*`)`

  Заполняет кешируемый набор строк данными из указанного результирующего набора.

- `String getTableName()`
- `void setTableName(String tableName)`

  Получают или устанавливают имя *таблицы*, данными из которой заполняется кешируемый набор строк.

- `int getPageSize()`
- `void setPageSize(int` *size*`)`

  Получают или устанавливают размер страницы.

- `boolean nextPage()`
- `boolean previousPage()`

  Загружают следующую или предыдущую страницу строк. Возвращают логическое значение `true`, если существует следующая или предыдущая страница.

- `void acceptChanges()`
- `void acceptChanges(Connection` *conn*`)`

  Повторно устанавливают соединение с базой данных и записывают в нее изменения, внесенные в набор строк. Если с момента заполнения набора содержимое базы данных изменилось, данные не могут быть записаны в нее обратно. В этом случае генерируется исключение типа `SyncProviderException`.

javax.sql.rowset.RowSetProvider 7

- `static RowSetFactory newFactory()`

  Создает фабрику наборов строк.

javax.sql.rowset.RowSetFactory 7

- `CachedRowSet createCachedRowSet()`
- `FilteredRowSet createFilteredRowSet()`
- `JdbcRowSet createJdbcRowSet()`
- `JoinRowSet createJoinRowSet()`
- `WebRowSet createWebRowSet()`

  Создают набор строк заданного типа.

Метаданные

В предыдущих разделах рассматривались способы ввода и обновления содержимого таблиц базы данных и составления запросов. Помимо этого, в JDBC предусмотрены дополнительные возможности для получения сведений о структуре таблиц и самой базы данных. В частности, можно получить список всех таблиц базы данных или имена всех столбцов с типами данных в них. Эти сведения вряд ли будут очень полезны при разработке прикладной программы, предназначенной для работы с конкретной базой данных, потому что в таких случаях ее структура точно известна. Но они пригодятся тем разработчикам, которые создают свои программные продукты для работы с любыми базами данных.

В языке SQL сведения о структуре базы данных и ее компонентов называются *метаданными*. Такое название выбрано лишь для того, чтобы как-то отличать сведения о базе данных от ее основного содержимого. Существуют метаданные трех типов, описывающие структуру базы данных, структуру результирующих наборов и параметры подготовленных операторов.

Для получения более подробных сведений о структуре базы данных требуется объект типа `DatabaseMetaData`, который можно получить из установленного соединения с базой данных следующим образом:

```
DatabaseMetaData meta = conn.getMetaData();
```

Далее можно приступать непосредственно к получению метаданных. Так, если вызвать приведенный ниже метод, то в итоге будет получен результирующий набор, содержащий сведения обо всех таблицах базы данных. (Параметры этого метода рассматриваются далее при описании соответствующего прикладного интерфейса API.)

```
ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
```

Каждая строка из получаемого в итоге результирующего набора содержит сведения об отдельной таблице, а третий столбец в ней — имя таблицы, как поясняется далее в описании соответствующего прикладного интерфейса API. В приведенном ниже фрагменте кода организуется цикл для сбора сведений об именах всех таблиц в базе данных.

```
while (mrs.next())
    tableNames.addItem(mrs.getString(3));
```

Метаданные базы данных находят еще одно полезное применение. Базы данных могут иметь очень сложную структуру, а стандарт SQL предоставляет немало места для отклонений от нормы. Поэтому в интерфейсе `DatabaseMetaData` предусмотрено более сотни разных методов, которые можно использовать для получения сведений о структуре базы данных. Ниже показаны примеры вызова таких методов с довольно необычными именами. Судя по названиям этих методов, они предназначены главным образом для очень опытных разработчиков, в том числе тех, кто занимается написанием переносимого кода, способного работать с разнотипными базами данных.

```
meta.supportsCatalogsInPrivilegeDefinitions()
meta.nullPlusNonNullIsNull()
```

Интерфейс `DatabaseMetaData` предоставляет сведения о базе данных, а сведения о результирующем наборе — второй интерфейс `ResultSetMetaData`. Получив результирующий набор по запросу, можно определить количество столбцов, имена столбцов, типы данных в них и ширину полей. Ниже приведен типичный цикл, в котором все эти сведения извлекаются с помощью соответствующих методов, выделенных полужирным.

```
ResultSet mrs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData meta = mrs.getMetaData();
for (int i = 1; i <= meta.getColumnCount(); i++)
{
  String columnName = meta.getColumnLabel(i);
  int columnWidth = meta.getColumnDisplaySize(i);
  . . .
}
```

В этом разделе поясняется, как создать простое инструментальное средство, предназначенное для просмотра и анализа структуры базы данных. Исходный код примера программы, реализующей это средство, приведен в листинге 4.4. В этой программе демонстрируется также применение кешированного набора строк.

В верхней части рабочего окна рассматриваемой здесь программы находится комбинированный список с именами всех таблиц базы данных. Как показано на рис. 4.6, после выбора какой-нибудь одной таблицы в центральной части фрейма будут представлены имена столбцов из этой таблицы, а также значения из первой строки. Для просмотра строк в таблице следует щелкнуть на кнопках Next (Следующая) и Previous (Предыдущая). Строки можно удалять, а также редактировать значения в них. Чтобы сохранить изменения в базе данных, следует щелкнуть на кнопке Save (Сохранить).

НА ЗАМЕТКУ! В состав баз данных обычно включаются инструментальные средства для просмотра и редактирования таблиц, обладающие намного большими возможностями. Если для вашей базы такое инструментальное средство отсутствует, попробуйте воспользоваться iSQL-Viewer (`http://isql.sourceforge.net`) или `SQuirreL` (`http://squirrel-sql.sourceforge.net`). Эти инструментальные средства позволяют просматривать таблицы любой базы данных, совместимой с прикладным интерфейсом JDBC. Рассматриваемая здесь программа отнюдь не претендует на то, чтобы соперничать с этими инструментальными средствами. Она лишь демонстрирует общие принципы создания программ для работы с произвольными таблицами базы данных.

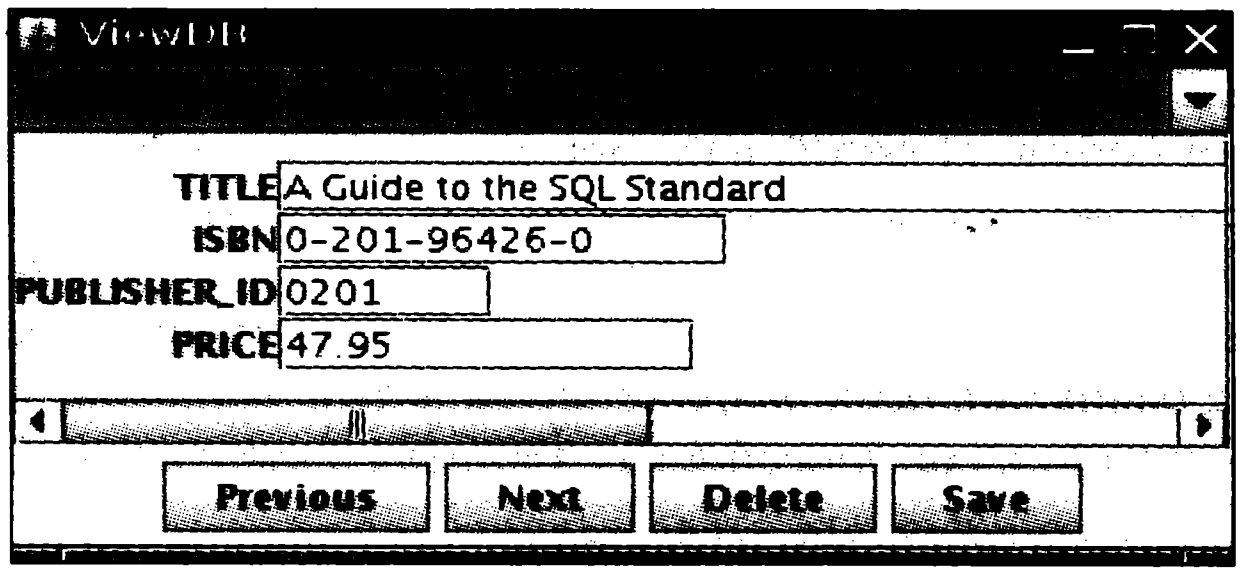


Рис. 4.6. Прикладная программа `ViewDB`

Листинг 4.4. Исходный код из файла `view/ViewDB.java`

```
1  package view;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.io.*;
6  import java.nio.file.*;
7  import java.sql.*;
8  import java.util.*;
9  import javax.sql.*;
10 import javax.sql.rowset.*;
11 import javax.swing.*;
12
13 /**
14  * В этой программе демонстрируется применение метаданных для
15  * отображения произвольно выбираемых таблиц в базе данных
16  * @version 1.32 2012-01-26
17  * @author Cay Horstmann
18 */
19 public class ViewDB
20 {
21   public static void main(String[] args)
22   {
23     EventQueue.invokeLater(new Runnable()
24       {
25         public void run()
26         {
27           JFrame frame = new ViewDBFrame();
28           frame.setTitle("ViewDB");
29           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30           frame.setVisible(true);
31         }
32       });
33   }
34 }
35
36 /**
37  * Фрейм, содержащий панель с кнопками перемещения по данным
38 */
39 class ViewDBFrame extends JFrame
```

```
40  {
41    private JButton previousButton;
42    private JButton nextButton;
43    private JButton deleteButton;
44    private JButton saveButton;
45    private DataPanel dataPanel;
46    private Component scrollPane;
47    private JComboBox<String> tableNames;
48    private Properties props;
49    private CachedRowSet crs;
50
51    public ViewDBFrame()
52    {
53      tableNames = new JComboBox<String>();
55      tableNames.addActionListener(new ActionListener()
56        {
57          public void actionPerformed(ActionEvent event)
58          {
59            showTable((String) tableNames.getSelectedItem());
60          }
61        });
62      add(tableNames, BorderLayout.NORTH);
63
64      try
65      {
66        readDatabaseProperties();
67        try (Connection conn = getConnection())
68        {
69          DatabaseMetaData meta = conn.getMetaData();
70          ResultSet mrs =
71            meta.getTables(null, null, null, new String[] { "TABLE" });
72          while (mrs.next())
73            tableNames.addItem(mrs.getString(3));
74        }
75      }
76      catch (SQLException e)
77      {
78        JOptionPane.showMessageDialog(this, e);
79      }
80      catch (IOException e)
81      {
82        JOptionPane.showMessageDialog(this, e);
83      }
84
85      JPanel buttonPanel = new JPanel();
86      add(buttonPanel, BorderLayout.SOUTH);
87
88      previousButton = new JButton("Previous");
89      previousButton.addActionListener(new ActionListener()
90        {
91          public void actionPerformed(ActionEvent event)
92          {
93            showPreviousRow();
94          }
95        });
96      buttonPanel.add(previousButton);
97      nextButton = new JButton("Next");
```

```
98        nextButton.addActionListener(new ActionListener()
99          {
100           public void actionPerformed(ActionEvent event)
101           {
102             showNextRow();
103           }
104         });
105       buttonPanel.add(nextButton);
106
107       deleteButton = new JButton("Delete");
108       deleteButton.addActionListener(new ActionListener()
109         {
110           public void actionPerformed(ActionEvent event)
111           {
112             deleteRow();
113           }
114         });
115       buttonPanel.add(deleteButton);
116
117       saveButton = new JButton("Save");
118       saveButton.addActionListener(new ActionListener()
119         {
120           public void actionPerformed(ActionEvent event)
121             {
122               saveChanges();
123             }
124         });
125       buttonPanel.add(saveButton);
126       pack();
127     }
128
129     /**
130      * Подготавливает текстовые поля для показа новой таблицы и
131      * отображает первую ее строку
132      * @param tableName Имя отображаемой таблицы
133     */
134     public void showTable(String tableName)
135     {
136       try
137       {
138         try (Connection conn = getConnection())
139         {
140           // получить результирующий набор
141           Statement stat = conn.createStatement();
142           ResultSet result =
143             stat.executeQuery("SELECT * FROM " + tableName);
144           // скопировать в кешируемый результирующий набор
145           RowSetFactory factory = RowSetProvider.newFactory();
146           crs = factory.createCachedRowSet();
147           crs.setTableName(tableName);
148           crs.populate(result);
149         }
150
151         if (scrollPane != null) remove(scrollPane);
152         dataPanel = new DataPanel(crs);
153         scrollPane = new JScrollPane(dataPanel);
154         add(scrollPane, BorderLayout.CENTER);
```

```
155         validate();
156         showNextRow();
157       }
158       catch (SQLException e)
159       {
160         JOptionPane.showMessageDialog(this, e);
161       }
162     }
163
164     /**
165      * Осуществляет переход к предыдущей строке таблицы
166     */
167     public void showPreviousRow()
168     {
169       try
170       {
171         if (crs == null || crs.isFirst()) return;
172         crs.previous();
173         dataPanel.showRow(crs);
174       }
175       catch (SQLException e)
176       {
177         for (Throwable t : e)
178         t.printStackTrace();
179       }
180     }
181
182     /**
183      * Осуществляет переход к следующей строке таблицы
184     */
185     public void showNextRow()
186     {
187       try
188       {
189         if (crs == null || crs.isLast()) return;
190         crs.next();
191         dataPanel.showRow(crs);
192       }
193       catch (SQLException e)
194       {
195         JOptionPane.showMessageDialog(this, e);
196       }
197     }
198
199     /**
200      * Удаляет строку из текущей таблицы
201     */
202     public void deleteRow()
203     {
204       try
205       {
206         try (Connection conn = getConnection())
207         {
208           crs.deleteRow();
209           crs.acceptChanges(conn);
210           if (crs.isAfterLast())
211             if (!crs.last()) crs = null;
```

```
212           dataPanel.showRow(crs);
213         }
214       }
215       catch (SQLException e)
216       {
217         JOptionPane.showMessageDialog(this, e);
218       }
219     }
220
221     /**
222      * Сохраняет все внесенные изменения
223     */
224     public void saveChanges()
225     {
226       try
227       {
228         try (Connection conn = getConnection())
229         {
230           dataPanel.setRow(crs);
231           crs.acceptChanges(conn);
232         }
233       }
234       catch (SQLException e)
235       {
236         JOptionPane.showMessageDialog(this, e);
237       }
238     }
239
240     private void readDatabaseProperties() throws IOException
241     {
242       props = new Properties();
243       try (InputStream in =
244         Files.newInputStream(Paths.get("database.properties")))
245       {
246         props.load(in);
247       }
248       String drivers = props.getProperty("jdbc.drivers");
249       if (drivers != null) System.setProperty("jdbc.drivers", drivers);
250     }
251
252     /**
253      * Получает соединение с базой данных из свойств,
254      * определенных в файле database.properties
255      * @return Возвращает соединение с базой данных
256     */
257     private Connection getConnection() throws SQLException
258     {
259       String url = props.getProperty("jdbc.url");
260       String username = props.getProperty("jdbc.username");
261       String password = props.getProperty("jdbc.password");
262
263       return DriverManager.getConnection(url, username, password);
264     }
265 }
266
267 /**
268  * На этой панели отображается содержимое результирующего набора
```

```
269 */
270 class DataPanel extends JPanel
271 {
272   private java.util.List<JTextField> fields;
273
274   /**
275    * Конструирует панель для отображения данных
276    * @param rs Результирующий набор, содержимое которого
277    *           отображается на данной панели
278   */
279   public DataPanel(RowSet rs) throws SQLException
280   {
281     fields = new ArrayList<>();
282     setLayout(new GridBagLayout());
283     GridBagConstraints gbc = new GridBagConstraints();
284     gbc.gridwidth = 1;
285     gbc.gridheight = 1;
286
287     ResultSetMetaData rsmd = rs.getMetaData();
288     for (int i = 1; i <= rsmd.getColumnCount(); i++)
289     {
290       gbc.gridy = i - 1;
291       String columnName = rsmd.getColumnLabel(i);
292       gbc.gridx = 0;
293       gbc.anchor = GridBagConstraints.EAST;
294       add(new JLabel(columnName), gbc);
295
296       int columnWidth = rsmd.getColumnDisplaySize(i);
297       JTextField tb = new JTextField(columnWidth);
298       if (!rsmd.getColumnClassName(i).equals("java.lang.String"))
299       tb.setEditable(false);
300
301       fields.add(tb);
302
303       gbc.gridx = 1;
304       gbc.anchor = GridBagConstraints.WEST;
305       add(tb, gbc);
306     }
307   }
308
309   /**
310    * Отображает строку из таблицы базы данных, заполняя
311    * все текстовые значениями из столбцов
312   */
313   public void showRow(ResultSet rs) throws SQLException
314   {
315     for (int i = 1; i <= fields.size(); i++)
316     {
317       String field = rs == null ? "" : rs.getString(i);
318       JTextField tb = fields.get(i - 1);
319       tb.setText(field);
320     }
321   }
322
323   /**
324    * Обновляет измененными данными текущую строку
325    * из результирующего набора
```

```
326    */
327    public void setRow(RowSet rs) throws SQLException
328    {
329      for (int i = 1; i <= fields.size(); i++)
330      {
331        String field = rs.getString(i);
332        JTextField tb = fields.get(i - 1);
333        if (!field.equals(tb.getText()))
334          rs.updateString(i, tb.getText());
335      }
336      rs.updateRow();
337    }
338 }
```

java.sql.Connection 1.1

- `DatabaseMetaData getMetaData()`

  Возвращает метаданные в виде объекта `DatabaseMetaData` для соединения с базой данных.

java.sql.DatabaseMetaData 1.1

- `ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])`

  Возвращает из указанного каталога описание всех таблиц, совпадающих с шаблонами схемы и имен таблиц, а также с заданными критериями типов. (*Схема* описывает группу связанных таблиц и полномочия доступа к ним, а *каталог* — группу связанных вместе схем. Эти понятия важны для структурирования крупных баз данных.)

  В качестве параметров `catalog` и *`schemaPattern`* могут быть указаны пустые символьные строки (""), чтобы извлечь таблицы без каталога и схемы, или же пустые значения `null`, если требуется возвратить таблицы независимо от каталога или схемы.

  Массив `types` содержит следующие имена типов таблиц: `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS` и `SYNONYM`. Если вместо массива `types` указано пустое значение `null`, то возвращаются таблицы всех типов.

  Результирующий набор состоит из пяти столбцов типа `String`, как показано ниже.

Столбец	*Имя*	*Описание*
1	`TABLE_CAT`	Каталог таблиц (может иметь пустое значение `null`)
2	`TABLE_SCHEM`	Схема (может иметь пустое значение `null`)
3	`TABLE_NAME`	Имя таблицы
4	`TABLE_TYPE`	Имя таблицы
5	`REMARKS`	Комментарии для таблицы

- `int getJDBCMajorVersion()` **1.4**
- `int getJDBCMinorVersion()` **1.4**
- Возвращают основной и дополнительный номер версии драйвера JDBC, устанавливающего соединение с базой данных. Например, драйвер JDBC 3.0 имеет основной номер версии **3** и дополнительный номер версии **0**.
- `int getMaxConnections()`
- Возвращает максимальное количество соединений с базой данных, которые допускается устанавливать одновременно.
- `int getMaxStatements()`
- Возвращает максимальное количество команд, которые допускается одновременно открывать на каждое соединение с базой данных. Если это количество неограничено или неизвестно, то возвращается нулевое значение.

java.sql.ResultSet 1.1

- `ResultSetMetaData getMetaData()`
  Возвращает метаданные, связанные со столбцами текущего результирующего набора типа `ResultSet`.

java.sql.ResultSetMetaData 1.1

- `int getColumnCount()`
  Возвращает количество столбцов для текущего результирующего набора типа `ResultSet`.
- `int getColumnDisplaySize(int column)`
  Возвращает максимальную ширину столбца по указанному целочисленному индексу.
  Параметры: `column` Номер столбца
- `String getColumnLabel(int column)`
  Возвращает предполагаемый заголовок для указанного столбца.
  Параметры: `column` Номер столбца
- `String getColumnName(int column)`
  Возвращает имя столбца по указанному целочисленному индексу.
  Параметры: `column` Номер столбца

Транзакции

Группа команд может быть оформлена в виде *транзакции*, которая может быть *зафиксирована* после успешного выполнения всех команд или *откачена*, если при выполнении хотя бы одной из команд произойдет какая-нибудь ошибка. Основной причиной для группирования команд в транзакции служит сохранение *целостности базы данных*. Допустим, требуется перевести денежные средства с одного банковского счета на другой. В этом случае нужно одновременно снять нужную сумму денег с одного счета и пополнить ею другой счет. Если после снятия суммы денег с одного счета, но

перед пополнением другого произойдет системная ошибка, то операция с первым счетом должна быть отменена.

Команды обновления базы данных могут быть сгруппированы в одну транзакцию. Если транзакция завершается полностью, то возможна ее *фиксация*. А если она не завершается полностью из-за каких-нибудь ошибок, то производится ее *откат*, т.е. отменяются все изменения в базе данных, которые выполнялись после предыдущей зафиксированной транзакции.

По умолчанию соединение с базой данных находится в режиме *автоматической фиксации*, т.е. результат выполнения каждой команды SQL фиксируется в базе данных после успешного завершения этой команды. Как только команда будет зафиксирована, откатить ее уже нельзя. Для отключения режима автоматической фиксации можно вызвать следующий метод:

```
conn.setAutoCommit(false);
```

Отключив автоматическую фиксацию, можно приступать к созданию объекта `Statement` обычным образом:

```
Statement stat = conn.createStatement();
```

Затем метод `executeUpdate()` вызывается нужное количество раз, как показано ниже.

```
stat.executeUpdate(команда1);
stat.executeUpdate(команда2);
stat.executeUpdate(команда3);
. . .
```

Если все эти команды завершатся успешно, то результаты их выполнения фиксируются. Для этого вызывается метод `commit()` следующим образом:

```
conn.commit();
```

А если при выполнении любой из этих команд произойдет ошибка, то производится откат всей транзакции. И для этого вызывается метод `rollback()` следующим образом:

```
conn.rollback();
```

При этом автоматически отменяются все команды, выполнявшиеся после фиксации последней транзакции. Откат обычно производится в том случае, если при выполнении транзакции генерируется исключение типа `SQLException`.

Точки сохранения

Некоторые драйверы позволяют повысить уровень контроля над процессом отката с помощью *точек сохранения*. При создании точки сохранения отмечается точка, в которую можно впоследствии вернуться, не отменяя всю транзакцию в целом. В приведенном ниже фрагменте кода показано, как это осуществляется на практике.

```
Statement stat = conn.createStatement(); // начать транзакцию;
    // переход в эту точку происходит при вызове метода rollback()
stat.executeUpdate(команда1);
Savepoint svpt = conn.setSavepoint(); // установить точку сохранения;
    // переход в эту точку происходит при вызове метода rollback(svpt)
stat.executeUpdate(команда2);
if (. . .) conn.rollback(svpt); // отменить результат выполнения
                                // команды2
. . .
conn.commit();
```

Если точка сохранения больше не нужна, ее нужно освободить следующим образом:

```
conn.releaseSavepoint(svpt);
```

Групповые обновления

Допустим, в программе требуется выполнить много команд INSERT для заполнения таблицы базы данных. Повысить производительность такой программы можно с помощью *группового обновления*. При групповом обновлении команды собираются вместе и выдаются группой, а не по отдельности.

НА ЗАМЕТКУ! Для того чтобы выяснить, поддерживается ли в базе данных групповое обновление, достаточно вызвать метод `supportsBatchUpdates()` из интерфейса `DatabaseMetaData`.

Помимо команд управления данными INSERT, UPDATE и DELETE, для группового обновления можно также использовать команды определения данных, в том числе CREATE TABLE и DROP TABLE. Но для этой цели не подходит команда SELECT, поскольку ее выполнение в группе с другими командами приводит к исключению. (В принципе команду SELECT не имеет смысла выполнять в групповом режиме, поскольку она возвращает результирующий набор, не обновляя базу данных.)

Для группового обновления сначала создается объект типа `Statement`:

```
Statement stat = conn.createStatement();
```

Затем вместо вызова метода `executeUpdate()` вызывается метод `addBatch()`:

```
String command = "CREATE TABLE . . ."
stat.addBatch(команда);

while (. . .)
{
  command = "INSERT INTO . . . VALUES (" + . . . + ")";
  stat.addBatch(команда);
}
```

И наконец, все команды выдаются вместе для группового обновления базы данных, как показано ниже. Метод `executeBatch()` возвращает массив подсчетов строк, обработанных при выполнении каждой командой и данной группы.

```
int[] counts = stat.executeBatch();
```

Для правильной обработки ошибок в групповом режиме групповое обновление следует рассматривать как единую транзакцию. Если в ходе группового обновления произойдет сбой или возникнет ошибка, то необходимо произвести откат в исходное состояние. Прежде всего следует отключить режим автоматической фиксации, собрать команды в группу, выполнить их и зафиксировать результаты, а затем восстановить режим автоматической фиксации, как выделено полужирным в приведенном ниже фрагменте кода.

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.getStatement();
. . .
// продолжать вызовы метода stat.addBatch(. . .);
. . .
stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```

java.sql.Connection 1.1

- `boolean getAutoCommit()`
- `void setAutoCommit(boolean b)`

  Получают или устанавливают режим автоматической фиксации для данного соединения с базой данных. Если параметр *b* принимает логическое значение `true`, то результаты выполнения всех команд автоматически фиксируются после их завершения.
- `void commit()`

  Фиксирует все команды, которые были выданы с момента последней фиксации.
- `void rollback()`

  Производит откат, отменяя все изменения, которые были внесены с момента последней фиксации.
- `Savepoint setSavepoint()` **1.4**
- `Savepoint setSavepoint(String name)` **1.4**

  Устанавливают безымянную или именованную точку сохранения.
- `void rollback(Savepoint svpt)` **1.4**

  Производит откат всех команд до указанной точки сохранения.
- `void releaseSavepoint(Savepoint svpt)` **1.4**

  Удаляет указанную точку сохранения.

java.sql.Savepoint 1.4

- `int getSavepointId()`

  Возвращает идентификатор данной безымянной точки сохранения. Если данная точка оказывается именованной, то генерируется исключение типа `SQLException`.
- `String getSavepointName()`

  Возвращает имя точки сохранения. Если данная точка оказывается безымянной, то генерируется исключение типа `SQLException`.

java.sql.Statement 1.1

- `void addBatch(String command)` **1.2**

  Включает указанную команду в текущую группу для группового обновления.
- `int[] executeBatch()` **1.2**

  Выполняет все команды из текущей группы. Каждое значение в возвращаемом массиве соответствует одной из команд, входящих в данную группу. Если это положительное значение, то оно обозначает подсчет обновленных строк. Если же это значение `SUCCESS_NO_INFO`, то оно обозначает, что команду удалось выполнить, но подсчет обновленных строк недоступен. А если это значение `EXECUTE_FAILED`, то оно обозначает, что выполнить команду не удалось.

java.sql.DatabaseMetaData 1.1

- `boolean supportsBatchUpdates()` **1.2**

  Возвращает логическое значение `true`, если драйвер поддерживает групповое обновление.

Расширенные типы данных SQL

В табл. 4.8 перечислены типы данных SQL, поддерживаемых в JDBC, а также их эквиваленты в Java.

Таблица 4.8. Типы данных SQL и соответствующие им типы в Java

Тип данных SQL	Тип данных Java
INTEGER или INT	int
SMALLINT	short
NUMERIC(*m*, *n*), DECIMAL(*m*, *n*) или DEC(*m*, *n*)	java.math.BigDecimal
FLOAT(*n*)	double
REAL	float
DOUBLE	double
CHARACTER(*n*) или CHAR(*n*)	String
VARCHAR(*n*), LONG VARCHAR	String
BOOLEAN	boolean
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
ROWID	java.sql.RowId
NCHAR(*n*), NVARCHAR(*n*), LONG NVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

Тип ARRAY представляет в SQL последовательность значений. Например, таблица Student может иметь столбец с оценками Scores типа ARRAY OF INTEGER, т.е. с массивом целочисленных значений. Для возвращения данных типа java.sql.Array служит метод getArray(). В интерфейсе java.sql.Array предусмотрены также методы извлечения значений из массива.

При получении большого объекта (LOB) или массива из базы данных конкретное содержимое извлекается из нее только после запроса отдельных значений. Это сделано для повышения эффективности работы с базой данных, поскольку большой объект может оказаться довольно объемистым.

В некоторых базах данных поддерживаются значения типа ROWID, описывающие местонахождение строки, что позволяет очень быстро извлечь ее из таблицы. В версии JDBC 4 внедрен интерфейс java.sql.RowId, предоставляющий методы для ввода идентификатора строки в запросы и извлечения его из получаемых результатов.

Строка национальных символов (тип NCHAR и его варианты) служит для хранения символьных строк в локальной кодировке, а также для их сортировки по заданным

условиям локальной сортировки. В версии JDBC 4 предоставляются методы для взаимного преобразования объектов Java типа `String` и строк национальных символов в запросах и получаемых результатах.

В некоторых базах данных допускается хранение данных, типы которых определяются пользователем. В версии JDBC 3 поддерживается механизм автоматического преобразования структурированных типов данных SQL в объекты Java. Кроме того, в некоторых базах данных предоставляется собственный механизм хранения данных в формате XML. В версии JDBC 4 внедрен интерфейс `SQLXML`, который может служить связующим звеном между внутренним представлением данных в формате XML и интерфейсами `Source/Result` модели DOM, а также двоичными потоками ввода-вывода. Дополнительные сведения об интерфейсе `SQLXML` можно найти в документации на соответствующий прикладной интерфейс API.

На этом рассмотрение расширенных типов данных SQL в этой главе завершается. Дополнительные сведения по этому вопросу можно найти в упоминавшейся ранее книге *JDBC™ API Tutorial and Reference, Third Edition* и спецификациях JDBC 4.

Управление соединениями с базами данных в веб-приложениях и производственных приложениях

Описанный ранее способ соединения с базой данных с помощью параметров из файла свойств `database.properties` подходит только для очень простых тестовых программ и совершенно не годится для крупномасштабных приложений. При установке приложения JDBC в корпоративной среде соединения с базами данных поддерживаются через интерфейс JNDI (Java Naming and Directory Interface — интерфейс для служб каталогов и именования в Java). Свойства источников данных в пределах всего предприятия хранятся в отдельном каталоге. Благодаря этому обеспечивается централизованное управление именами пользователей, паролями и URL в JDBC. В такой среде для установления соединения с базой данных рекомендуется использовать код, подобный следующему:

```
Context jndiContext = new InitialContext();
DataSource source =
    (DataSource) jndiContext.lookup("java:comp/env/jdbc/corejava");
Connection conn = source.getConnection();
```

Обратите внимание на то, что в этом коде уже не используется диспетчер драйверов типа `DriverManager`. Вместо него для поиска *источника данных* применяется служба JNDI. В качестве источника данных служит интерфейс `DataSource`, позволяющий устанавливать простые соединения типа JDBC, а также выполнять ряд более сложных функций, например, распределенные транзакции с несколькими базами данных. Интерфейс `DataSource` входит в пакет `javax.sql`, расширяющий стандартную библиотеку Java.

НА ЗАМЕТКУ! В контейнере Java EE не нужно даже программировать поиск в службе JNDI. Достаточно сделать следующую аннотацию `Resource` к полю типа `DataSource`, чтобы во время загрузки приложения был автоматически задан эталонный источник данных:

```
@Resource(name="jdbc/corejava")
private DataSource source;
```

Очевидно, что источник данных нуждается в настройке. Так, если прикладная программа для работы с базой данных должна выполняться в контейнере сервлетов (например, Apache Tomcat) или на сервере приложений (например, GlassFish), то сведения о настройке базы данных (в том числе имя службы JNDI, URL в JDBC, имя пользователя и пароль) целесообразно разместить в конфигурационном файле или же задать в ГПИ администратора.

Управление именами пользователей и регистрационными данными — это лишь один из вопросов, требующих особого внимания. Второй вопрос связан со стоимостью установлениях соединений с базами данных. В примерах программ, представленных в этой главе, применяются две методики для получения требующегося соединения с базой данных. Так, в самом начале программы `QueryDB` из листинга 4.3 устанавливается единственное соединение с базой данных, которое разрывается по завершении программы. А в программе `ViewDB` из листинга 4.4 новое соединение устанавливается всякий раз, когда в нем возникает потребность.

Однако ни одну из этих методик нельзя считать удовлетворительной. Ведь соединения с базами данных — это конечный ресурс. И если пользователь приостановит работу с приложением на некоторое время, то соединение не следует оставлять установленным. А с другой стороны, получение соединения для каждого запроса и последующий его разрыв обходится очень дорого.

В качестве выхода из этого положения целесообразно организовать *пул* соединений. Это означает, что соединения с базами данных не разрываются физически, а сохраняются в очереди и повторно используются для других запросов. Организация пулов соединений является важной служебной функцией, и поэтому в спецификации JDBC предоставляются вспомогательные средства для ее реализации. Следует, однако, иметь в виду, что в различных базах данных пул соединений может быть реализован по-разному. Причем он не всегда входит в состав драйвера JDBC. Некоторые поставщики веб-контейнеров и серверов приложений предлагают реализации пулов соединений в составе своих приложений.

Использование пула соединений полностью прозрачно для программиста. Для того чтобы извлечь соединение из пула, достаточно получить соответствующий источник данных и вызвать метод `getConnection()`. А по окончании работы с базой данный через это соединение следует вызвать метод `close()`. При этом соединение физически не разрывается, но в пул соединений поступает сообщение о том, что оно больше не требуется. Как правило, в пуле соединений принимаются необходимые меры к тому, чтобы сохранить в нем и подготовленные операторы.

Итак, вы ознакомились с самыми основами JDBC, которые требуются для разработки простых прикладных программ, взаимодействующих с базами данных. Но, как отмечалось в начале этой главы, базы данных могут иметь очень сложную структуру, а более сложные вопросы работы с ними выходят за рамки рассмотрения данной книги. Поэтому интересующихся подобными вопросами еще раз отсылаем к книге *JDBC™ API Tutorial and Reference, Third Edition* и спецификациям JDBC.

Из этой главы вы узнали о том, каким образом организуется взаимодействие с реляционными базами данных в Java. А следующая глава посвящена еще одной важной теме — интернационализации. В ней будет показано, как сделать прикладные программы пригодными для пользователей во всех странах мира.

ГЛАВА 5

Интернационализация

В этой главе...

- Региональные настройки
- Числовые форматы
- Форматирование даты и времени
- Сортировка
- Форматирование сообщений
- Текстовые файлы и наборы символов
- Комплекты ресурсов
- Пример интернационализации прикладной программы

Создавая приложение или аплет, разработчики надеются, что их программным продуктом заинтересуются многие пользователи. К тому же благодаря Интернету преодолеваются границы между разными странами. С другой стороны, если разработчики не принимают во внимание международных пользователей, то сами создают искусственные препятствия для широкого распространения программных продуктов по всему миру.

Java стал первым языком программирования, в котором изначально были предусмотрены средства интернационализации. Строки в Java формируются из символов в уникоде (Unicode). Поддержка этого стандарта кодирования символов позволяет разрабатывать программы на Java, способные обрабатывать тексты на любом из языков, существующих в мире.

Многие программисты считают, что для интернационализации своих приложений им достаточно воспользоваться уникодом и перевести на нужный язык все сообщения пользовательского интерфейса. Но этого явно недостаточно, потому что интернационализация программы означает нечто большее, чем поддержка кодировки символов в уникоде. Дата, время, денежные суммы и даже числа могут по-разному

представляться на различных языках. Необходимо также найти простой способ настройки команд меню, надписей на кнопках, сообщений и комбинаций клавиш на выбранный язык и региональный стандарт.

В этой главе рассматриваются способы интернационализации приложений и аплетов на Java, а также особенности представления времени, даты, чисел, текста и элементов ГПИ с учетом региональных стандартов. Кроме того, в ней обсуждаются некоторые инструментальные средства, предназначенные для интернационализации программ. А в конце этой главы в качестве примера будет рассмотрена программа калькуляции пенсионных сбережений с пользовательским интерфейсом на английском, немецком и китайском языках.

Региональные настройки

Приложение, которое адаптировано для международного рынка, легко определить по возможности выбора языка, используемого для работы с ним. Но профессионально адаптированные приложения могут иметь разные региональные настройки даже для тех стран, в которых используется одинаковый язык. Эту ситуацию очень точно подметил некогда Оскар Уайльд: "Теперь у нас, действительно, все, как в Америке, естественно, кроме языка".

В любом случае команды меню, надписи на кнопках и программные сообщения должны быть переведены на местный язык, возможно, с использованием другого алфавита. Существует еще много других, более тонких различий. Например, в Англии и Германии для представления десятичных чисел применяются разные форматы. Число, понятное англичанам в формате **123,456.78**, должно быть отображено в формате **123.456,78** для пользователей из Германии. Иными словами, точка и запятая в качестве разделителя дробной части и разделителя групп используются *по-разному* в этих странах!

Похожие различия можно заметить и в способах представления даты. В США привыкли отображать даты в формате месяц/день/год, в Германии используют более практичный порядок — день/месяц/год, а в Китае все наоборот — год/месяц/день. Таким образом, дата **3/22/61** должна быть представлена в формате **22.03.1961** для немецкого пользователя. Безусловно, если названия месяцев записаны полностью, то разница в способах представления дат становится еще более очевидной. Например, дата `March` **22, 1961,** понятная для американского пользователя, должна быть представлена как **22. März 1961** для немецкого пользователя.

Имеется ряд классов, выполняющих форматирование, принимая во внимание упомянутые выше региональные отличия. Для управления процессом форматирования служит класс `Locale`. *Региональные настройки* определяют следующее:

- язык;
- географическое местоположение (как правило);
- письмо (дополнительно; поддерживается с версии Java SE 7);
- вариант языка, определяющий различные особенности, включая диалекты и орфографические правила (дополнительно).

Например, в США используются следующие региональные настройки:

```
язык=английский, географическое местоположение=Соединенные Штаты
```

А в Германии региональные настройки такие:

```
язык=немецкий, географическое местоположение=Германия
```

В Швейцарии употребляются четыре официальных языка: немецкий, французский, итальянский и ретороманский. Поэтому для немецкоговорящих пользователей в Швейцарии, вероятнее всего, подойдут следующие региональные настройки:

```
язык=немецкий, географическое местоположение=Швейцария
```

При таких региональных настройках текст, даты и числа будут форматироваться таким же образом, как и в Германии, но денежные суммы будут отображаться в швейцарских франках, а не в евро.

Если же указать в региональных настройках только язык, как показано ниже, то особенности конкретной страны (например, формат представления денежных единиц) не будут учтены.

```
язык=немецкий
```

Для выражения местного языка и географического местоположения в компактной и стандартной форме в Java применяются коды, определенные Международной организацией по стандартизации (ISO). Язык обозначается двумя строчными буквами по стандарту ISO 639-1, а страна (географическое местоположение) — двумя прописными буквами по стандарту ISO 3166-1. В табл. 5.1 и 5.2 приведены наиболее употребительные коды языков и стран.

Таблица 5.1. Коды языков мира по стандарту ISO 639-1

Язык	Код
Английский	**en**
Голландский	**nl**
Греческий	**el**
Датский	**da**
Испанский	**sp**
Итальянский	**it**
Китайский	**zh**
Корейский	**ko**
Немецкий	**de**
Норвежский	**no**
Русский	**ru**
Португальский	**pt**
Турецкий	**tr**
Украинский	**uk**
Финский	**fi**
Французский	**fr**
Шведский	**sv**
Японский	**ja**

Таблица 5.2. Коды стран мира по стандарту ISO 3166-1

Страна	Код
Австрия	**AT**
Бельгия	**BE**
Великобритания	**GB**
Германия	**DE**
Греция	**GR**
Дания	**DK**
Ирландия	**IE**
Испания	**ES**
Италия	**IT**
Канада	**CA**
Китай	**CN**
Корея	**KR**
Нидерланды	**NL**
Норвегия	**NO**
Португалия	**PT**
Россия	**RU**
Соединенные Штаты Америки	**US**
Тайвань	**TW**
Турция	**TR**
Украина	**UK**
Финляндия	**FI**
Швейцария	**CH**
Швеция	**SE**
Япония	**JP**

Исходя из кода языка и страны мира, можно установить региональные настройки, построив объект типа `Locale`, как показано в приведенном ниже фрагменте кода.

```
Locale german = new Locale("de");
Locale germanGermany = new Locale("de", "DE");
Locale germanSwitzerland = new Locale("de", "CH");
```

Для удобства разработчиков в Java SE предусмотрен целый ряд предопределенных объектов с региональными настройками. Ниже перечислены некоторые из них.

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
```

```
Locale.PRC
Locale.TAIWAN
Locale.UK
Locale.US
```

Для языков мира в Java SE имеется также целый ряд предопределенных объектов, позволяющих задать в региональных настройках язык, не указывая страну. Ниже перечислены некоторые из них.

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

НА ЗАМЕТКУ! Если в региональных настройках требуется задать не только язык и страну, то придется досконально изучить стандарт IETF BCP 47 (с отличным введением в этот стандарт можно ознакомиться по адресу `www.w3.org/International/articles/language-tags`). С версии Java SE 7 в Java поддерживаются языковые дескрипторы по стандарту IETF BCP 47. Так, если требуется построить объект с региональными настройками по языковому дескриптору для китайскоязычных пользователей из Гонконга, достаточно сделать следующий вызов:

```
Locale chineseTraditionalCharactersHongKong = Locale.
forLanguageTag("zh-Hant-HK");
```

Помимо вызова конструктора или выбора предопределенных объектов, имеются еще два способа для получения объектов с региональными настройками. В частности, статический метод `getDefault()` из класса `Locale` позволяет определить региональные настройки, которые используются в операционной системе по умолчанию. Изменить эти настройки по умолчанию можно, вызвав метод `setDefault()`, но не следует забывать, что действие этого метода распространяется только на прикладную программу на Java, а не на всю операционную систему в целом.

И наконец, все зависимые от региональных стандартов вспомогательные классы могут возвращать массив поддерживаемых региональных настроек. Например, приведенный ниже метод возвращает все региональные настройки, поддерживаемые в классе `DateFormat`.

```
Locale[] supportedLocales = DateFormat.getAvailableLocales();
```

СОВЕТ. Для проверки интернационализации своей программы попробуйте сменить региональные настройки по умолчанию. С этой целью укажите язык и страну при запуске программы на выполнение. Например, в приведенной ниже команде запуска программы задается немецкий язык для Швейцарии.

```
java -Duser.language=de -Duser.region=CH МояПрограмма
```

Что же можно сделать с полученными региональными настройками? Оказывается, не так уж и много. Единственную пользу можно извлечь из тех методов в классе `Locale`, которые определяют коды языка и страны. Наиболее важным из них является метод `getDisplayName()`, который возвращает символьную строку с описанием

региональной настройки. Она содержит не какие-то загадочные двухбуквенные коды, а вполне удобочитаемое описание, как в приведенном ниже примере.

```
German (Switzerland)
```

Но дело в том, что эта строка отображается на используемом по умолчанию языке, что далеко не всегда удобно. Так, если пользователь выбрал немецкий язык интерфейса, символьную строку описания региональной настройки следует отобразить именно на немецком языке, передав в качестве параметра соответствующую региональную настройку, как показано ниже.

```
Locale loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

В результате выполнения этого фрагмента кода описание региональной настройки будет выведено на указанном в ней языке:

```
Deutsch (Schweiz)
```

Из данного примера становится ясно, зачем нужны объекты типа `Locale`. Передавая их методам, способным реагировать на региональные настройки, можно отображать текст на языке, понятном пользователю. Примеры применения таких объектов рассматриваются далее в этой главе.

java.util.Locale 1.1

- `Locale(String` *language*`)`
- `Locale(String` *language*`, String` *country*`)`
- `Locale(String` *language*`, String` *country*`, String` *variant*`)`

  Создает объект типа `Locale` региональных настроек с учетом указанного языка, страны и варианта языка. Вместо вариантов языка в новом коде рекомендуется использовать языковые дескрипторы по стандарту IETF BCP 47.

- `static Locale forLanguageTag(String` *languageTag*`)` **7**

  Создает объект типа `Locale` региональных настроек по заданному языковому дескриптору.

- `static Locale getDefault()`

  Возвращает региональные настройки, используемые по умолчанию.

- `static void setDefault(Locale` *loc*`)`

  Задает региональные настройки, используемые по умолчанию.

- `String getDisplayName()`

  Возвращает символьную строку наименования с описанием текущей региональной настройки. Эта строка составляется на том языке, который указан в текущих региональных настройках.

- `String getDisplayName(Locale` *loc*`)`

  Возвращает символьную строку наименования с описанием заданных региональных настроек. Эта строка составляется на том языке, который указан в заданных региональных настройках.

- `String getLanguage()`

  Возвращает код языка (две строчные буквы) по стандарту ISO 639.

- `String getDisplayLanguage()`
  Возвращает название языка в формате текущих региональных настроек.
- `String getDisplayLanguage(Locale` *loc*`)`
  Возвращает название языка в формате заданных региональных настроек.
- `String getCountry()`
  Возвращает код страны (две прописные буквы) по стандарту ISO 3166.
- `String getDisplayCountry()`
  Возвращает название страны в формате текущих региональных настроек.
- `String getDisplayCountry(Locale` *loc*`)`
  Возвращает название страны в формате заданных региональных настроек.
- `String toLanguageTag()` **7**
  Возвращает языковой дескриптор по стандарту IETF BCP 47 для текущих региональных настроек, например `"de-CH"` (немецкий язык, Швейцария).
- `String toString()`
  Возвращает описание региональных настроек в виде кодов языка и страны, разделенных знаком подчеркивания (например, `"de_CH"`, т.е. немецкий язык, Швейцария). Этим методом рекомендуется пользоваться только для отладки.

Числовые форматы

Как упоминалось выше, в разных странах и регионах применяются различные способы представления чисел и денежных сумм. В пакете `java.text` содержатся классы, позволяющие форматировать числа и выполнять синтаксический анализ их строкового представления. Для форматирования числа в соответствии с конкретными региональными настройками необходимо выполнить следующие действия.

1. Получить объект региональных настроек, как пояснялось в предыдущем разделе.
2. Вызвать фабричный метод для получения форматирующего объекта.
3. Применить форматирующий объект для форматирования числа или синтаксического анализа его строкового представления.

В качестве фабричных служат статические методы `getNumberInstance()`, `getCurrencyInstance()` и `getPercentInstance()` из класса `NumberFormat`. Они получают в качестве параметра объект типа `Locale` и возвращают объекты, предназначенные для форматирования чисел, денежных сумм и числовых величин в процентах. Например, для выражения денежной суммы в формате, принятом в Германии, служит приведенный ниже фрагмент кода.

```
Locale loc = new Locale("de", "DE");
NumberFormat currFmt = NumberFormat.getCurrencyInstance(loc);
double amt = 123456.78;
String result = currFmt.format(amt);
```

В результате выполнения этого фрагмента кода получается следующая символьная строка:

```
123.456,78 €
```

Для обозначения европейской валюты в данном случае используется знак €, который располагается в конце строки. Обратите также внимание на местоположение знаков, обозначающих дробную часть числа и разделяющих десятичные разряды в его целой части. Порядок их следования обратный принятому в англоязычных странах.

Рассмотрим обратную задачу преобразования в число символьной строки, составленной в соответствии с конкретными региональными настройками. Для этого предусмотрен метод `parse()`, выполняющий синтаксический анализ символьной строки, автоматически используя заданные по умолчанию региональные настройки. В приведенном ниже примере кода демонстрируется преобразование в число символьной строки, введенной пользователем в текстовом поле. Метод `parse()` способен преобразовывать символьные строки в числа, где в качестве разделителей используются точки и запятые в соответствии с принятыми региональными стандартами.

```
TextField inputField;
. . .
NumberFormat fmt = NumberFormat.getNumberInstance();
// получить форматирующий числа объект для региональных настроек,
// используемых по умолчанию
Number input = fmt.parse(inputField.getText().trim());
double x = input.doubleValue();
```

Метод `parse()` возвращает результат абстрактного типа `Number`. На самом деле возвращаемый объект является экземпляром класса `Long` или `Double` в зависимости от того, представляет ли исходная символьная строка целое число или же число с плавающей точкой. Если это не так важно, то для получения упакованного числового значения достаточно вызвать метод `doubleValue()` из класса `Number`.

ВНИМАНИЕ! Для объектов типа `Number` не поддерживается автоматическая распаковка, поэтому их нельзя присвоить непосредственно переменным примитивных типов. Вместо этого нужно явным образом вызвать метод `doubleValue()` или `intValue()`.

Если число представлено в неверном формате, генерируется исключение типа `ParseException`. Например, не допускается наличие пробелов в начале символьной строки, преобразуемой в число. (Для их удаления следует вызвать метод `trim()`.) Но любые символы, следующие в строке после числа, просто игнорируются, и поэтому исключение в этом случае не возникает.

Следует иметь в виду, что классы, возвращаемые фабричными методами типа `getXxxInstance()`, являются экземплярами не абстрактного класса `NumberFormat`, а одного из его подклассов. Фабричным методам известно только, как найти объект, относящийся к конкретным региональным настройкам.

Для получения списка поддерживаемых региональных настроек можно вызвать статический метод `getAvailableLocales()`. Этот метод возвращает массив региональных настроек, для которых могут быть получены форматирующие объекты.

В примере программы, рассматриваемой в этом разделе, предоставляется возможность поэкспериментировать с разными способами форматирования чисел, как показано на рис. 5.1. В верхней части рабочего окна этой программы находится список всех доступных региональных настроек со средствами форматирования чисел. Ниже этого списка расположена группа кнопок-переключателей, где можно выбрать способ форматирования чисел, денежных сумм или числовых величин в процентах. После выбора новой региональной настройки или способа форматирования чисел число в текстовом поле автоматически переформатируется.

Просмотрев лишь несколько вариантов форматирования чисел и денежных сумм в зависимости от выбранных региональных настроек, пользователь, несомненно, составит ясное представление о разнообразии существующих форматов чисел. В текстовом поле можно ввести любое число и щелкнуть на кнопке Parse, в результате чего будет вызван метод `parse()`, выполняющий синтаксический анализ введенной символьной строки. При удачном исходе синтаксического анализа результат передается методу `format()`, а затем отображается на экране. В противном случае в текстовом поле появляется сообщение `"Parse error"` (Ошибка синтаксического анализа).

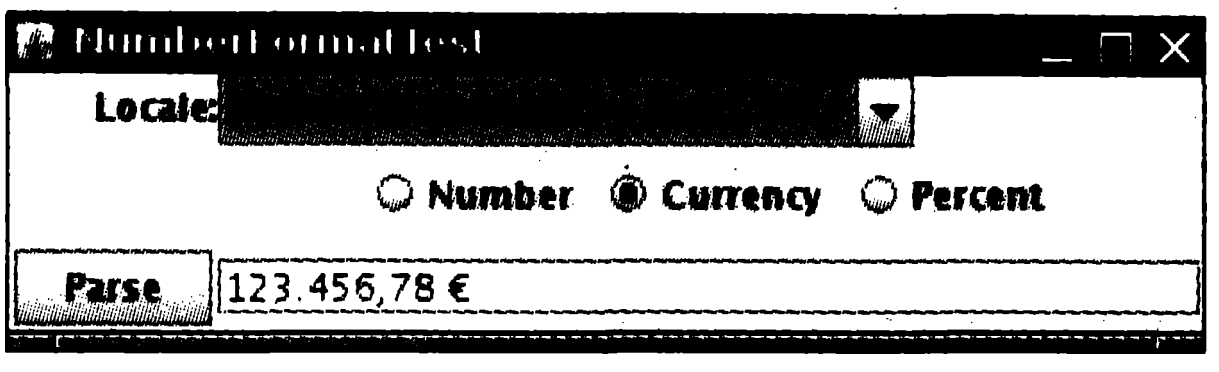


Рис. 5.1. Рабочее окно программы `NumberFormatTest`

Как показано в листинге 5.1, исходный код рассматриваемой здесь программы имеет довольно простую структуру. Сначала в конструкторе вызывается метод `NumberFormat.getAvailableLocales()`. Затем для каждого вида поддерживаемых региональных настроек вызывается метод `getDisplayName()`, а возвращаемые этим методом результаты вводятся в список. (Символьные строки не сортируются; подробнее об этом речь пойдет далее, в разделе "Сортировка".) Всякий раз, когда пользователь выбирает другие региональные настройки или способ форматирования чисел, создается новый форматирующий объект и обновляется содержимое текстового поля. Если пользователь щелкнет на кнопке Parse, то вызывается метод `parse()`, преобразующий в число символьную строку в соответствии с выбранными региональными настройками.

Листинг 5.1. Исходный код из файла `numberFormat/NumberFormatTest.java`

```
1   package numberFormat;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.text.*;
6   import java.util.*;
7
8   import javax.swing.*;
9   /**
10   * В этой программе демонстрируется форматирование чисел
11   * при разных региональных настройках
12   * @version 1.13 2007-07-25
13   * @author Cay Horstmann
14  */
15  public class NumberFormatTest
16  {
17    public static void main(String[] args)
18    {
19      EventQueue.invokeLater(new Runnable()
20        {
21          public void run()
22          {
```

```
23              JFrame frame = new NumberFormatFrame();
24              frame.setTitle("NumberFormatTest");
25              frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26              frame.setVisible(true);
27           }
28         });
29    }
30 }
31
32 /**
33  * Этот фрейм содержит кнопки-переключатели для выбора способа
34  * форматирования чисел, комбинированный список для выбора
35  * региональных настроек, текстовое поле для отображения
36  * отформатированного числа, а также кнопку для активизации
37  * синтаксического анализа содержимого текстового поля
38 */
39 class NumberFormatFrame extends JFrame
40 {
41    private Locale[] locales;
42    private double currentNumber;
43    private JComboBox<String> localeCombo = new JComboBox<>();
44    private JButton parseButton = new JButton("Parse");
45    private JTextField numberText = new JTextField(30);
46    private JRadioButton numberRadioButton = new JRadioButton("Number");
47    private JRadioButton currencyRadioButton = new JRadioButton("Currency");
48    private JRadioButton percentRadioButton = new JRadioButton("Percent");
49    private ButtonGroup rbGroup = new ButtonGroup();
50    private NumberFormat currentNumberFormat;
51
52    public NumberFormatFrame()
53    {
54       setLayout(new GridBagLayout());
55
56       ActionListener listener = new ActionListener()
57          {
58             public void actionPerformed(ActionEvent event)
59             {
60                updateDisplay();
61             }
62          };
63
64       JPanel p = new JPanel();
65       addRadioButton(p, numberRadioButton, rbGroup, listener);
66       addRadioButton(p, currencyRadioButton, rbGroup, listener);
67       addRadioButton(p, percentRadioButton, rbGroup, listener);
68
69       add(new JLabel("Locale:"), new GBC(0, 0).setAnchor(GBC.EAST));
70       add(p, new GBC(1, 1));
71       add(parseButton, new GBC(0, 2).setInsets(2));
72       add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
73       add(numberText, new GBC(1, 2).setFill(GBC.HORIZONTAL));
74       locales = (Locale[]) NumberFormat.getAvailableLocales().clone();
75       Arrays.sort(locales, new Comparator<Locale>()
76          {
77             public int compare(Locale l1, Locale l2)
78             {
79                return l1.getDisplayName().compareTo(l2.getDisplayName());
80             }
81          });
```

```
82        for (Locale loc : locales)
83        localeCombo.addItem(loc.getDisplayName());
84        localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
85        currentNumber = 123456.78;
86        updateDisplay();
87
88        localeCombo.addActionListener(listener);
89
90        parseButton.addActionListener(new ActionListener()
91          {
92            public void actionPerformed(ActionEvent event)
93            {
94              String s = numberText.getText().trim();
95              try
96              {
97                Number n = currentNumberFormat.parse(s);
98                if (n != null)
99                {
100                 currentNumber = n.doubleValue();
101                 updateDisplay();
102               }
103               else
104               {
105                 numberText.setText("Parse error: " + s);
106               }
107             }
108             catch (ParseException e)
109             {
110               numberText.setText("Parse error: " + s);
111             }
112           }
113         });
114       pack();
115     }
116
117     /**
118      * Вводит кнопки-переключатели в контейнер
119      * @param p Контейнер для размещения кнопок-переключателей
120      * @param b Кнопка-переключатель
121      * @param g Группа кнопок-переключателей
122      * @param listener Приемник событий от кнопок-переключателей
123     */
124     public void addRadioButton(
125       Container p, JRadioButton b, ButtonGroup g, ActionListener listener)
126     {
127       b.setSelected(g.getButtonCount() == 0);
128       b.addActionListener(listener);
129       g.add(b);
130       p.add(b);
131     }
132
133     /**
134      * Обновляет отображаемое число и форматирует его
135      * в соответствии с пользовательскими установками
136     */
137     public void updateDisplay()
138     {
139       Locale currentLocale = locales[localeCombo.getSelectedIndex()];
140       currentNumberFormat = null;
```

```
141      if (numberRadioButton.isSelected()) currentNumberFormat =
142         NumberFormat.getNumberInstance(currentLocale);
143       else if (currencyRadioButton.isSelected()) currentNumberFormat =
144         NumberFormat.getCurrencyInstance(currentLocale);
145       else if (percentRadioButton.isSelected()) currentNumberFormat =
146         NumberFormat.getPercentInstance(currentLocale);
147       String n = currentNumberFormat.format(currentNumber);
148       numberText.setText(n);
149    }
150 }
```

java.text.NumberFormat 1.1

- `static Locale[] getAvailableLocales()`

  Возвращает массив объектов типа `Locale`, для которых доступны форматирующие объекты типа `NumberFormat`.

- `static NumberFormat getNumberInstance()`
- `static NumberFormat getNumberInstance(Locale l)`
- `static NumberFormat getCurrencyInstance()`
- `static NumberFormat getCurrencyInstance(Locale l)`
- `static NumberFormat getPercentInstance()`
- `static NumberFormat getPercentInstance(Locale l)`

  Возвращают объект, форматирующий числа, денежные суммы или числовые величины в процентах в соответствии с текущими или заданными региональными настройками.

- `String format(double x)`
- `String format(long x)`

  Возвращают символьную строку, получаемую в результате форматирования заданного числа с плавающей точкой или целого числа.

- `Number parse(String s)`

  Возвращает число, получаемое в результате синтаксического анализа символьной строки. Это число может иметь тип `Long` или `Double`, а символьная строка не должна начинаться с пробелов. Любые символы, следующие в строке после анализируемого числа, игнорируются. При неудачном исходе синтаксического анализа символьной строки генерируется исключение типа `ParseException`.

- `void setParseIntegerOnly(boolean b)`
- `boolean isParseIntegerOnly()`

  Устанавливают или получают признак, указывающий на то, что данный форматирующий объект предназначен для синтаксического анализа только целочисленных значений.

- `void setGroupingUsed(boolean b)`
- `boolean isGroupingUsed()`

  Устанавливают или получают признак, указывающий на то, что данный форматирующий объект предназначен для распознавания и разделения групп десятичных разрядов (например, **100,000**) в анализируемых числах.

- `void setMinimumIntegerDigits(int n)`
- `int getMinimumIntegerDigits()`
- `void setMaximumIntegerDigits(int n)`
- `int getMaximumIntegerDigits()`
- `void setMinimumFractionDigits(int n)`
- `int getMinimumFractionDigits()`
- `void setMaximumFractionDigits(int n)`
- `int getMaximumFractionDigits()`

  Устанавливают или получают максимальное или минимальное количество цифр в целой или дробной части числа.

Форматирование денежных сумм в разных валютах

Для форматирования денежных сумм служит метод `NumberFormat.getCurrencyInstance()`. Но этот метод не очень удобен, поскольку он возвращает форматирующий объект только для одной валюты. Допустим, выписывается счет-фактура для американского заказчика, в котором одни суммы представлены в долларах США, а другие в евро. Для решения этой задачи нельзя воспользоваться просто двумя форматирующими объектами, как показано ниже. Счет, в котором фигурируют такие суммы, как **$100,000** и **100.000€**, будет выглядеть не совсем обычно. Ведь при представлении сумм в евро для разделения групп разрядов используется точка, а сумм в долларах США — запятая.

```
NumberFormat dollarFormatter = NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance(Locale.
GERMANY);
```

Для управления форматированием денежных сумм в разных валютах лучше воспользоваться классом `Currency`. Сначала получается объект типа `Currency`, для чего статическому методу `Currency.getInstance()` передается идентификатор валюты. Затем вызывается метод `setCurrency()` для каждого форматирующего объекта. В приведенном ниже фрагменте показано, как настроить объект, форматирующий денежные суммы в евро, на американского заказчика.

```
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance(Locale.US);
euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

Идентификаторы валют определяются по стандарту ISO 4217 (http://www.currency-iso.org/en/home.html). Некоторые из них приведены в табл. 5.3.

Таблица 5.3. Идентификаторы валют

Валюта	Идентификатор
Доллар США	**USD**
Евро	**EUR**
Английский фунт	**GBP**
Японская иена	**JPY**
Китайский юань	**CNY**
Индийская рупия	**INR**
Российский рубль	**RUB**

java.util.Currency 1.4

- `static Currency getInstance(String` *`currencyCode`*`)`
- `static Currency getInstance(Locale` *`locale`*`)`

  Возвращают экземпляр класса `Currency`, соответствующий заданному коду валюты по стандарту ISO 4217 или стране, указанной в текущих региональных настройках.

- `String toString()`
- `String getCurrencyCode()`

  Получают код текущей валюты по стандарту ISO 4217.

- `String getSymbol()`
- `String getSymbol(Locale` *`locale`*`)`

  Получают форматирующий знак текущей валюты в соответствии с текущими или заданными региональными настройками. Например, доллар США (**USD**) может обозначаться как **$** или **US$** в зависимости от используемых региональных настроек.

- `int getDefaultFractionDigits()`

  Получает принятое по умолчанию количество цифр в дробной части денежной суммы в текущей валюте.

- `static Set<Currency> getAvailableCurrencies()` **7**

  Получает все имеющиеся валюты.

Форматирование даты и времени

При форматировании даты и времени в соответствии с региональными настройками необходимо иметь в виду четыре особенности.

- Названия месяцев и дней недели должны быть представлены на местном языке.
- Порядок указания года, месяца и числа отличается в разных странах и регионах.
- Для отображения дат может использоваться календарь, отличный от григорианского.
- Следует учитывать часовые пояса.

Для учета перечисленных выше особенностей форматирования дат и времени в Java предусмотрен класс `DateFormat`, который используется почти так же, как и класс `NumberFormat`. Сначала нужно получить региональные настройки. Для этого можно выбрать региональные настройки по умолчанию или вызвать статический метод `getAvailableLocales()`, чтобы получить массив региональных настроек, поддерживающих форматирование даты и времени. Затем вызывается один из следующих трех фабричных методов:

```
fmt = DateFormat.getDateInstance(dateStyle, loc);
fmt = DateFormat.getTimeInstance(timeStyle, loc);
fmt = DateFormat.getDateTimeInstance(dateStyle, timeStyle, loc);
```

Для указания нужного стиля в этих фабричных методах предусмотрен отдельный параметр, в качестве которого задается одна из следующих констант:

1. `DateFormat.DEFAULT`
2. `DateFormat.FULL` (например, **Wednesday, September 12, 2007 8:51:03 PM PDT** для региональных настроек США).
3. `DateFormat.LONG` (например, `September` **12, 2007 8:51:03 PM PDT** для региональных настроек США).
4. `DateFormat.MEDIUM` (например, **Sep 12, 2007 8:51:03 PM** для региональных настроек США).
5. `DateFormat.SHORT` (например, **9/12/07 8:51 PM** для региональных настроек США).

Фабричный метод возвращает форматирующий объект, который можно использовать для форматирования дат следующим образом:

```
Date now = new Date();
String s = fmt.format(now);
```

Для преобразования символьной строки в дату служит метод `parse()`, действующий аналогично одноименному методу из класса `NumberFormat`. Например, в приведенном ниже фрагменте кода производится синтаксический анализ и преобразование в дату символьной строки, введенной пользователем в текстовом поле, а полученный результат преобразуется в дату с учетом региональных настроек по умолчанию.

```
TextField inputField;
. . .
DateFormat fmt = DateFormat.getDateInstance(DateFormat.MEDIUM);
Date input = fmt.parse(inputField.getText().trim());
```

К сожалению, пользователь должен вводить дату строго в конкретном формате. Так, если установлен формат даты `MEDIUM` для региональных настроек США, то предполагается, что введенная дата должна иметь вид **Sep 12, 2007**. Но если пользователь введет дату **Sep 12 2007** (т.е. без запятой) или **9/12/07** (т.е. в кратком формате), то это приведет к исключению типа `ParseException`.

Для менее строгой интерпретации неточно указанных дат предусмотрен признак `lenient`. Если этот признак установлен, то неверно заданная дата **February 30, 2007** будет автоматически преобразована в дату **March 2, 2007**. Но такую интерпретацию дат вряд ли можно считать безопасной, поэтому данный признак рекомендуется сбросить. И если пользователь попытается ввести дату

в неверной последовательности дня, месяца и года, то в процессе синтаксического анализа и преобразования строки в дату будет сгенерировано исключение типа `IllegalArgumentException`.

В примере программы, исходный код которой приведен в листинге 5.2, демонстрируется применение класса `DateFormat` на практике. Эта программа позволяет выбирать различные региональные настройки и наблюдать за тем, как в разных странах мира форматируются дата и время. Если на компьютере не установлены необходимые для отображения шрифты, то вместо некоторых символов будут отображаться знаки вопроса, как, например, при выборе китайских региональных настроек. На рис. 5.2 показано рабочее окно данной программы после установки китайских шрифтов на компьютере. Как видите, даты выводятся на экран в правильном формате для китайских региональных настроек.

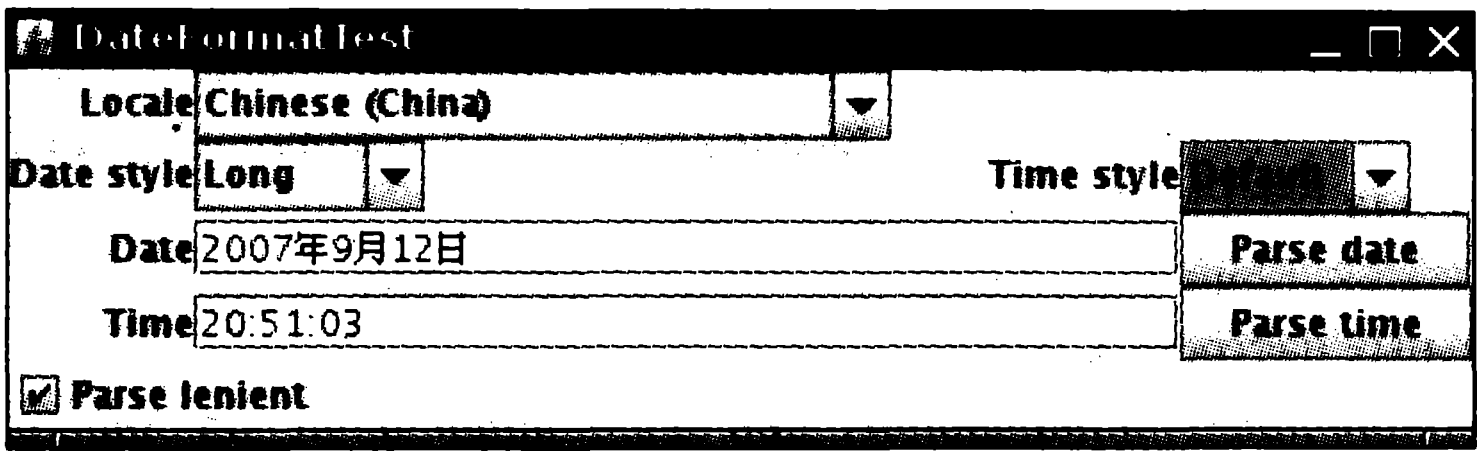


Рис. 5.2. Отображение даты на китайском языке в рабочем окне программы `DateFormatTest`

Листинг 5.2. Исходный код из файла `dateFormat/DateFormatTest.java`

```
1   package dateFormat;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.text.*;
6   import java.util.*;
7   import javax.swing.*;
8
9   /**
10   * В этой программе демонстрируется форматирование дат
11   * при выборе разных региональных настроек
12   * @version 1.13 2007-07-25
13   * @author Cay Horstmann
14  */
15  public class DateFormatTest
16  {
17    public static void main(String[] args)
18    {
19      EventQueue.invokeLater(new Runnable()
20        {
21          public void run()
22          {
23            JFrame frame = new DateFormatFrame();
24            frame.setTitle("DateFormatTest");
25            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26            frame.setVisible(true);
27          }
28        });
```

```
29    }
30  }
31
32  /**
33    * Этот фрейм содержит комбинированные списки для выбора региональных
34    * настроек и форматов даты и времени, текстовые поля для отображения
35    * отформатированных даты и времени, а также кнопки синтаксического
36    * анализа содержимого текстовых полей и флажок для установки режима
37    * нестрогой интерпретации вводимых дат и времени
38  */
39  class DateFormatFrame extends JFrame
40  {
41    private Locale[] locales;
42    private Date currentDate;
43    private Date currentTime;
44    private DateFormat currentDateFormat;
45    private DateFormat currentTimeFormat;
46    private JComboBox<String> localeCombo = new JComboBox<>();
47    private JButton dateParseButton = new JButton("Parse date");
48    private JButton timeParseButton = new JButton("Parse time");
49    private JTextField dateText = new JTextField(30);
50    private JTextField timeText = new JTextField(30);
51    private JCheckBox lenientCheckbox =
52        new JCheckBox("Parse lenient", true);
53    private EnumCombo dateStyleCombo = new EnumCombo(DateFormat.class,
54        "Default", "Full", "Long", "Medium", "Short");
55     private EnumCombo timeStyleCombo = new EnumCombo(DateFormat.class,
56        "Default", "Full", "Long", "Medium", "Short");
57
58    public DateFormatFrame()
59    {
60      setLayout(new GridBagLayout());
61      add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
62      add(new JLabel("Date style"), new GBC(0, 1).setAnchor(GBC.EAST));
63      add(new JLabel("Time style"), new GBC(2, 1).setAnchor(GBC.EAST));
64      add(new JLabel("Date"), new GBC(0, 2).setAnchor(GBC.EAST));
65      add(new JLabel("Time"), new GBC(0, 3).setAnchor(GBC.EAST));
66      add(localeCombo, new GBC(1, 0, 2, 1).setAnchor(GBC.WEST));
67      add(dateStyleCombo, new GBC(1, 1).setAnchor(GBC.WEST));
68      add(timeStyleCombo, new GBC(3, 1).setAnchor(GBC.WEST));
69      add(dateParseButton, new GBC(3, 2).setAnchor(GBC.WEST));
70      add(timeParseButton, new GBC(3, 3).setAnchor(GBC.WEST));
71      add(lenientCheckbox, new GBC(0, 4, 2, 1).setAnchor(GBC.WEST));
72      add(dateText, new GBC(1, 2, 2, 1).setFill(GBC.HORIZONTAL));
73      add(timeText, new GBC(1, 3, 2, 1).setFill(GBC.HORIZONTAL));
74
75      locales = (Locale[]) DateFormat.getAvailableLocales().clone();
76      Arrays.sort(locales, new Comparator<Locale>()
77        {
78          public int compare(Locale l1, Locale l2)
79          {
80            return l1.getDisplayName().compareTo(l2.getDisplayName());
81          }
82        });
83      for (Locale loc : locales)
84        localeCombo.addItem(loc.getDisplayName());
85      localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
86      currentDate = new Date();
87      currentTime = new Date();
```

```
88        updateDisplay();
89
90        ActionListener listener = new ActionListener()
91          {
92            public void actionPerformed(ActionEvent event)
93            {
94              updateDisplay();
95            }
96          };
97
98        localeCombo.addActionListener(listener);
99        dateStyleCombo.addActionListener(listener);
100       timeStyleCombo.addActionListener(listener);
101
102       dateParseButton.addActionListener(new ActionListener()
103         {
104           public void actionPerformed(ActionEvent event)
105           {
106             String d = dateText.getText().trim();
107             try
108             {
109               currentDateFormat.setLenient(lenientCheckbox.isSelected());
110               Date date = currentDateFormat.parse(d);
111               currentDate = date;
112               updateDisplay();
113             }
114             catch (ParseException e)
115             {
116               dateText.setText("Parse error: " + d);
117             }
118             catch (IllegalArgumentException e)
119             {
120               dateText.setText("Argument error: " + d);
121             }
122           }
123         });
124       timeParseButton.addActionListener(new ActionListener()
125         {
126           public void actionPerformed(ActionEvent event)
127           {
128             String t = timeText.getText().trim();
129             try
130             {
131               currentDateFormat.setLenient(lenientCheckbox.isSelected());
132               Date date = currentTimeFormat.parse(t);
133               currentTime = date;
134               updateDisplay();
135             }
136             catch (ParseException e)
137             {
138               timeText.setText("Parse error: " + t);
139             }
140             catch (IllegalArgumentException e)
141             {
142               timeText.setText("Argument error: " + t);
143             }
144           }
145         });
146       pack();
```

```
147    }
148
149    /**
150     * Обновляет отображаемые дату и время и форматирует
151     * их в соответствии с пользовательскими установками
152    */
153    public void updateDisplay()
154    {
155      Locale currentLocale = locales[localeCombo.getSelectedIndex()];
156      int dateStyle = dateStyleCombo.getValue();
157      currentDateFormat =
158          DateFormat.getDateInstance(dateStyle, currentLocale);
159      String d = currentDateFormat.format(currentDate);
160      dateText.setText(d);
161      int timeStyle = timeStyleCombo.getValue();
162      currentTimeFormat =
163      DateFormat.getTimeInstance(timeStyle, currentLocale);
164      String t = currentTimeFormat.format(currentTime);
165      timeText.setText(t);
166    }
167 }
```

Для проверки правильности синтаксического анализа и преобразования символьной строки в дату достаточно ввести дату или время в текстовом поле Date или Time соответственно, установить флажок Parse lenient (Нестрогий синтаксический анализ), а затем щелкнуть на кнопке Parse date (Произвести синтаксический анализ даты) или Parse time (Произвести синтаксический анализ времени).

В рассматриваемой здесь программе используется вспомогательный класс `EnumCombo`, исходный код которого приведен в листинге 5.3. Он служит для заполнения комбинированного списка значениями типа `Short`, `Medium` и `Long`, а также для автоматического преобразования выбранного пользователем варианта в целочисленное значение `DateFormat.SHORT`, `DateFormat.MEDIUM` или `DateFormat.LONG`. Вместо написания повторяющегося кода в данном случае применяется рефлексия. Символьная строка, соответствующая выбранному пользователем варианту, преобразуется в верхний регистр, пробелы заменяются символами подчеркивания, после чего определяется значение в статическом поле с этим именем. (Более подробно рефлексия рассматривается в главе 5 первого тома данной книги.)

Листинг 5.3. Исходный код из файла `dateFormat/EnumCombo.java`

```
1  package dateFormat;
2
3  import java.util.*;
4  import javax.swing.*;
5  /**
6   * Комбинированный список для выбора среди значений статических полей,
7   * имена которых задаются в конструкторе вспомогательного класса
8   * @version 1.14 2012-01-26
9   * @author Cay Horstmann
10 */
11 public class EnumCombo extends JComboBox<String>
12 {
13   private Map<String, Integer> table = new TreeMap<>();
14
15   /**
16    * Конструирует объект вспомогательного класса EnumCombo
```

```
17     * @param cl Класс
18     * @param labels Массив имен статических полей из класса cl
19    */
20    public EnumCombo(Class<?> cl, String... labels)
21    {
22      for (String label : labels)
23      {
24        String name = label.toUpperCase().replace(' ', '_');
25        int value = 0;
26        try
27        {
28          java.lang.reflect.Field f = cl.getField(name);
29          value = f.getInt(cl);
30        }
31        catch (Exception e)
32        {
33          label = "(" + label + ")";
34        }
35        table.put(label, value);
36        addItem(label);
37      }
38      setSelectedItem(labels[0]);
39    }
40    /**
41     * Возвращает значение поля, выбранного пользователем
42     * @return Возвращает значение статического поля
43    */
44    public int getValue()
45    {
46      return table.get(getSelectedItem());
47    }
48  }
```

java.text.DateFormat 1.1

- `static Locale[] getAvailableLocales()`

  Возвращает массив объектов типа `Locale`, для которых доступны форматирующие объекты типа `DateFormat`.

- `static DateFormat getDateInstance(int dateStyle)`
- `static DateFormat getDateInstance(int dateStyle, Locale l)`
- `static DateFormat getTimeInstance(int timeStyle)`
- `static DateFormat getTimeInstance(int timeStyle, Locale l)`
- `static DateFormat getDateTimeInstance(int dateStyle, int timeStyle)`
- `static DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale l)`

  Возвращают объект, форматирующий дату, время или то и другое в соответствии с текущими или заданными региональными настройками.

Параметры:	*dateStyle*, *timeStyle*	Одно из значений констант DEFAULT, FULL, LONG, MEDIUM или SHORT

- `String format(Date d)`

  Возвращает символьную строку, получаемую в результате форматирования указанных даты и времени.

- `Date parse(String s)`

  Производит синтаксический анализ символьной строки и возвращает указанные в ней дату и время. Символьная строка не должна начинаться с пробелов. Любые символы, следующие в данной строке после даты и времени, игнорируются. При удачном исходе синтаксического анализа генерируется исключение типа `ParseException`.

- `void setLenient(boolean b)`
- `boolean isLenient()`

  Устанавливают или получают состояние признака, управляющего нестрогим режимом синтаксического анализа символьной строки, преобразуемой в дату. Если данный признак установлен, то неверно заданная дата, например **`February 30, 1999`**, будет автоматически преобразована в дату **`March 2, 1999`**. Нестрогий режим синтаксического анализа включается по умолчанию.

- `void setCalendar(Calendar cal)`
- `Calendar getCalendar()`

  Устанавливают или получают календарный объект, используемый для извлечения года, месяцев, дней, часов, минут и секунд из объекта типа `Date`. Эти методы служат для замены выбираемого по умолчанию календаря (обычно григорианского).

- `void setTimeZone(TimeZone tz)`
- `TimeZone getTimeZone()`

  Устанавливают или получают объект, определяющий часовой пояс и используемый для форматирования времени. Эти методы служат для замены часового пояса, выбираемого в соответствии с региональными настройками, устанавливаемыми по умолчанию на уровне операционной системы. Для других региональных настроек часовой пояс определяется на основании географического местоположения.

- `void setNumberFormat(NumberFormat f)`
- `NumberFormat getNumberFormat()`

  Устанавливают или получают формат чисел, представляющих годы, месяцы, дни, часы, минуты и секунды.

java.util.TimeZone 1.1

- `static String[] getAvailableIDs()`

  Получает идентификаторы всех поддерживаемых часовых поясов.

- `static TimeZone getDefault()`

  Получает объект типа `TimeZone`, представляющий часовой пояс (объект типа `TimeZone`), выбираемый по умолчанию на данном компьютере.

- `static TimeZone getTimeZone(String` *`timeZoneId`*`)`

  Получает идентификатор данного часового пояса.

- `String getDisplayName()`
- `String getDisplayName(Locale` *`locale`*`)`
- `String getDisplayName(boolean` *`daylight`*`, int` *`style`*`)`
- `String getDisplayName(boolean` *`daylight`*`, int` *`style`*`, Locale` *`locale`*`)`

  Получают имя часового пояса в соответствии с региональными настройками, устанавливаемыми по умолчанию или непосредственно. Если параметр *`daylight`* принимает логическое значение `true`, то возвращается имя часового пояса с учетом перехода на летнее время. А параметр *`style`* может принимать значение `SHORT` или `LONG`.

- `boolean useDaylightTime()`

  Возвращает логическое значение `true`, если в данном объекте типа `TimeZone` учитывается переход на летнее время.

- `boolean inDaylightTime(Date` *`date`*`)`

  Возвращает логическое значение `true`, если в указанной дате учитывается переход на летнее время в данном часовом поясе.

Сортировка

Большинству программирующих на Java известно, как сравнивать символьные строки с помощью метода `compareTo()` из класса `String`. Результат сравнения `a.compareTo(b)` будет отрицательным, если строка `a` лексикографически меньше строки `b`; нулевым, если символьные строки идентичны; или положительным, если строка `a` лексикографически больше строки `b`.

К сожалению, данный метод удобен лишь в тех случаях, когда слова состоят из прописных букв английского алфавита в коде ASCII. Дело в том, что в методе `compareTo()` для сравнения используются порядковые номера символов в уникоде. Например, строчные буквы в уникоде имеют бо́льшие значения, чем прописные, а буквы с ударением — еще бо́льшие значения. Это приводит нередко к абсурдным результатам, как, например, в приведенных ниже символьных строках, отсортированных с помощью метода `compareTo()`.

```
America
Zulu
able
zebra
Ångström
```

Для сортировки этих слов в лексикографическом порядке строчные и прописные буквы должны считаться равнозначными. Для англоязычного пользователя правильным будет следующий порядок:

```
able
America
Ångström
zebra
Zulu
```

Но для шведского пользователя такой порядок будет неправильным. В шведском языке буквы **A** и **Å** совершенно разные. Более того, буква **Å** следует *после* буквы **Z**! Поэтому для шведского пользователя эти слова должны быть отсортированы следующим образом:

```
able
America
zebra
Zulu
Ångström
```

Правда, разрешается это затруднение очень просто. Сначала получается объект типа `Locale`, а затем вызывается фабричный метод `getInstance()` для получения сортирующего объекта типа `Collator`. И наконец, для сортирующего объекта типа `Collator` вызывается метод `compare()`, а *не* метод `compareTo()` из класса `String`, как показано ниже.

```
Locale loc = . . .;
Collator coll = Collator.getInstance(loc);
if (coll.compare(a, b) < 0) // строка a предшествует строке b . . .;
```

Но самое главное, что сортирующий класс `Collator` реализует интерфейс `Comparator`. Поэтому для сортировки списка символьных строк можно передать объект типа `Collator` методу `Collections.sort()` следующим образом:

```
Collections.sort(strings, coll);
```

Избирательность сортировки

Для сортировки предусмотрены три уровня избирательности: *первостепенный*, *второстепенный*, *третьестепенный* и *идентичный*. Например, в английском языке отличие букв **A** и **Z** считается первостепенным, букв **A** и **Å** — второстепенным, а букв **A** и **a** — третьестепенным.

Для того чтобы внимание при сортировке обращалось только на первостепенные отличия, следует задать уровень ее избирательности `Collator.PRIMARY`. А если задать уровень избирательности `Collator.SECONDARY`, то будут учтены и второстепенные отличия. Таким образом, вероятность найти отличия в двух символьных строках будет больше при установке более высокого уровня избирательности, как показано в табл. 5.4.

Таблица 5.4. Сортировка с разными уровнями избирательности (английские региональные настройки)

Первостепенный уровень	Второстепенный уровень	Третьестепенный уровень
Angstrom = Ångström	Angstrom ≠ Ångström	Angstrom ≠ Ångström
Able = able	Able = able	Able ≠ able

Если установить уровень избирательности `Collator.IDENTICAL`, отличия не допускаются. Этот уровень избирательности используется главным образом вместе с *режимом разложения на составляющие*, который устанавливается для сортировки и рассматривается в следующем разделе.

Разложение на составляющие

Иногда символ или последовательность символов могут быть описаны не только в уникоде. Например, символу **Å** в уникоде соответствует код **U+00C5**. С другой стороны, его можно представить в виде последовательности символов **A** (код **U+0065**) и ° (кружок сверху; код U+030A). Еще удивительнее, что последовательность символов "ffi" может быть описана одним символом "латинская малая лигатура **ffi**" с кодом U+FB03. (Можно, конечно, спорить, что это вопрос представления символов, решение которого не должно приводить к появлению разных символов в уникоде, но правила установлены не нами.)

В стандарте на уникод определяются четыре *формы нормализации* символьных строк (D, KD, C и KC; подробнее об этом см. по адресу http://www.unicode.org/unicode/reports/tr15/tr15-23.html). Две из этих форм используются для сортировки. В форме нормализации D символы с ударением раскладываются на составляющие их буквы и ударения. Например, символ **Å** раскладывается на составляющие символы **A** и °. А в форме нормализации KD на составляющие раскладываются такие *символы совместимости*, как лигатура **ffi** или символ торговой марки ™.

Для сортировки можно выбрать определенную степень нормализации. Так, если установить значение константы Collator.NO_DECOMPOSITION, то символьные строки вообще не будут нормализованы при сортировке. В этом режиме сортировка выполняется быстрее, но он может быть непригодным для сортировки текста, где символы выражаются во многих формах. По умолчанию устанавливается значение константы Collator.CANONICAL_DECOMPOSITION, определяющее режим, в котором используется форма нормализации D. Это самая полезная форма для сортировки текста, содержащего символы с ударениями, но не лигатуры. И, наконец, в режиме полного разложения на составляющие используется форма нормализации KD. Характерные примеры сортировки в режимах разложения на составляющие приведены в табл. 5.5.

Таблица 5.5. Сортировка в разных режимах разложения на составляющие

Без разложения на составляющие	Каноническое разложение на составляющие	Полное разложение на составляющие
Å ≠ A	Å = A	Å = A
™ ≠ TM	™ ≠ TM	™ = TM

Если одна символьная строка сравнивается многократно с другими строками, то во избежание ее повторного разложения на составляющие и ради повышения эффективности результат разложения на составляющие следует сохранить в объекте *ключа сортировки*. Например, в приведенном ниже фрагменте кода метод getCollationKey() возвращает объект типа CollationKey, используемый для ускорения всех последующих операций сравнения.

```
String a = . . .;
CollationKey aKey = coll.getCollationKey(a);
if(aKey.compareTo(coll.getCollationKey(b)) == 0) // быстрое сравнение
. . .
```

И наконец, символьные строки иногда требуется преобразовать в их нормализованные формы, не прибегая к сортировке. Такая потребность возникает, например, при сохранении символьных строк в базе данных или при взаимодействии с другой программой. Для этой цели служит класс java.text.Normalizer, выполняющий процесс нормализации, как показано ниже. Нормализованная строка содержит

десять символов. Символы Å и ö заменяются последовательностями символов "A°" и "oë".

```
String name = "Ångström";
String normalized =
  Normalizer.normalize(name, Normalizer.Form.NFD); // использовать форму
                                                   // нормализации D
```

Тем не менее это обычно не самая лучшая форма для хранения и передачи символьных строк. В форме нормализации C сначала выполняется разложение на составляющие, а затем присоединяются ударения в установленном порядке. В соответствии с рекомендациями консорциума W3C такой режим является наиболее предпочтительным для передачи данных через Интернет.

Программа, исходный код которой приведен в листинге 5.4, позволяет экспериментировать с разными видами сортировки. Для этого достаточно ввести слово в текстовом поле и щелкнуть на кнопке Add, чтобы добавить введенное слово в список. Список сортируется заново после добавления в него каждого слова, изменения региональных настроек (в раскрывающемся списке Locale), уровня избирательности сортировки (в раскрывающемся списке Strength) или режима разложения на составляющие (в раскрывающемся списке Decomposition). Знак равенства (=) обозначает, что слова считаются одинаковыми (рис. 5.3).

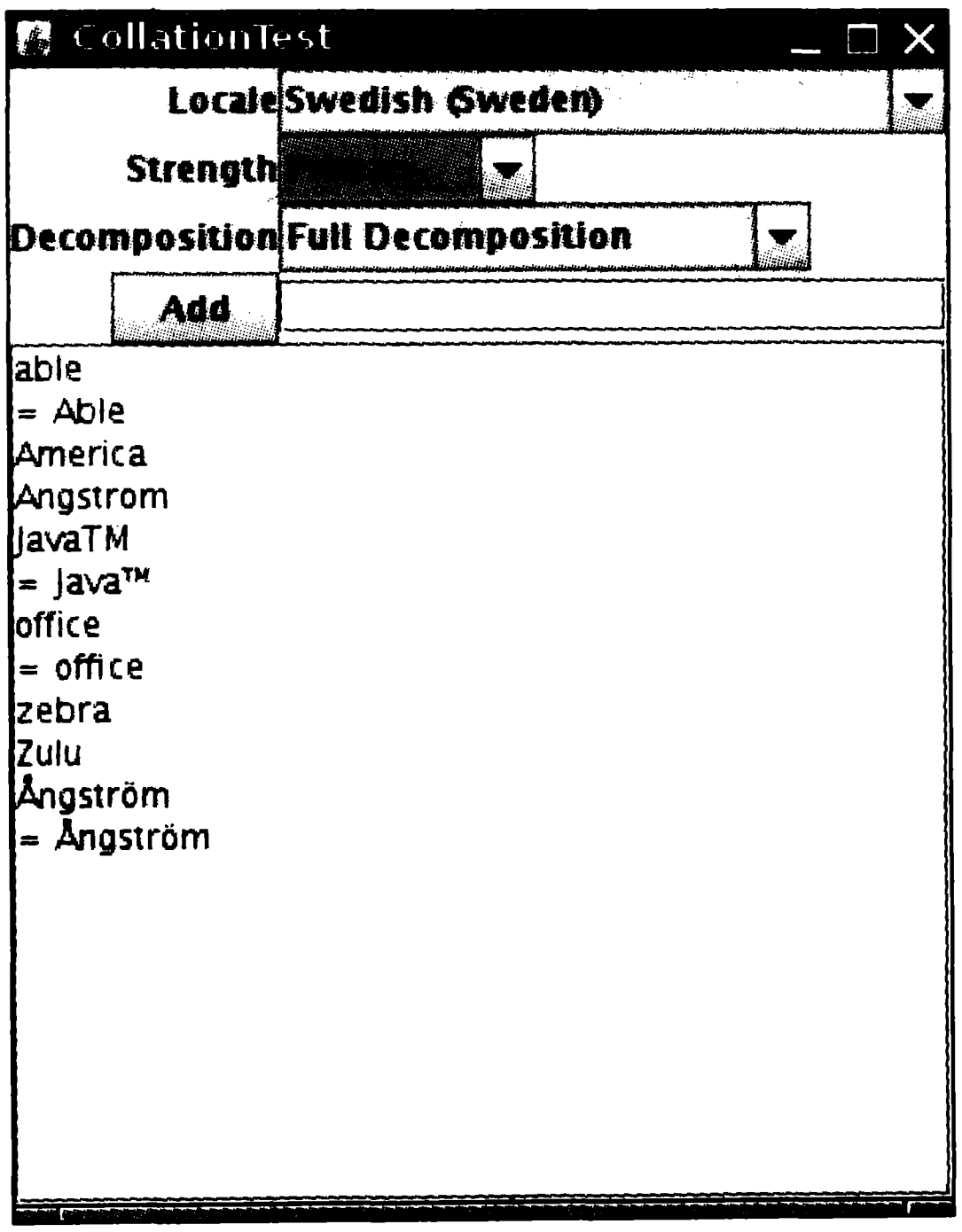


Рис. 5.3. Рабочее окно программы `CollationTest`

Имена региональных настроек в раскрывающемся списке Locale отображаются в порядке, отсортированном в соответствии с региональными настройками, устанавливаемыми по умолчанию. Так, если запустить рассматриваемую здесь программу при стандартных региональных настройках `US English`, то региональные настройки

Norwegian (Norway, Nynorsk) окажутся выше в данном списке, чем региональные настройки Norwegian (Norway), несмотря на то, что значение знака запятой в юникоде больше, чем значение знака закрывающей скобки.

Листинг 5.4. Исходный код из файла collation/CollationTest.java

```
1   package collation;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.text.*;
6   import java.util.*;
7   import java.util.List;
8
9   import javax.swing.*;
10
11  /**
12   * В этой программе демонстрируется сортировка символьных
13   * строк при выборе разных региональных настроек
14   * @version 1.14 2012-01-26
15   * @author Cay Horstmann
16  */
17  public class CollationTest
18  {
19    public static void main(String[] args)
20    {
21      EventQueue.invokeLater(new Runnable()
22        {
23          public void run()
24          {
25            JFrame frame = new CollationFrame();
26            frame.setTitle("CollationTest");
27            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28            frame.setVisible(true);
29          }
30        });
31    }
32  }
33
34  /**
35   * Этот фрейм содержит комбинированные списки для выбора
36   * региональных настроек, уровня избирательности сортировки и
37   * режимов разложения на составляющие, а также текстовое поле
38   * и кнопку для ввода новых символьных строк
39  */
40  class CollationFrame extends JFrame
41  {
42    private Collator collator =
43        Collator.getInstance(Locale.getDefault());
44    private List<String> strings = new ArrayList<>();
45    private Collator currentCollator;
46    private Locale[] locales;
47    private JComboBox<String> localeCombo = new JComboBox<>();
48    private JTextField newWord = new JTextField(20);
49    private JTextArea sortedWords = new JTextArea(20, 20);
50    private JButton addButton = new JButton("Add");
51    private EnumCombo strengthCombo = new EnumCombo(Collator.class,
52        "Primary", "Secondary", "Tertiary", "Identical");
```

```
53    private EnumCombo decompositionCombo = new EnumCombo(
54          Collator.class, "Canonical Decomposition",
55          "Full Decomposition", "No Decomposition");
56
57    public CollationFrame()
58    {
59      setLayout(new GridBagLayout());
60      add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
61      add(new JLabel("Strength"), new GBC(0, 1).setAnchor(GBC.EAST));
62      add(new JLabel("Decomposition"), new GBC(0, 2).setAnchor(GBC.EAST));
63      add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
64      add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
65      add(strengthCombo, new GBC(1, 1).setAnchor(GBC.WEST));
66      add(decompositionCombo, new GBC(1, 2).setAnchor(GBC.WEST));
67      add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
68      add(new JScrollPane(sortedWords),
69          new GBC(0, 4, 2, 1).setFill(GBC.BOTH));
70
71      locales = (Locale[]) Collator.getAvailableLocales().clone();
72      Arrays.sort(locales, new Comparator<Locale>()
73        {
74          public int compare(Locale l1, Locale l2)
75          {
76            return collator.compare(l1.getDisplayName(),
77                l2.getDisplayName());
78          }
79        });
80      for (Locale loc : locales)
81        localeCombo.addItem(loc.getDisplayName());
82      localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
83
84      strings.add("America");
85      strings.add("able");
86      strings.add("Zulu");
87      strings.add("zebra");
88      strings.add("\u00C5ngstr\u00F6m");
89      strings.add("A\u030angstro\u0308m");
90      strings.add("Angstrom");
91      strings.add("Able");
92      strings.add("office");
93      strings.add("o\uFB03ce");
94      strings.add("Java\u2122");
95      strings.add("JavaTM");
96      updateDisplay();
97
96      addButton.addActionListener(new ActionListener()
97        {
98          public void actionPerformed(ActionEvent event)
99          {
100           strings.add(newWord.getText());
101           updateDisplay();
102         }
103       });
104
105     ActionListener listener = new ActionListener()
106      {
107       public void actionPerformed(ActionEvent event)
108       {
109         updateDisplay();
```

```
110         }
111       };
112
113       localeCombo.addActionListener(listener);
114       strengthCombo.addActionListener(listener);
115       decompositionCombo.addActionListener(listener);
116       pack();
117    }
118
119    /**
120       * Обновляет отображаемые строки и сортирует их
121       * в соответствии с пользовательскими установками
122    */
123    public void updateDisplay()
124    {
125       Locale currentLocale = locales[localeCombo.getSelectedIndex()];
126       localeCombo.setLocale(currentLocale);
127
128       currentCollator = Collator.getInstance(currentLocale);
129       currentCollator.setStrength(strengthCombo.getValue());
130       currentCollator.setDecomposition(decompositionCombo.getValue());
131       Collections.sort(strings, currentCollator);
132
133       sortedWords.setText("");
134       for (int i = 0; i < strings.size(); i++)
134       {
135          String s = strings.get(i);
136          if (i > 0 && currentCollator.compare(s, strings.get(i - 1)) == 0)
137             sortedWords.append("= ");
138          sortedWords.append(s + "\n");
139       }
140       pack();
141    }
142 }
```

java.text.Collator 1.1

- `static Locale[] getAvailableLocales()`

  Возвращает массив объектов типа `Locale`, для которых существуют сортирующие объекты типа `Collator`.

- `static Collator getInstance()`
- `static Collator getInstance(Locale l)`

  Возвращают объект типа `Collator` для текущих или заданных региональных настроек.

- `int compare(String a, String b)`

  Возвращает отрицательное значение, если строка *a* предшествует строке *b*; нулевое значение, если строки считаются одинаковыми; или положительное значение, если строка *b* предшествует строке *a*.

- `boolean equals(String a, String b)`

  Возвращает логическое значение `true`, если строки *a* и *b* считаются одинаковыми, а иначе — логическое значение `false`.

- `void setStrength(int strength)`
- `int getStrength()`

  Устанавливают или получают уровень избирательности сортировки. Чем выше уровень избирательности, тем больше вероятность того, что сортируемые слова будут признаны разными. Поддерживаются следующие уровни избирательности сортировки: `Collator.PRIMARY`, `Collator.SECONDARY` и `Collator.TERTIARY`.

- `void setDecomposition(int decomp)`
- `int getDecompositon()`

  Устанавливают или получают режим разложения на составляющие при сортировке символьных строк. Чем выше степень разложения на составляющие, тем строже выполняется сравнение сортируемых символьных строк. Поддерживаются следующие режимы разложения на составляющие: `Collator.NO_DECOMPOSITION`, `Collator.CANONICAL_DECOMPOSITION` и `Collator.FULL_DECOMPOSITION`.

- `CollationKey getCollationKey(String a)`

  Возвращает ключ сортировки с разложенными на составляющие символами, чтобы быстро сравнить их по другому ключу сортировки.

java.text.CollationKey 1.1

- `int compareTo(CollationKey b)`

  Возвращает отрицательное значение, если данный ключ сортировки предшествует ключу *b*; нулевое значение, если ключи одинаковы; или положительное значение, если данный ключ следует за ключом *b*.

java.text.Normalizer 6

- `static String normalize(CharSequence str, Normalizer.Form form)`

  Возвращает нормализованную форму символьной строки *str*. Параметр *form* может принимать одно из следующих значений: `ND`, `NKD`, `NC` или `NKC`.

Форматирование сообщений

В состав библиотеки Java входит класс `MessageFormat` для форматирования текста, содержащего фрагменты с переменными, как показано ниже.

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

Здесь номера в фигурных скобках служат в качестве заполнителей для подлинных имен и значений. А подставить значения переменных можно с помощью статического метода `MessageFormat.format()`. С версии JDK 5.0 поддерживаются методы с переменным числом параметров, поэтому подстановка значений переменных может быть произведена следующим образом:

```
String msg = MessageFormat.format("On {2}, a {0} destroyed {1} houses
      and caused {3} of damage.", "hurricane", 99,
      new GregorianCalendar(1999, 0, 1).getTime(), 10.0E8);
```

В данном примере заполнитель `{0}` замещается значением `"hurricane"`, заполнитель `{1}` — значением **99** и т.д. А в результате подстановки получается следующая текстовая строка:

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses and caused 100,000,000 of
damage.
```

Результат для начала неплохой, но вряд ли может устроить полностью. В частности, время `12:00 AM` отображать не следует, а сумму ущерба от урагана нужно представить в денежных единицах. Это можно сделать, указав формат для некоторых переменных, как выделено ниже полужирным.

```
"On {2,date,long}, a {0} destroyed {1} houses and caused {3,number,currency}
   of damage."
```

В результате очередной подстановки получается следующая строка:

```
On January 1, 1999, a hurricane destroyed 99 houses and
   caused $100,000,000 of damage.
```

Обычно после заполнителя допускается задавать *тип* и *стиль*, разделяя их запятыми. Ниже перечислены допустимые типы.

```
number
time
date
choice
```

Если указан тип `number`, то допускаются следующие стили:

```
integer
currency
percent
```

В качестве стиля может быть также указан шаблон числового формата, например **`$,##0`**. (Подробнее об этом см. в документации на класс `DecimalFormat`.)

Для типа `time` или `date` может быть указан один из следующих стилей:

```
short
medium
long
full
```

Аналогично числам, в качестве стиля может быть указан шаблон даты. (Допустимые форматы подробно рассматриваются в документации на класс `SimpleDateFormat`.) Форматы выбора (типа `choice`) имеют более сложную структуру и подробнее рассматриваются далее в этой главе.

ВНИМАНИЕ! Статический метод `format()` форматирует значения с учетом текущих региональных настроек. Форматировать сообщения средствами класса `MessageFormat` с учетом произвольных региональных настроек немного сложнее, поскольку в этом классе отсутствует метод с переменным числом аргументов. Поэтому форматируемые значения придется разместить в массиве `Object[]`, как показано ниже.

```
MessageFormat mf = new MessageFormat(pattern, loc);
String msg = mf.format(new Object[] { значения });
```

java.text.MessageFormat 1.1

- `MessageFormat(String` *pattern*`)`
- `MessageFormat(String` *pattern*`, Locale` *loc*`)`

  Создают объект, форматирующий сообщения по заданному шаблону и указанным региональным настройкам.
- `void applyPattern(String` *pattern*`)`

  Задает шаблон для объекта, форматирующего сообщения.
- `void setLocale(Locale` *loc*`)`
- `Locale getLocale()`

  Устанавливают или получают региональные настройки для заполнителей в сообщении. Региональные настройки пригодны *только* для последующих шаблонов, задаваемых с помощью метода `applyPattern()`.
- `static String format(String pattern, Object...` *args*`)`

  Форматирует символьную строку по шаблону *pattern*, подставляя вместо заполнителей `{i}` объекты из массива `args[i]`.
- `StringBuffer format(Object` *args*`, StringBuffer` *result*`, FieldPosition` *pos*`)`

  Форматирует шаблон данного объекта типа `MessageFormat`. Параметр *args* принимает массив объектов. Форматируемая строка добавляется к значению параметра *result*, которое затем возвращается. Если параметр *pos* принимает ссылку на новый объект `new FieldPosition(MessageFormat.Field.ARGUMENT)`, его свойства `beginIndex` и `endIndex` устанавливаются в соответствии с расположением текста, который подставляется вместо заполнителя `{1}`. Если же сведения о расположении подстановочного текста не важны, в качестве параметра *pos* следует задать пустое значение `null`.

java.text.Format 1.1

- `String format(Object` *obj*`)`

  Форматирует заданный объект по правилам, определяемым текущим форматирующим объектом. С этой целью вызывается метод `format(obj, new StringBuffer(), new FieldPosition(1)).toString()`.

Форматы выбора

Вернемся к следующему шаблону из предыдущего раздела, чтобы рассмотреть его подробнее:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

Если вместо заполнителя `{0}`, обозначающего вид стихийного бедствия, подставить значение `earthquake` (землетрясение), то получится следующее предложение, нарушающее правила английской грамматики в отношении используемых артиклей:

```
On January 1, 1999, a earthquake destroyed . . .
```

Для устранения этой грамматической ошибки артикль **a** придется ввести в заполнитель `{0}` следующим образом:

```
"On {2}, {0} destroyed {1} houses and caused {3} of damage."
```

Теперь вместо заполнителя `{0}` будет подставлен текст `"a hurricane"` или `"an earthquake"`. Такой способ особенно удобен для перевода сообщений на языки, где в каждом роде употребляется отдельный артикль. Например, на немецком языке этот шаблон должен выглядеть так, как показано ниже.

```
"{0} zerstörte am {2} {1} Häuser und richtete einen Schaden von {3} an."
```

В этом случае заполнитель будет заменяться грамматически правильными сочетаниями артикля и имени существительного, например `"Ein Wirbelsturm"` и `"Eine Naturkatastrophe"`.

Теперь рассмотрим заполнитель `{1}`. Если стихийное бедствие оказалось не очень разрушительным, то вместо этого заполнителя можно подставить значение **1**. Но и в этом случае получится предложение с нарушением правил английской грамматики, как показано ниже.

```
On January 1, 1999, a mudslide destroyed 1 houses and . . .
```

Желательно, чтобы текст сообщения грамотно изменялся в соответствии с одним из следующих подставляемых значений:

```
no houses
one house
2 houses
. . .
```

Именно для этой цели и был внедрен формат выбора типа `choice`. В соответствии с этим форматом задается последовательность пар значений, каждая из которых содержит *нижний предел* и *форматирующую строку*. Нижний предел и форматирующая строка разделяются знаком **#**, а для разделения пар значений служит знак **|**. Ниже приведен пример заполнителя `{1}`, где формат выбора выделен полужирным. А результаты форматирования текста сообщения в зависимости от значения, подставляемого вместо заполнителя `{1}`, представлены в табл. 5.6.

```
{1,choice,0#no houses|1#one house|2#{1} houses}
```

Таблица 5.6. Текст сообщения, отформатированный по выбору

{1}	Результат
0	"no houses"
1	"one house"
3	"3 houses"
-1	"no houses"

А зачем в форматирующей строке дважды указывается заполнитель `{1}`? Когда к этому заполнителю применяется формат выбора и значение оказывается равным **2**, возвращается символьная строка `"{1} houses"`. Эта строка форматируется еще раз и включается в результирующую строку сообщения.

НА ЗАМЕТКУ! Приведенный выше пример показывает, что разработчики формата выбора приняли не самое лучшее решение. Так, если имеются три форматирующие строки, то для их разделения требуются два предела. Как правило, количество пределов должно быть на *единицу меньше*, чем количество форматирующих строк. И как следует из табл. 5.6, первый предел в классе `MessageFormat` вообще игнорируется. Синтаксис форматирования сообщений мог бы быть более понятным, если бы пределы указывались *между* выбираемыми вариантами, например, следующим образом:

```
no houses|1|one house|2|{1} houses // более понятный, но не действующий формат
```

С помощью знака **<** можно указать, что предлагаемый вариант должен быть выбран, если нижний предел оказывается строго меньше подставляемого значения. Вместо знака **#** можно также указывать знак ≤ (**\u2264** в уникоде). По желанию можно даже указать для первого значения нижний предел равным **-∞** (**-\u221E** в уникоде), как показано ниже.

```
-∞<no houses|0<one house|2≤{1} houses
```

А непосредственно в уникоде это же выражение будет выглядеть следующим образом:

```
-\u221E<no houses|0<one house|2\u2264{1} houses
```

В завершение примера форматирования текстового сообщения о последствиях стихийного бедствия разместим строку с условиями выбора в исходной строке сообщения. В результате получится следующий шаблон форматирования на английском языке:

```
String pattern = "On {2,date,long},
    {0} destroyed {1,choice,0#no houses|1#one house|2#{1} houses}" +
    "and caused {3,number,currency} of damage.";
```

А для форматирования на немецком языке этот шаблон будет выглядеть следующим образом:

```
String pattern = "{0} zerstörte am {2,date,long}
    {1,choice,0#kein Haus|1#ein Haus|2#{1} Häuser}"
    + "und richtete einen Schaden von {3,number,currency} an.";
```

Примечательно, что порядок слов в шаблонах форматирования на английском и немецком языках разный, но методу `format()` передается *тот же самый* массив объектов. Под требуемый порядок слов в форматирующей строке подстраивается только последовательность заполнителей.

Текстовые файлы и наборы символов

Как вам должно быть уже известно, кодирование символов в Java основывается на уникоде. Но во многих операционных системах применяются собственные варианты кодирования символов, часто несовместимые с другими, например по стандарту ISO 8859-1 (8-разрядная кодировка, иногда называемая также кодом ANSI) в США или BIG5 на Тайване.

Сохранять данные в текстовом файле нужно с учетом местной кодировки символов, чтобы пользователь мог открыть и нормально прочитать этот файл в другом приложении. Ниже показано, каким образом требующаяся кодировка символов

указывается в конструкторе класса `FileWriter`. А полный перечень кодировок приведен в табл. 1.1 главы 1.

```
out = new FileWriter(filename, "ISO-8859-1");
```

К сожалению, до сих пор не существует надлежащей связи между региональными настройками и кодировками символов. Например, при выборе пользователем региональных настроек `zh_TW`, принятых на Тайване, в Java не предусмотрен метод, который сообщил бы, что наиболее подходящей кодировкой в этом случае является BIG5.

Кодирование символов в исходных файлах

Не следует забывать, что при написании программы на Java приходится иметь дело с компилятором. И это приходится делать, неизбежно пользуясь инструментальными средствами в *локальной* операционной системе. Допустим, для создания исходных файлов программы на Java используется стандартный текстовый редактор Notepad в китайской версии Windows. Полученный в итоге исходный код не является переносимым (т.е. независимым от платформы) из-за того, что в нем используется локальная кодировка символов (GB или BIG5, в зависимости от региональных настроек операционной системы). Классы становятся переносимыми только после компиляции, и в этом случае идентификаторы и текстовые сообщения представляются с помощью модифицированной кодировки UTF-8. А это означает, что при компиляции и выполнении программы предполагается использование трех кодировок символов.

- Исходные файлы: локальная кодировка.
- Файлы классов: модифицированная кодировка UTF-8.
- Виртуальная машина: кодировка UTF-16.

(Модифицированная кодировка UTF-8 и кодировка UTF-16 описаны в главе 1.)

СОВЕТ. Конкретную кодировку исходного файла можно указать в командной строке с помощью параметра **`-encoding`**, как показано ниже.

```
javac -encoding Big5 Myfile.java
```

Для того чтобы сделать исходные файлы переносимыми, следует ограничиться только символами в коде ASCII. Иными словами, все символы, отсутствующие в коде ASCII, необходимо заменить их эквивалентными представлением в уникоде. Например, вместо символьной строки `"Häuser"` следует использовать символьную строку `"H\u0084user"`. Для преобразования текстов, набранных в локальной кодировке, в символы кода ASCII можно применить утилиту `native2ascii`, которая входит в состав JDK. Эта утилита заменяет каждый символ, отсутствующий в коде ASCII, кодовой последовательностью `\u`*XXXX*, где *XXXX* — четыре шестнадцатеричные цифры кода данного символа в уникоде. Применяя утилиту `native2ascii`, в командной строке нужно указать имя исходного и целевого файлов следующим образом:

```
native2ascii Myfile.java Myfile.temp
```

А для обратного преобразования служит параметр **`-reverse`**, как показано ниже.

```
native2ascii -reverse Myfile.temp Myfile.java
```

И наконец, для указания другой кодировки служит параметр **`-encoding`**, после которого в командной строке указывается название требующейся кодировки, как

показано в приведенном ниже примере. Допустимые названия кодировок приведены в табл. 1.1 главы 1.

```
native2ascii -encoding Big5 Myfile.java Myfile.temp
```

СОВЕТ. В именах классов рекомендуется ограничиться только символами в коде ASCII. Название файла класса происходит от имени соответствующего класса, поэтому особенности обработки в именах файлов тех символов, которые отсутствуют в коде ASCII, зависят от операционной системы. Например, в европейской версии Windows 95 для записи имен файлов используется кодировка Code Page 437 (первоначальная кодировка для ПК). Поэтому если создать класс с именем `Bär` и попытаться выполнить его в ОС Windows 95, то в конечном итоге будет получено сообщение об ошибке `"cannot find class Bär"` (не удается найти класс `BΣr`).

Комплекты ресурсов

При интернационализации приложений приходится переводить на другие языки огромное количество сообщений, надписей на кнопках и т.п. Для упрощения этой задачи рекомендуется собрать все переводимые символьные строки в отдельном месте, которое обычно называется *ресурсом*. В этом случае переводчику достаточно отредактировать файлы ресурсов, не затрагивая исходный код программы. В Java для определения строковых ресурсов используются файлы свойств, а для других разновидностей ресурсов создаются классы ресурсов.

НА ЗАМЕТКУ! Технология использования ресурсов в Java отличается от аналогичной технологии в операционных системах Windows и Mac OS. В программе, предназначенной для выполнения под Windows или Mac OS, такие ресурсы, как меню, диалоговые окна, пиктограммы и сообщения, хранятся отдельно от самой программы. Поэтому специальный редактор ресурсов позволяет просматривать и видоизменять эти ресурсы, не затрагивая программный код.

НА ЗАМЕТКУ! В главе 10 первого тома данной книги описывается принцип размещения ресурсов (файлов данных, звука и изображения) в архивном JAR-файле. Метод `getResource()` из класса `Class` находит нужный файл, открывает его и возвращает URL, указывающий на ресурс. При размещении файлов в архивном JAR-файле поиск файлов возлагается на загрузчик классов. Следует, однако, иметь в виду, что этот механизм не поддерживает региональные настройки.

Обнаружение комплектов ресурсов

Для интернационализации приложений создаются так называемые *комплекты ресурсов*. Каждый комплект представляет собой файл свойств или класс, который описывает элементы, специфические для конкретных региональных настроек, например сообщения, надписи и т.д. Для каждого комплекта ресурсов должны быть предоставлены все региональные настройки, поддержка которых предусматривается в прикладной программе.

Комплекты ресурсов именуются по специальным условным обозначениям. Например, ресурсы, специфические для Германии, размещаются в файле *имяКомплекта*`_de_DE`, а ресурсы, общие для стран, в которых используется немецкий язык, — в классе *имяКомплекта*`_de`. В целом комплекты ресурсов для отдельных стран именуются следующим образом:

имяКомплекта_язык_страна

А комплекты ресурсов для всех одноязычных стран именуются так:

имяКомплекта_язык

И наконец, в качестве резерва ресурсы, применяемые по умолчанию, размещаются в файле, имя которого не указывается без упомянутых выше суффиксов. А загружается комплект ресурсов следующим образом:

```
ResourceBundle currentResources =
    ResourceBundle.getBundle(имяКомплекта, текущиеРегиональныеНастройки);
```

Метод `getBundle()` пытается загрузить комплект ресурсов, совпадающий с текущими региональными настройками по языку и стране. Если попытка загрузки завершится неудачей, то поочередно опускается страна и язык. Затем аналогичный поиск ресурсов осуществляется в региональных настройках по умолчанию, и, наконец, происходит обращение к комплекту ресурсов, выбираемому по умолчанию. Если и эта попытка оказывается безуспешной, то генерируется исключение типа `MissingResourceException`.

Таким образом, метод `getBundle()` пытается загрузить комплекты ресурсов в приведенной ниже последовательности, где *ТРН* — текущие региональные настройки, а *РНУ* — региональные настройки по умолчанию.

имяКомплекта_языкТРН_страна ТРН
имяКомплекта_языкТРН
имяКомплекта_языкТРН_страна РНУ
имяКомплекта_языкРНУ
имяКомплекта

Даже после того, как метод `getBundle()` обнаружит комплект ресурсов, например *имяКомплекта*`_de_DE`, он, тем не менее, продолжает поиск ресурсов в комплектах `имяКомплекта_de` и *имяКомплекта*. Если эти комплекты ресурсов существуют, то они становятся *родительскими* по отношению к комплекту *имяКомплекта*`_de_DE` в *иерархии ресурсов*. Родительские комплекты требуются на тот случай, если нужный ресурс не удастся обнаружить в текущем комплекте. Иными словами, если нужный ресурс не найден в комплекте *имяКомплекта*`_de_DE`, его поиск продолжается в комплектах в *имяКомплекта*`_de` и *имяКомплекта*.

Очевидно, что это очень полезный механизм, но для его реализации пришлось бы немало программировать вручную. Поэтому механизм поддержки комплектов ресурсов в Java автоматически находит ресурсы, в наибольшей степени соответствующие конкретным региональным настройкам. Для того чтобы ввести в существующую программу новые региональные настройки, достаточно дополнить ими соответствующие комплекты ресурсов.

НА ЗАМЕТКУ! Обнаружение комплектов ресурсов рассматривается здесь в упрощенном виде. Если региональные настройки содержат письмо или вариант языка, поиск ресурсов значительно усложняется. Подробнее об этом можно узнать из документации на метод `ResourceBundle.Control.getCandidateLocales()`.

СОВЕТ. При разработке прикладной программы совсем не обязательно размещать все ресурсы в одном комплекте. Вместо этого один комплект ресурсов можно создать для надписей на кнопках, другой — для сообщений об ошибках и т.д.

Файлы свойств

Интернационализация символьных строк осуществляется довольно просто. Для этого достаточно разместить все символьные строки в файле свойств, например `MyProgramStrings.properties`. Файл свойств представляет собой обычный текстовый файл, каждая строка которого содержит ключ и значение, как показано ниже.

```
computeButton=Rechnen
colorName=black
defaultPaperSize=210x297
```

Файлы свойств именуются по принципу, описанному в предыдущем разделе, например, следующим образом:

```
MyProgramStrings.properties
MyProgramStrings_en.properties
MyProgramStrings_de_DE.properties
```

Комплект ресурсов можно загрузить аналогично приведенному ниже.

```
ResourceBundle bundle = ResourceBundle.getBundle("MyProgramStrings", locale);
```

А для поиска конкретной символьной строки потребуется вызов, подобный следующему:

```
String computeButtonLabel = bundle.getString("computeButton");
```

ВНИМАНИЕ! Файлы свойств могут содержать только символы в коде ASCII. Для размещения в них символов в уникоде следует использовать формат `\uxxxx`. Например, строка `colorName=Grün` будет иметь следующий вид:

```
colorName=Gr\u00FCn
```

Для подобного преобразования символов в файлах свойств можно также воспользоваться утилитой `native2ascii`.

Классы комплектов ресурсов

Для поддержки ресурсов, не являющихся символьными строками, следует определить классы, производные от класса `ResourceBundle`. Выбор имен для таких классов осуществляется в соответствии со стандартными обозначениями, как показано в приведенном ниже примере.

```
MyProgramResources.java
MyProgramResources_en.java
MyProgramResources_de_DE.java
```

Для загрузки класса комплекта ресурсов используется тот же самый метод `getBundle()`, что и для загрузки файла свойств, как показано ниже.

```
RésourceBundle bundle = ResourceBundle.getBundle("MyProgramResources", locale);
```

ВНИМАНИЕ! Если два комплекта ресурсов, один из которых реализован в виде класса, а другой — в виде файла свойств, имеют одинаковые имена, то при загрузке предпочтение отдается классу.

В каждом классе комплекта ресурсов реализуется таблица поиска. Для получения каждого интернационализируемого значения следует предоставить символьную строку с соответствующим ключом, как показано ниже.

```
Color backgroundColor = (Color) bundle.getObject("backgroundColor");
double[] paperSize = (double[]) bundle.getObject("defaultPaperSize");
```

Самый простой способ реализовать класс комплекта ресурсов — создать подкласс, производный от класса `ListResourceBundle`. Класс `ListResourceBundle` позволяет разместить все ресурсы в массиве объектов и затем выполняет их поиск. Общая форма реализации подкласса, производного от класса `ListResourceBundle`, выглядит следующим образом:

```
public class bundleName_language_country extends ListResourceBundle
{
  private static final Object[][] contents =
  {
    { key1, value2 },
    { key2, value2 },
    . . .
  }
  public Object[][] getContents() { return contents; }
}
```

Ниже приведены некоторые примеры реализации подклассов, производных от класса `ListResourceBundle`.

```
public class ProgramResources_de extends ListResourceBundle
{
  private static final Object[][] contents =
  {
    { "backgroundColor", Color.black },
    { "defaultPaperSize", new double[] { 210, 297 } }
  }
  public Object[][] getContents() { return contents; }
}

public class ProgramResources_en_US extends ListResourceBundle
{
  private static final Object[][] contents =
  {
    { "backgroundColor", Color.blue },
    { "defaultPaperSize", new double[] { 216, 279 } }
  }
public Object[][] getContents() { return contents; }
}
```

НА ЗАМЕТКУ! Размеры бумаги заданы в миллиметрах. Во всех странах мира, кроме США и Канады, используется бумага, размеры которой определяются по стандарту ISO 216 (дополнительные сведения по данному вопросу можно найти в документе по адресу `http://www.cl.cam.ac.uk/~mgk25/iso-paper.html`). По данным Метрической ассоциации США (U.S. Metric Association; `http://lamar.colostate.edu/~hillger`) лишь три страны в мире — Либерия, Бирма и США — официально не приняли метрическую систему.

Класс комплекта ресурсов можно также создать как подкласс, производный от класса `ResourceBundle`. В этом случае придется реализовать два приведенных ниже метода для получения объекта типа `Enumeration`, содержащего

ключи, а также для извлечения конкретного значения по заданному ключу. Метод `getObject()` из класса `ResourceBundle` вызывает определяемый пользователем метод `handleGetObject()`.

```
Enumeration<String> getKeys()
Object handleGetObject(String key)
```

java.util.ResourceBundle 1.1

- `static ResourceBundle getBundle(String baseName, Locale loc)`
- `static ResourceBundle getBundle(String baseName)`
- Загружают класс комплекта ресурсов по указанному имени, а также его родительские классы в соответствии с заданными или устанавливаемыми по умолчанию региональными настройками. Если классы комплектов ресурсов находятся в пакете, то должно быть указано полное имя такого класса, например `intl.Programresources`. Классы комплектов ресурсов должны быть объявлены открытыми (`public`), чтобы метод `getBundle()` мог обращаться к ним.
- `Object getObject(String name)`
- Извлекает объект из комплекта ресурсов или его родительских комплектов.
- `String getString(String name)`
- Извлекает объект из комплекта ресурсов или его родительских комплектов и приводит его к типу `String`.
- `String[] getStringArray(String name)`
- Извлекает объект из комплекта ресурсов или его родительских комплектов и представляет его в виде массива символьных строк.
- `Enumeration<String> getKeys()`
- Возвращает объект перечисления типа `Enumeration`, содержащий ключи из текущего комплекта ресурсов. При этом объекте перечисляются также ключи из родительских комплектов ресурсов.
- `Object handleGetObject(String key)`
- Если реализуется собственный механизм поиска ресурсов, этот метод следует переопределить таким образом, чтобы он возвращал значение по заданному ключу.

Пример интернационализации прикладной программы

В этом разделе изложенный выше материал применяется на практике для интернационализации прикладной программы калькуляции пенсионных сбережений. В этой программе определяется, достаточно ли отчисляется денежных средств для выхода на пенсию. Для этого следует ввести свой возраст, сумму, которая отчисляется каждый месяц, а также указать другие данные (рис. 5.4)

В текстовой области и на графике представлены данные о ежегодном состоянии пенсионного счета. Если с возрастом остатки на пенсионном счету становятся отрицательными, что наглядно показано на части графика, построенной ниже оси абсцисс, в таком случае следует принять какие-то меры, например строже экономить деньги, отложить дату выхода на пенсию и т.п.

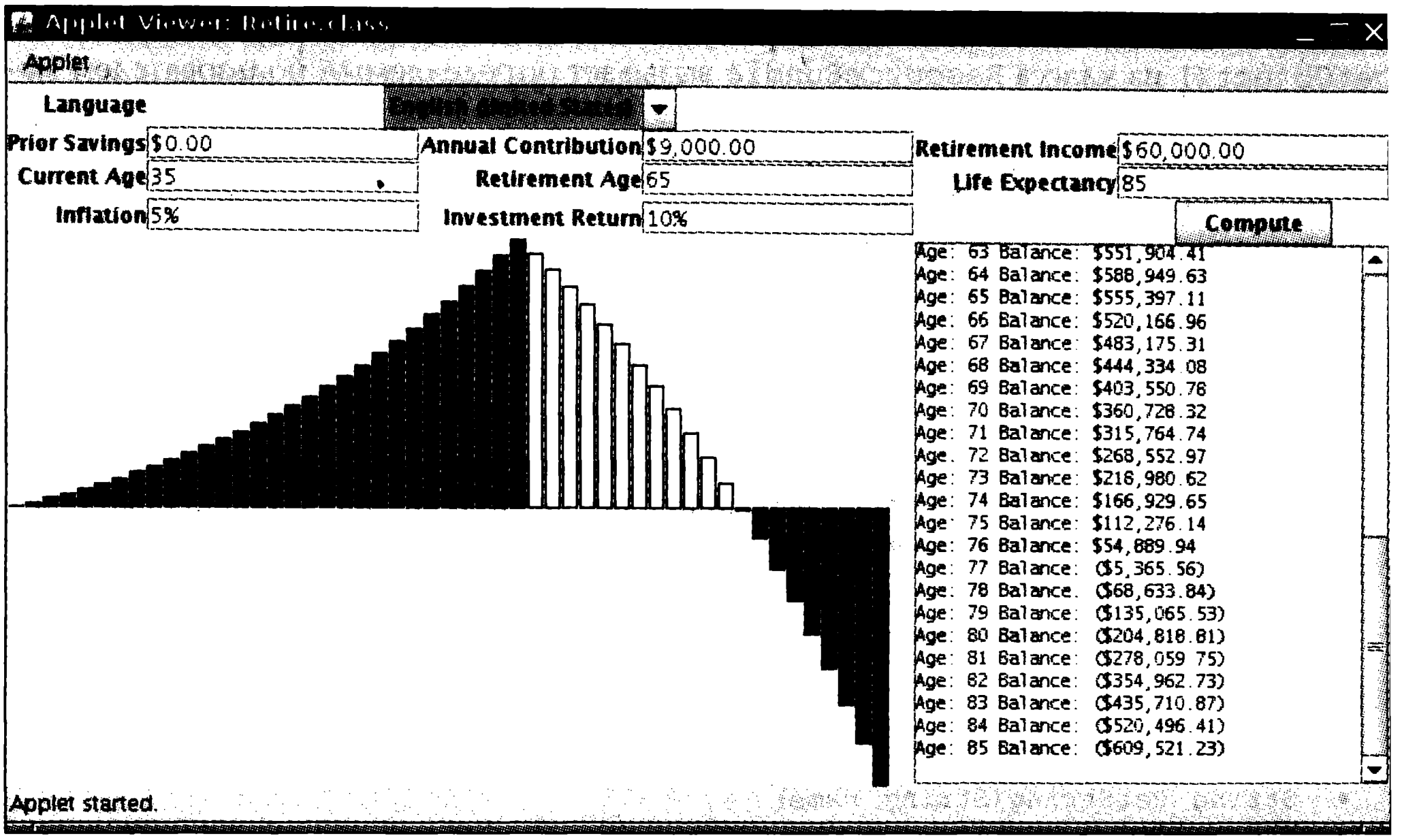


Рис. 5.4. Рабочее окно программы калькуляции пенсионных сбережений с ГПИ на английском языке

Программа калькуляции пенсионных сбережений имеет три варианта пользовательского интерфейса на английском, немецком и китайском языках. Ниже перечислены основные особенности интернационализации данной прикладной программы.

- Надписи и сообщения переводятся с английского на немецкий и китайский языки. Соответствующие ресурсы находятся в классах `RetireResources_de` и `Retire Resources_zh`, а ресурсы для пользовательского интерфейса на английском языке используются в качестве резервных и находятся в классе `RetireResources`. Для составления сообщений на китайском языке был использован текстовый редактор Notepad в китайской версии Windows, а затем с помощью утилиты `native2ascii` текст был преобразован в уникод.
- При изменении региональных настроек заново форматируются надписи и содержимое текстовых полей.
- Содержимое текстовых полей для ввода чисел, денежных сумм и процентов представляется в формате, соответствующем выбранным региональным настройкам.
- Для форматирования расчетного поля используется класс `MessageFomat`. Форматирующая строка хранится в комплекте ресурсов для каждого языка.
- Цвет графика меняется в зависимости от выбранного языка. Это никак не влияет на выполнение программой ее функций, а лишь демонстрирует подобную возможность.

В листингах 5.5–5.8 приведен исходный код рассматриваемой здесь прикладной программы, а в листингах 5.9–5.11 — содержимое файлов свойств для интернационализированных строк. На рис. 5.5 и 5.6 показаны немецкий и китайский варианты

пользовательского интерфейса данной программы. Для работы с данной программой на китайском языке необходимо запустить ее в китайской версии Windows или установить самостоятельно китайские шрифты в операционной системе.

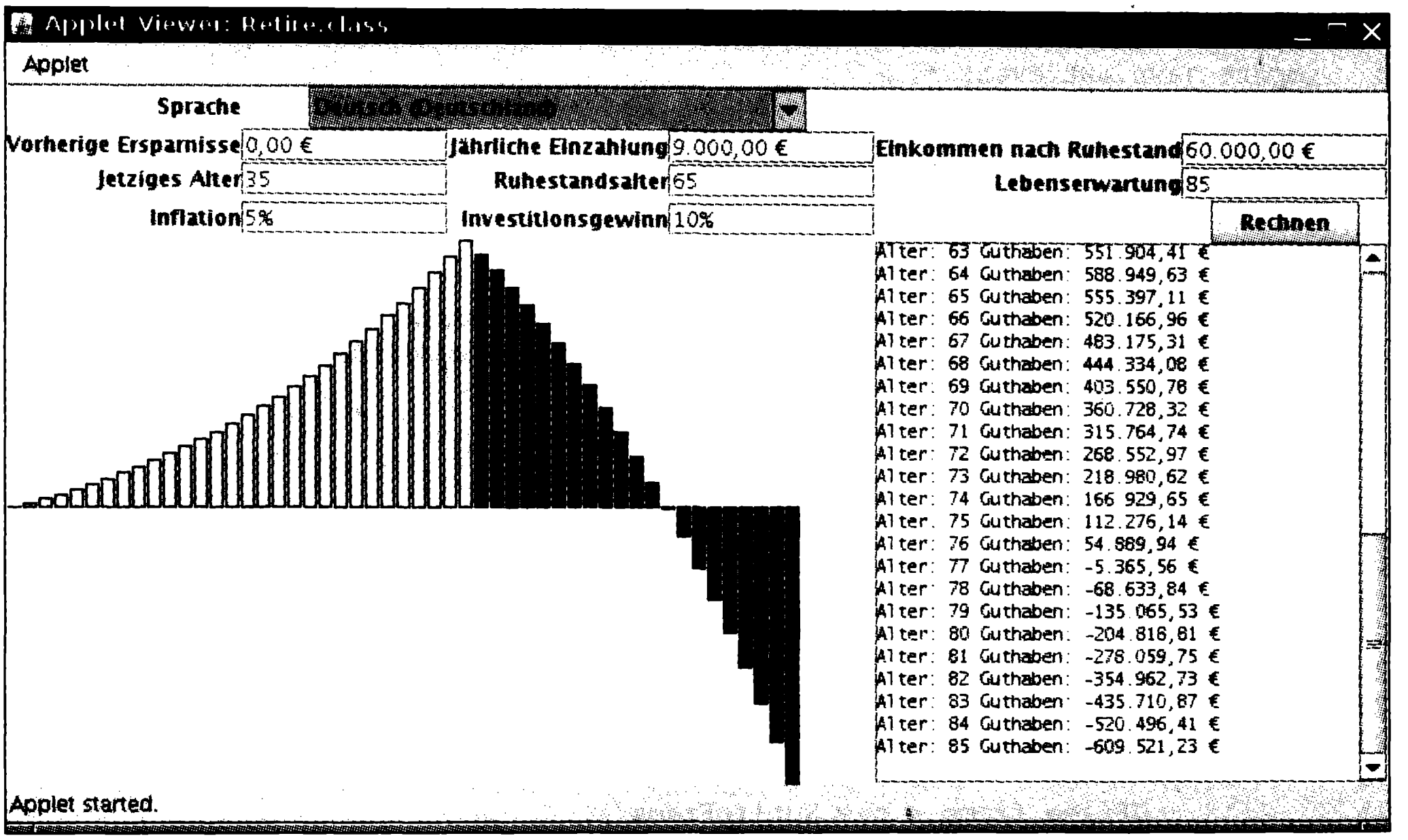


Рис. 5.5. Рабочее окно программы калькуляции пенсионных сбережений с ГПИ на немецком языке

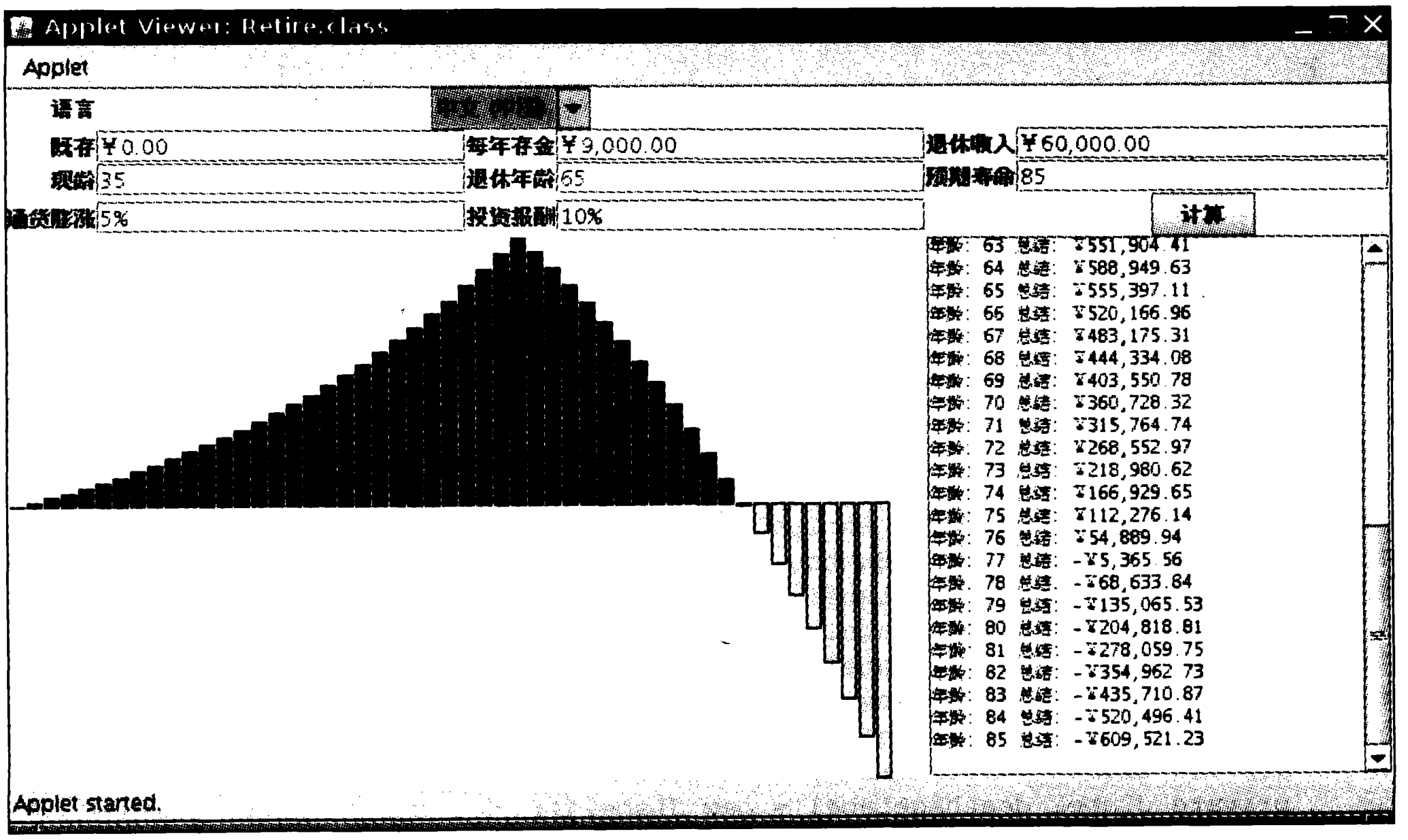


Рис. 5.6. Рабочее окно программы калькуляции пенсионных сбережений с ГПИ на китайском языке

Листинг 5.5. Исходный код из файла `retire/Retire.java`

```
1   package retire;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.awt.geom.*;
6   import java.util.*;
7   import java.text.*;
8   import javax.swing.*;
9
10  /**
11   * В этой программе демонстрируется калькуляция пенсионных сбережений.
12   * Ее пользовательский интерфейс представлен на английском, немецком
13   * и китайском языках
14   * @version 1.23 2012-06-07
15   * @author Cay Horstmann
16  */
17  public class Retire
18  {
19    public static void main(String[] args)
20    {
21      EventQueue.invokeLater(new Runnable()
22        {
23          public void run()
24          {
25            JFrame frame = new RetireFrame();
26            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27            frame.setVisible(true);
28          }
29        });
30    }
31  }
32
33  class RetireFrame extends JFrame
34  {
35    private JTextField savingsField = new JTextField(10);
36    private JTextField contribField = new JTextField(10);
37    private JTextField incomeField = new JTextField(10);
38    private JTextField currentAgeField = new JTextField(4);
39    private JTextField retireAgeField = new JTextField(4);
40    private JTextField deathAgeField = new JTextField(4);
41    private JTextField inflationPercentField = new JTextField(6);
42    private JTextField investPercentField = new JTextField(6);
43    private JTextArea retireText = new JTextArea(10, 25);
44    private RetireComponent retireCanvas = new RetireComponent();
45    private JButton computeButton = new JButton();
46    private JLabel languageLabel = new JLabel();
47    private JLabel savingsLabel = new JLabel();
48    private JLabel contribLabel = new JLabel();
49    private JLabel incomeLabel = new JLabel();
50    private JLabel currentAgeLabel = new JLabel();
51    private JLabel retireAgeLabel = new JLabel();
52    private JLabel deathAgeLabel = new JLabel();
53    private JLabel inflationPercentLabel = new JLabel();
54    private JLabel investPercentLabel = new JLabel();
55    private RetireInfo info = new RetireInfo();
```

```
56    private Locale[] locales = { Locale.US, Locale.CHINA, Locale.GERMANY };
57    private Locale currentLocale;
58    private JComboBox<Locale> localeCombo = new LocaleCombo(locales);
59    private ResourceBundle res;
60    private ResourceBundle resStrings;
61    private NumberFormat currencyFmt;
62    private NumberFormat numberFmt;
63    private NumberFormat percentFmt;
64
65    public RetireFrame()
66    {
67      setLayout(new GridBagLayout());
68      add(languageLabel, new GBC(0, 0).setAnchor(GBC.EAST));
69      add(savingsLabel, new GBC(0, 1).setAnchor(GBC.EAST));
70      add(contribLabel, new GBC(2, 1).setAnchor(GBC.EAST));
71      add(incomeLabel, new GBC(4, 1).setAnchor(GBC.EAST));
72      add(currentAgeLabel, new GBC(0, 2).setAnchor(GBC.EAST));
73      add(retireAgeLabel, new GBC(2, 2).setAnchor(GBC.EAST));
74      add(deathAgeLabel, new GBC(4, 2).setAnchor(GBC.EAST));
75      add(inflationPercentLabel, new GBC(0, 3).setAnchor(GBC.EAST));
76      add(investPercentLabel, new GBC(2, 3).setAnchor(GBC.EAST));
77      add(localeCombo, new GBC(1, 0, 3, 1));
78      add(savingsField, new GBC(1, 1).setWeight(100, 0).
79          setFill(GBC.HORIZONTAL));
80      add(contribField, new GBC(3, 1).setWeight(100, 0).
81          setFill(GBC.HORIZONTAL));
82      add(incomeField, new GBC(5, 1).setWeight(100, 0).
83          setFill(GBC.HORIZONTAL));
84      add(currentAgeField, new GBC(1, 2).setWeight(100, 0).
85          setFill(GBC.HORIZONTAL));
86      add(retireAgeField, new GBC(3, 2).setWeight(100, 0).
87          setFill(GBC.HORIZONTAL));
88      add(deathAgeField, new GBC(5, 2).setWeight(100, 0).
89          setFill(GBC.HORIZONTAL));
90      add(inflationPercentField, new GBC(1, 3).setWeight(100, 0).
91          setFill(GBC.HORIZONTAL));
92      add(investPercentField, new GBC(3, 3).setWeight(100, 0).
93          setFill(GBC.HORIZONTAL));
94      add(retireCanvas, new GBC(0, 4, 4, 1).setWeight(100, 100).
95          setFill(GBC.BOTH));
96      add(new JScrollPane(retireText), new GBC(4, 4, 2, 1).
97          setWeight(0, 100).setFill(GBC.BOTH));
98
99      computeButton.setName("computeButton");
100     computeButton.addActionListener(new ActionListener()
101       {
102         public void actionPerformed(ActionEvent event)
103         {
104           getInfo();
105           updateData();
106           updateGraph();
107         }
108       });
109     add(computeButton, new GBC(5, 3));
110
111     retireText.setEditable(false);
112     retireText.setFont(new Font("Monospaced", Font.PLAIN, 10));
```

```
113
114     info.setSavings(0);
115     info.setContrib(9000);
116     info.setIncome(60000);
117     info.setCurrentAge(35);
118     info.setRetireAge(65);
119     info.setDeathAge(85);
120     info.setInvestPercent(0.1);
121     info.setInflationPercent(0.05);
122
123     int localeIndex = 0; // региональные настройки США
124                          // выбираются по умолчанию
125     for (int i = 0; i < locales.length; i++)
126       // если текущие региональные настройки относятся к
127       // числу выбираемых, выбрать их
128       if (getLocale().equals(locales[i])) localeIndex = i;
129     setCurrentLocale(locales[localeIndex]);
130
131     localeCombo.addActionListener(new ActionListener()
132       {
133         public void actionPerformed(ActionEvent event)
134         {
135           setCurrentLocale((Locale) localeCombo.getSelectedItem());
136           validate();
137         }
138       });
139     pack();
140   }
141
142   /**
143    * Устанавливает текущие региональные настройки
144    * @param locale Требующиеся региональные настройки
145   */
146   public void setCurrentLocale(Locale locale)
147   {
148     currentLocale = locale;
149     localeCombo.setSelectedItem(currentLocale);
150     localeCombo.setLocale(currentLocale);
151
152     res = ResourceBundle.getBundle(
153           "retire.RetireResources", currentLocale);
154     resStrings = ResourceBundle.getBundle(
155           "retire.RetireStrings", currentLocale);
156     currencyFmt = NumberFormat.getCurrencyInstance(currentLocale);
157     numberFmt = NumberFormat.getNumberInstance(currentLocale);
158     percentFmt = NumberFormat.getPercentInstance(currentLocale);
159
160     updateDisplay();
161     updateInfo();
162     updateData();
162     updateGraph();
164   }
165
166   /**
167    * Обновляет все метки при отображении
168   */
169   public void updateDisplay()
```

```
170   {
171     languageLabel.setText(resStrings.getString("language"));
172     savingsLabel.setText(resStrings.getString("savings"));
173     contribLabel.setText(resStrings.getString("contrib"));
174     incomeLabel.setText(resStrings.getString("income"));
175     currentAgeLabel.setText(resStrings.getString("currentAge"));
176     retireAgeLabel.setText(resStrings.getString("retireAge"));
177     deathAgeLabel.setText(resStrings.getString("deathAge"));
178     inflationPercentLabel.setText(
179         resStrings.getString("inflationPercent"));
180     investPercentLabel.setText(resStrings.getString("investPercent"));
181     computeButton.setText(resStrings.getString("computeButton"));
182   }
183
184   /**
185    * Обновляет данные в текстовых полях
184   */
185   public void updateInfo()
186   {
187     savingsField.setText(currencyFmt.format(info.getSavings()));
188     contribField.setText(currencyFmt.format(info.getContrib()));
189     incomeField.setText(currencyFmt.format(info.getIncome()));
190     currentAgeField.setText(numberFmt.format(info.getCurrentAge()));
191     retireAgeField.setText(numberFmt.format(info.getRetireAge()));
192     deathAgeField.setText(numberFmt.format(info.getDeathAge()));
193     investPercentField.setText(
194         percentFmt.format(info.getInvestPercent()));
195     inflationPercentField.setText(
196         percentFmt.format(info.getInflationPercent()));
197   }
198
199   /**
200    * Обновляет данные, отображаемые в текстовой области
201   */
202   public void updateData()
203   {
204     retireText.setText("");
205     MessageFormat retireMsg = new MessageFormat("");
206     retireMsg.setLocale(currentLocale);
207     retireMsg.applyPattern(resStrings.getString("retire"));
208
209     for (int i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
210     {
211       Object[] args = { i, info.getBalance(i) };
212       retireText.append(retireMsg.format(args) + "\n");
213     }
214   }
215
216   /**
217    * Обновляет график
218   */
219   public void updateGraph()
220   {
221     retireCanvas.setColorPre((Color) res.getObject("colorPre"));
222     retireCanvas.setColorGain((Color) res.getObject("colorGain"));
223     retireCanvas.setColorLoss((Color) res.getObject("colorLoss"));
224     retireCanvas.setInfo(info);
```

```
225       repaint();
226    }
227
228    /**
229       * Считывает данные, вводимые пользователем в текстовых полях
230    */
231    public void getInfo()
232    {
233       try
234       {
235          info.setSavings(currencyFmt.parse(
236                savingsField.getText()).doubleValue());
237          info.setContrib(currencyFmt.parse(
238                contribField.getText()).doubleValue());
239          info.setIncome(currencyFmt.parse(
240                incomeField.getText()).doubleValue());
241          info.setCurrentAge(numberFmt.parse(
242                currentAgeField.getText()).intValue());
243          info.setRetireAge(numberFmt.parse(
244                retireAgeField.getText()).intValue());
245          info.setDeathAge(numberFmt.parse(
246                deathAgeField.getText()).intValue());
247           info.setInvestPercent(percentFmt.parse(
248                investPercentField.getText()).doubleValue());
249           info.setInflationPercent(percentFmt.parse(
250                inflationPercentField.getText()).doubleValue());
251        }
252        catch (ParseException ex)
253       {
254          ex.printStackTrace();
255       }
256    }
257 }
258
259 /**
260    * Данные, требующиеся для расчета пенсионных отчислений
262 */
263 class RetireInfo
264 {
265    private double savings;
266    private double contrib;
267    private double income;
268    private int currentAge;
269    private int retireAge;
270    private int deathAge;
271    private double inflationPercent;
272    private double investPercent;
273    private int age;
274    private double balance;
275
276    /**
277       * Получает остаток на счету, имеющийся на указанный год
278       * @param year Год для расчета остатка на счету
277       * @return Возвращает сумму, имеющуюся (или требуемую)
278       *         на указанный год
279    */
280    public double getBalance(int year)
```

```
281   {
282     if (year < currentAge) return 0;
283     else if (year == currentAge)
284     {
285       age = year;
286       balance = savings;
287       return balance;
288     }
289     else if (year == age) return balance;
290     if (year != age + 1) getBalance(year - 1);
291     age = year;
292     if (age < retireAge) balance += contrib;
293     else balance -= income;
294     balance = balance * (1 + (investPercent - inflationPercent));
295     return balance;
296   }
297
298   /**
299    * Получает сумму предыдущих сбережений
300    * @return Возвращает сумму сбережений
301   */
302   public double getSavings()
304   {
305     return savings;
306   }
307
308   /**
309    * Устанавливает сумму предыдущих сбережений
310    * @param newValue Сумма сбережений
311   */
312   public void setSavings(double newValue)
313   {
314     savings = newValue;
315   }
316
317   /**
318    * Получает сумму ежегодных отчислений на пенсионный счет
319    * @return Возвращает сумму ежегодных отчислений
320   */
321   public double getContrib()
323   {
324     return contrib;
325   }
326
327   /**
328    * Устанавливает сумму ежегодных отчислений на пенсионный счет
329    * @param newValue Сумма отчислений
330   */
331   public void setContrib(double newValue)
332   {
333     contrib = newValue;
334   }
335
336   /**
337    * Получает сумму ежегодного дохода
338    * @return Возвращает сумму ежегодного дохода
339   */
```

```
340   public double getIncome()
341   {
342     return income;
343   }
344
345   /**
346    * Устанавливает сумму ежегодного дохода
347    * @param newValue Сумма ежегодного дохода
348   */
349   public void setIncome(double newValue)
350   {
351     income = newValue;
353   }
354
355   /**
356    * Получает текущий возраст
357    * @return Возвращает текущий возраст
358   */
359   public int getCurrentAge()
360   {
361     return currentAge;
362   }
363
364   /**
365    * Устанавливает текущий возраст
366    * @param newValue Текущий возраст
367   */
368   public void setCurrentAge(int newValue)
369   {
370     currentAge = newValue;
371   }
372
373   /**
374    * Получает возраст, требующийся для выхода на пенсию
375    * @return Возвращает пенсионный возраст
376   */
377   public int getRetireAge()
378   {
379     return retireAge;
380   }
381
382   /**
383    * Устанавливает возраст, требующийся для выхода на пенсию
384    * @param newValue Возраст
385   */
386   public void setRetireAge(int newValue)
387   {
388     retireAge = newValue;
389   }
390
391   /**
392    * Получает предполагаемую продолжительность жизни
393    * @return Возвращает предполагаемую продолжительность жизни
394   */
395   public int getDeathAge()
396   {
397     return deathAge;
```

```
398    }
399
400    /**
401     * Устанавливает предполагаемую продолжительность жизни
402     * @param newValue Предполагаемая продолжительность жизни
403    */
404    public void setDeathAge(int newValue)
405    {
406      deathAge = newValue;
407    }
408
409    /**
410     * Получает предполагаемый уровень инфляции в процентах
411     * @return Возвращает уровень инфляции в процентах
412    */
413    public double getInflationPercent()
414    {
415      return inflationPercent;
416    }
417
418    /**
419     * Устанавливает предполагаемый уровень инфляции в процентах
420     * @param newValue Уровень инфляции в процентах
421    */
422    public void setInflationPercent(double newValue)
423    {
424      inflationPercent = newValue;
425    }
426
427    /**
428     * Получает предполагаемый доход от капиталовложений
429     * @return Возвращает доход от капиталовложений
430    */
431    public double getInvestPercent()
432    {
433      return investPercent;
434    }
435
436    /**
437     * Устанавливает предполагаемый доход от капиталовложений
438     * @param newValue Доход от капиталовложений в процентах
439    */
440    public void setInvestPercent(double newValue)
441    {
442      investPercent = newValue;
443    }
444 }
445
446 /**
447    * Этот компонент рисует график результатов пенсионных вложений
448    */
449 class RetireComponent extends JComponent
450 {
451    private static final int DEFAULT_WIDTH = 800;
452    private static final int DEFAULT_HEIGHT = 600;
453    private static final int PANEL_WIDTH = 400;
454    private static final int PANEL_HEIGHT = 200;
```

```
455
456   private RetireInfo info = null;
457   private Color colorPre;
458   private Color colorGain;
459   private Color colorLoss;
460
461   public RetireComponent()
462   {
463     setSize(PANEL_WIDTH, PANEL_HEIGHT);
464   }
465
466   /**
467    * Устанавливает данные для построения графика пенсионных вложений
468    * @param newInfo Новые данные о пенсионных вложениях
469   */
470   public void setInfo(RetireInfo newInfo)
471   {
472     info = newInfo;
473     repaint();
474   }
475
476   public void paintComponent(Graphics g)
477   {
478     Graphics2D g2 = (Graphics2D) g;
479     if (info == null) return;
480
481     double minValue = 0;
482     double maxValue = 0;
483     int i;
484     for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
485     {
486       double v = info.getBalance(i);
487       if (minValue > v) minValue = v;
488       if (maxValue < v) maxValue = v;
489     }
490     if (maxValue == minValue) return;
491
492     int barWidth = getWidth() / (info.getDeathAge() -
493       info.getCurrentAge() + 1);
494     double scale = getHeight() / (maxValue - minValue);
495
496     for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
497     {
498       int x1 = (i - info.getCurrentAge()) * barWidth + 1;
499       int y1;
500       double v = info.getBalance(i);
501       int height;
502       int yOrigin = (int) (maxValue * scale);
503
504       if (v >= 0)
505       {
506         y1 = (int) ((maxValue - v) * scale);
507         height = yOrigin - y1;
508       }
509       else
510       {
511         y1 = yOrigin;
```

```
512           height = (int) (-v * scale);
513         }
514
515         if (i < info.getRetireAge()) g2.setPaint(colorPre);
516         else if (v >= 0) g2.setPaint(colorGain);
517         else g2.setPaint(colorLoss);
518         Rectangle2D bar =
519             new Rectangle2D.Double(x1, y1, barWidth - 2, height);
520         g2.fill(bar);
521         g2.setPaint(Color.black);
522         g2.draw(bar);
523       }
524    }
525
526    /**
527     * Устанавливает цвет графика для периода до выхода на пенсию
528     * @param color Требующийся цвет
529    */
530    public void setColorPre(Color color)
531    {
532      colorPre = color;
533      repaint();
534    }
535
536    /**
537     * Устанавливает цвет графика для периода после выхода на пенсию,
538     * когда остаток на пенсионном счету еще положительный
539      * @param color Требующийся цвет
540    */
541    public void setColorGain(Color color)
542    {
543      colorGain = color;
544      repaint();
545    }
546
547    /**
548      * Устанавливает цвет графика для периода после выхода на пенсию,
549      * когда остаток на пенсионном счету уже отрицательный
550      * @param color Требующийся цвет
551    */
552    public void setColorLoss(Color color)
553    {
554      colorLoss = color;
555      repaint();
556    }
557
558    public Dimension getPreferredSize()
559           { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
560 }
```

Листинг 5.6. Исходный код из файла `retire/RetireResources.java`

```
1  package retire;
2
3  import java.awt.*;
4
```

```
5  /**
6   * Нестроковые ресурсы для пользовательского интерфейса на английском
7   * языке программы калькуляции пенсионных сбережений
8   * @version 1.21 2001-08-27
9   * @author Cay Horstmann
10 */
11 public class RetireResources extends java.util.ListResourceBundle
12 {
13   private static final Object[][] contents = {
14   // НАЧАЛО ИНТЕРНАЦИОНАЛИЗАЦИИ
15      { "colorPre", Color.blue }, { "colorGain", Color.white },
16      { "colorLoss", Color.red }
17   // КОНЕЦ ИНТЕРНАЦИОНАЛИЗАЦИИ
18   };
19
20   public Object[][] getContents()
21   {
22     return contents;
23   }
24 }
```

Листинг 5.7. Исходный код из файла `retire/RetireResources_de.java`

```
1  package retire;
2
3  import java.awt.*;
4
5  /**
6   * Нестроковые ресурсы для пользовательского интерфейса на немецком
7   * языке программы калькуляции пенсионных сбережений
8   * @version 1.21 2001-08-27
9   * @author Cay Horstmann
10 */
11 public class RetireResources_de extends java.util.ListResourceBundle
12 {
13   private static final Object[][] contents = {
14   // НАЧАЛО ИНТЕРНАЦИОНАЛИЗАЦИИ
15      { "colorPre", Color.yellow }, { "colorGain", Color.black },
16      { "colorLoss", Color.red }
17   // КОНЕЦ ИНТЕРНАЦИОНАЛИЗАЦИИ
18   };
19
20   public Object[][] getContents()
21   {
22     return contents;
23   }
24 }
```

Листинг 5.8. Исходный код из файла `retire/RetireResources_zh.java`

```
1  package retire;
2
3  import java.awt.*;
4
5  /**
6   * Нестроковые ресурсы для пользовательского интерфейса на китайском
7   * языке программы калькуляции пенсионных сбережений
```

```
8    * @version 1.21 2001-08-27
9    * @author Cay Horstmann
10 */
11 public class RetireResources_zh extends java.util.ListResourceBundle
12 {
13   private static final Object[][] contents = {
14   // НАЧАЛО ИНТЕРНАЦИОНАЛИЗАЦИИ
15      { "colorPre", Color.red }, { "colorGain", Color.blue },
16      { "colorLoss", Color.yellow }
17   // КОНЕЦ ИНТЕРНАЦИОНАЛИЗАЦИИ
18   };
19
20   public Object[][] getContents()
21   {
22     return contents;
23 }
24 }
```

Листинг 5.9. Содержимое файла свойств `retire/RetireStrings.properties`

```
1 language=Language
2 computeButton=Compute
3 savings=Prior Savings
4 contrib=Annual Contribution
5 income=Retirement Income
6 currentAge=Current Age
7 retireAge=Retirement Age
8 deathAge=Life Expectancy
9 inflationPercent=Inflation
10 investPercent=Investment Return
11 retire=Age: {0,number} Balance: {1,number,currency}
```

Листинг 5.10. Содержимое файла свойств `retire/RetireStrings_de.properties`

```
1 language=Sprache
2 computeButton=Rechnen
3 savings=Vorherige Ersparnisse
4 contrib=J\u00e4hrliche Einzahlung
5 income=Einkommen nach Ruhestand
6 currentAge=Jetziges Alter
7 retireAge=Ruhestandsalter
8 deathAge=Lebenserwartung
9 inflationPercent=Inflation
10 investPercent=Investitionsgewinn
11 retire=Alter: {0,number} Guthaben: {1,number,currency}
```

Листинг 5.11. Содержимое файла свойств `retire/RetireStrings_zh.properties`

```
1 language=\u8bed\u8a00
2 computeButton=\u8ba1\u7b97
3 savings=\u65e2\u5b58
4 contrib=\u6bcf\u5e74\u5b58\u91d1
5 income=\u9000\u4f11\u6536\u5165
6 currentAge=\u73b0\u9f84
7 retireAge=\u9000\u4f11\u5e74\u9f84
```

```
8 deathAge=\u9884\u671f\u5bff\u547d
9 inflationPercent=\u901a\u8d27\u81a8\u6da8
10 investPercent=\u6295\u8d44\u62a5\u916c
11 retire=\u5e74\u9f84: {0,number} \u603b\u7ed3: {1,number,currency}
```

Теперь вам должно быть понятно, как пользоваться средствами интернационализации в Java. В частности, для перевода текстовой информации на многие языки служат комплекты ресурсов, а средства форматирования и сортировки применяются для обработки текста с учетом региональных настроек. В следующей главе будут рассмотрены вопросы программирования графики расширенными средствами из библиотеки Swing.

ГЛАВА 6

Расширенные средства Swing

В этой главе...

- Списки
- Таблицы
- Деревья
- Текстовые компоненты
- Индикаторы состояния
- Организаторы и декораторы компонентов

В этой главе продолжается начатое в первом томе обсуждение набора компонентов из библиотеки Swing, предназначенных для создания графического пользовательского интерфейса (ГПИ). В первом томе данной книги были представлены только самые основные и наиболее употребительные компоненты, хотя в библиотеке Swing имеется немало других много более сложных компонентов. Сначала в этой главе рассматриваются списки, таблицы и деревья, которым посвящена большая ее часть. Затем обсуждаются текстовые компоненты и приводятся дополнительные сведения о простых текстовых полях и областях, о которых уже упоминалось в первом томе. По ходу изложения материала главы будет показано, каким образом в текстовые поля добавляются средства проверки достоверности вводимых данных и счетчики, а также отображается структурированный текст, например в формате HTML. Далее будут рассмотрены компоненты для индикации хода выполнения длительных операций. И в завершение главы будут представлены организаторы компонентов в виде панелей с вкладками и настольных панелей с внутренними фреймами.

Списки

Для того чтобы предоставить пользователю возможность выбирать среди нескольких значений параметров, вместо групп кнопок-переключателей или флажков (которые занимают довольно много места) можно воспользоваться обычными или комбинированными списками. Комбинированные списки имеют довольно простую структуру и подробно рассматриваются в первом томе данной книги. Компонент `JList`, реализующий списки, обладает намного более развитыми средствами для создания сложных структур, подобных деревьям и таблицам. Именно поэтому в этой главе сначала рассматриваются простые списки.

Чаще всего применяются списки, состоящие из символьных строк, но элементами списка могут быть произвольные объекты, допускающие управление своим внешним видом. Компонент `JList` обладает изящной внутренней структурой, благодаря которой и достигается такая универсальность. К сожалению, разработчики библиотеки Swing уделили чрезмерно большое внимание изящности структуры данного компонента в ущерб простоте его использования. В ряде простых ситуаций компонент `JList` оказывается не очень удобным в употреблении, поскольку вынуждает пользоваться универсальными механизмами. Сначала в этом разделе рассматривается простой и очень распространенный пример организации списка символьных строк, а затем более сложный пример создания списка, который наглядно демонстрирует гибкость компонента `JList`.

Компонент `JList`

Этот компонент служит для отображения ряда элементов в одном окне со списком. На рис. 6.1 показан очень простой пример списка, в котором пользователь может подобрать характеристики для лисы: `quick` (шустрая), `brown` (бурая), `hungry` (голодная), `wild` (дикая), а также `static`, `private` и `final` (тут авторы исчерпали известные им характеристики животных и предложили первые пришедшие на память ключевые слова Java). Эти характеристики будут использованы в предложении о лисе, которая перепрыгивает через ленивую собаку. Например, при выборе в списке характеристик `static` и `final` получится предложение `The static final fox jumps over the lazy dog`, которое появится в рабочем окне рассматриваемого здесь примера программы.

Начиная с версии Java SE 7 компонент `JList` стал обобщенным. В качестве параметра типа в нем служат типы значений, которые может выбирать пользователь. В рассматриваемом здесь примере используется строковый тип `JList<String>`.

Составление такого списка следует начинать с создания массива символьных строк, а затем передать его конструктору компонента `Jlist` следующим образом:

```
String[] words= { "quick", "brown", "hungry", "wild", . . . };
JList<String> wordList = new JList<>(words);
```

В простых списках не предусмотрены средства автоматической прокрутки. Поэтому для просмотра длинного списка его нужно расположить на панели с полосой прокрутки типа `JScrollPane`, как показано ниже, а затем вставить эту панель (а не список) в другую, окружающую ее панель.

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

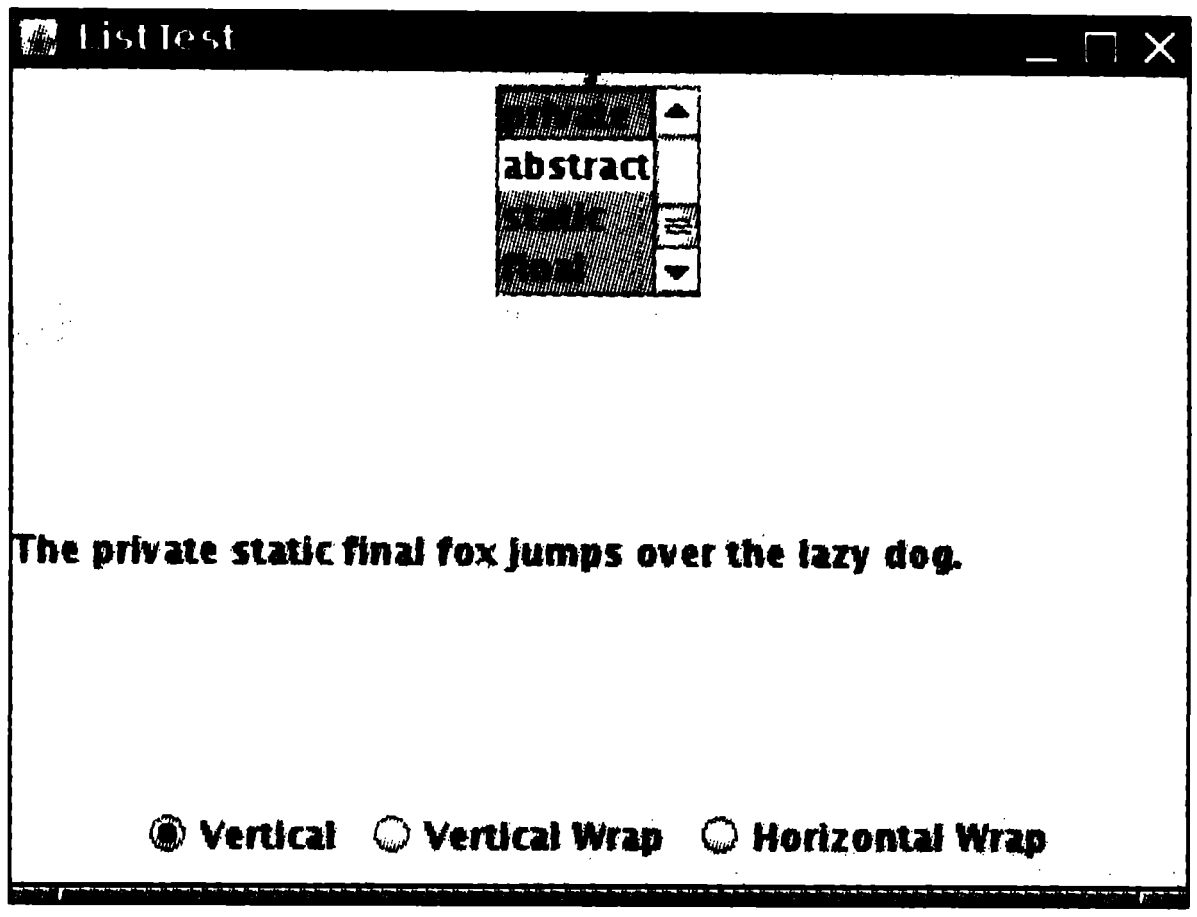


Рис. 6.1. Пример простого списка

Следует признать, что разделение функций отображения списка и механизма прокрутки выглядит изящно теоретически, но неудобно на практике. Ведь редкий список обходится без прокрутки его содержимого. Поэтому совершенно неоправданно заставлять разработчиков программировать самостоятельно те функции, которые могли бы быть реализованы по умолчанию, только для того, чтобы они оценили универсальность и изящество структуры компонента `JList`.

По умолчанию в окне списка отображаются восемь элементов. Но это количество можно изменить с помощью метода `setVisibleRowCount()`, как показано в приведенном ниже примере.

```
wordList.setVisibleRowCount(4); // отобразить четыре элемента списка
```

При составлении списка можно выбрать один из трех вариантов расположения его элементов.

- **`JList.VERTICAL` (по умолчанию).** Все элементы списка располагаются вертикально друг под другом.
- **`JList.VERTICAL_WRAP`.** Если количество элементов списка превышает заданное число отображаемых символьных строк, формируются новые столбцы (рис. 6.2).
- **`JList.HORIZONTAL_WRAP`.** Если количество элементов списка превышает заданное число отображаемых символьных строк, формируются новые столбцы, но они заполняются в горизонтальном направлении. Отличие между вариантами расположения `JList.VERTICAL_WRAP` и `JList.HORIZONTAL_WRAP` нетрудно понять, сравнив на рис. 6.2 расположение элементов `quick`, `brown` и `hungry`.

По умолчанию пользователь может выбрать сразу несколько элементов из списка. Например, для выбора несмежных элементов из списка нужно нажать клавишу <Ctrl> и, не опуская ее, щелкнуть на них по очереди кнопкой мыши. А для выбора нескольких смежных элементов из списка следует нажать клавишу <Shift> и, не опуская ее, щелкнуть кнопкой мыши сначала на первом из них, а затем на последнем.

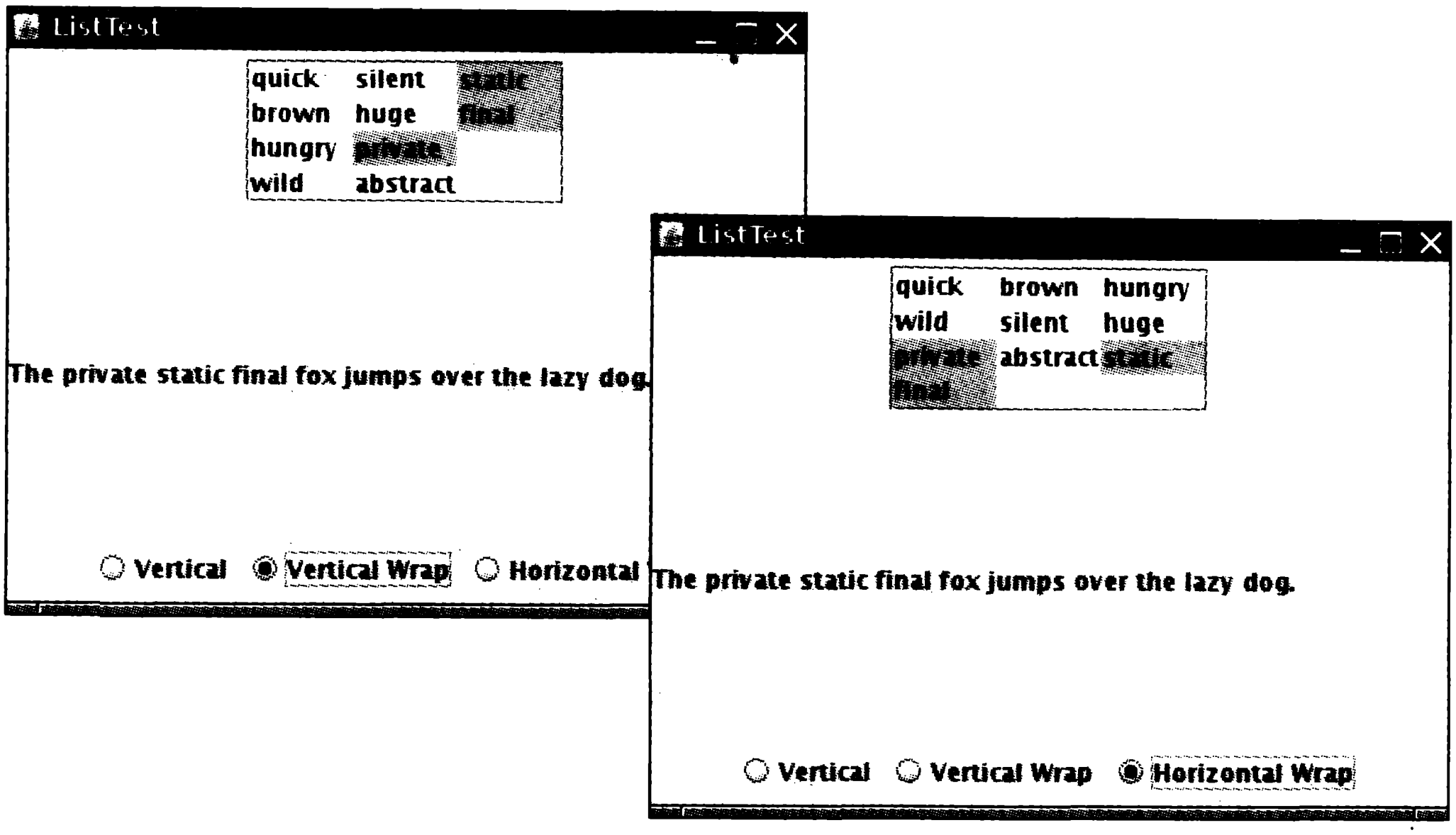


Рис. 6.2. Списки с вертикальным горизонтальным расположением элементов в дополнительных столбцах

Для ограничения вариантов выбора из списка предусмотрен метод `setSelectionMode()`. В приведенном ниже фрагменте кода показано, каким образом накладывается подобное ограничение.

```
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
// разрешить выбор элементов по одному
wordList.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
// разрешить выбор одного элемента или ряда элементов
```

Как упоминалось в первом томе данной книги, базовые компоненты пользовательского интерфейса посылают сообщения о событиях, связанных с действиями пользователя. А в списках используется другой механизм уведомления о наступающих событиях. Вместо события действия при обращении к спискам следует принимать событие выбора из списка. Для этого в компонент составления списков нужно ввести приемник событий выбора из списка и реализовать в нем следующий метод:

```
public void valueChanged(ListSelectionEvent evt)
```

Когда пользователь выбирает элемент из списка, инициируется несколько событий. Допустим, пользователь щелкнул на элементе списка. При нажатии кнопки мыши передается событие об изменении выбора. Это промежуточное событие. Приведенный ниже метод возвращает логическое значение `true`, если выбор не был окончательным.

```
event.getValueIsAdjusting()
```

При отпускании кнопки мыши генерируется еще одно событие. На этот раз метод `isAdjusting()` возвращает логическое значение `false`. Если обработка промежуточных событий не нужна, то можно отслеживать только те события, для которых метод `isAdjusting()` возвращает логическое значение `false`. Но для организации

быстрого реагирования на действия пользователя с мышью придется обрабатывать все события.

После уведомления о событии необходимо выяснить, какие именно элементы были выбраны из списка. Если список находится в режиме выбора только одного элемента, следует вызвать метод `getSelectedValue()`, чтобы получить значение, обозначающее тип элемента, выбранного из списка. В противном случае следует вызвать метод `getSelectedValuesList()`, возвращающий список, содержащий все выбранные элементы. Полученный результат обрабатывается обычным образом, как показано ниже.

```
for (String value : wordList.getSelectedValuesList())
   // сделать что-нибудь с полученным значением value
```

НА ЗАМЕТКУ! В компоненте `JList` не предусмотрено реагирование на двойной щелчок кнопкой мыши. Разработчики библиотеки Swing предполагали, что список предназначается только для выбора элементов, а для выполнения какого-нибудь другого действия следует щелкнуть на соответствующей кнопке. Но в некоторых реализациях пользовательского интерфейса двойной щелчок кнопкой мыши рассматривается одновременно как выбор элемента и команда на выполнение соответствующего действия. Поэтому если требуется предоставить пользователю такую возможность, необходимо сначала связать со списком обработчик событий от мыши, а затем перехватывать и обрабатывать подобные события, как показано ниже.

```
public void mouseClicked(MouseEvent evt)
{
  if (evt.getClickCount() == 2)
  {
    JList source = (JList) evt.getSource();
    Object[] selection = source.getSelectedValues();
    doAction(selection);
  }
}
```

В листинге 6.1 приведен исходный код примера программы, демонстрирующего применение списка, элементами которого являются символьные строки. Обратите внимание на метод `valueChanged()`, формирующий символьную строку сообщения из элементов, выбираемых из списка.

Листинг 6.1. Исходный код из файла `list/ListFrame.java`

```
1  package list;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import javax.swing.event.*;
7
8  /**
9   * Этот фрейм содержит список слов и метку, отображающую
10  * предложение, составляемое из слов, выбираемых из данного списка
11  * Для того чтобы выбрать сразу несколько слов из списка, можно
12  * воспользоваться комбинациями <Ctrl+щелчок> и <Shift+щелчок>
13 */
15 class ListFrame extends JFrame
```

```
16 {
17   private static final int DEFAULT_WIDTH = 400;
18   private static final int DEFAULT_HEIGHT = 300;
19
20   private JPanel listPanel;
21   private JList<String> wordList;
22   private JLabel label;
23   private JPanel buttonPanel;
24   private ButtonGroup group;
25   private String prefix = "The ";
26   private String suffix = "fox jumps over the lazy dog.";
27
28   public ListFrame()
29   {
30     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
31
32     String[] words =
33       { "quick", "brown", "hungry", "wild", "silent",
34         "huge","private", "abstract", "static", "final" };
35
36    wordList = new JList<>(words);
37    wordList.setVisibleRowCount(4);
38    JScrollPane scrollPane = new JScrollPane(wordList);
39
40    listPanel = new JPanel();
41    listPanel.add(scrollPane);
42    wordList.addListSelectionListener(new ListSelectionListener()
43      {
44        public void valueChanged(ListSelectionEvent event)
45        {
46          StringBuilder text = new StringBuilder(prefix);
47          for (String value : wordList.getSelectedValuesList())
48          {
49            text.append(value);
50            text.append(" ");
51          }
52          text.append(suffix);
53
54          label.setText(text.toString());
55        }
56      });
57
58    buttonPanel = new JPanel();
59    group = new ButtonGroup();
60    makeButton("Vertical", JList.VERTICAL);
61    makeButton("Vertical Wrap", JList.VERTICAL_WRAP);
62    makeButton("Horizontal Wrap", JList.HORIZONTAL_WRAP);
63
64    add(listPanel, BorderLayout.NORTH);
65    label = new JLabel(prefix + suffix);
66    add(label, BorderLayout.CENTER);
67    add(buttonPanel, BorderLayout.SOUTH);
68  }
69
70  /**
71    * Создает кнопку-переключатель для выбора варианта
72    * расположения элементов в списке
73    * @param label Метка кнопки
```

```
74      * @param orientation Вариант расположения элементов в списке
75    */
76    private void makeButton(String label, final int orientation)
77    {
78      JRadioButton button = new JRadioButton(label);
79      buttonPanel.add(button);
80      if (group.getButtonCount() == 0) button.setSelected(true);
81      group.add(button);
82      button.addActionListener(new ActionListener()
83        {
84          public void actionPerformed(ActionEvent event)
85          {
86            wordList.setLayoutOrientation(orientation);
87            listPanel.revalidate();
88          }
89        });
90    }
91  }
```

javax.swing.JList<E> 1.2

- `JList(E[] items)`

  Составляет список из указанных элементов для их отображения.

- `int getVisibleRowCount()`
- `void setVisibleRowCount(int c)`

  Устанавливают или получают количество строк с элементами, которые можно увидеть в списке без прокрутки.

- `int getLayoutOrientation()` **1.4**
- `void setLayoutOrientation(int orientation)` **1.4**

  Устанавливают или получают вариант расположения элементов в списке.

  Параметры: *orientation* Принимает значение одной из следующих констант: VERTICAL, VERTICAL_WRAP или HORIZONTAL_WRAP

- `int getSelectionMode()`
- `void setSelectionMode(int mode)`

  Устанавливают или получают режим, определяющий возможность выбрать сразу один или несколько элементов из списка.

  Параметры: *mode* Принимает значение одной из следующих констант: SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, MULTIPLE_INTERVAL_SELECTION

- `void addListSelectionListener(ListSelectionListener listener)`

  Связывает со списком приемник событий, который уведомляется о каждом изменении выбора элементов из списка.

- `List<E> getSelectedValues()` **7**

  Возвращает выбранные из списка значения или пустой список, если ничего не выбрано.

- `E getSelectedValue()`

  Возвращает первое выбранное из списка значение или пустое значение `null`, если ничего не выбрано.

javax.swing.event.ListSelectionListener 1.2

- `void valueChanged(ListSelectionEvent e)`

  Вызывается при каждом изменении выбора элементов из списка.

Модели списков

В предыдущем разделе описан наиболее употребительный способ использования компонента `JList` для составления списков. В частности, для составления простого списка необходимо выполнить следующие действия.

1. Указать фиксированный ряд символьных строк для отображения в списке.
2. Расположить список на панели, снабженной полосой прокрутки.
3. Организовать обработку событий, наступающих при выборе элементов из списка.

А далее рассматриваются следующие более сложные варианты списков.

- Очень длинные списки.
- Списки с изменяющимся содержимым.
- Списки, состоящие из других элементов, кроме символьных строк.

В предыдущем примере рассматривалось построение с помощью компонента `JList` списка с фиксированным набором символьных строк. Но набор элементов списка может изменяться. Как же организовать ввод и удаление элементов из списка? Как ни странно, но в классе `JList` отсутствуют методы для подобных целей. Поэтому придется более подробно рассмотреть внутреннюю структуру списка. В частности, для построения списков в компоненте `JList` применяется шаблон проектирования "модель–представление–контроллер" (MVC), чтобы отделить базовые данные (т.е. набор объектов) от их визуального представления (т.е. отображения элементов списка столбцами).

Класс `JList` отвечает за визуальное представление данных, и ему ничего неизвестно о способе их хранения. Ему известно только, что данные можно извлечь, обратившись к объекту из класса, реализующего интерфейс `ListModel`:

```
public interface ListModel<E>
{
  int getSize();
  E getElementAt(int i);
```

```
  void addListDataListener(ListDataListener l);
  void removeListDataListener(ListDataListener l);
}
```

С помощью этого интерфейса компонент `JList` может получать подсчет элементов списка и извлекать любой из них. Кроме того, объект типа `JList` может выступать в качестве приемника событий типа `ListDataListener`. В этом случае он уведомляется об изменении состава списка, чтобы перерисовать его. В чем же польза от такой универсальности? Почему бы не поступить проще, сохраняя в объекте типа `JList` массив других объектов?

Дело в том, что в данном интерфейсе не указывается способ хранения данных. Более того, он даже не требует сохранять данные! В частности, метод `getElementAt()` можно переопределить таким образом, чтобы он вычислял требуемое значение всякий раз, когда он вызывается. Такой способ удобен для представления очень большого набора данных без сохранения значений.

Рассмотрим еще один, немного упрощенный пример. Речь пойдет о составлении списка, из которого пользователь может выбрать любой из всех возможных вариантов трехбуквенных слов на английском языке (рис. 6.3). В английском языке насчитывается **26×26×26=17576** трехбуквенных сочетаний. Вместо хранения всех этих сочетаний было бы разумнее вычислять их во время прокрутки списка.

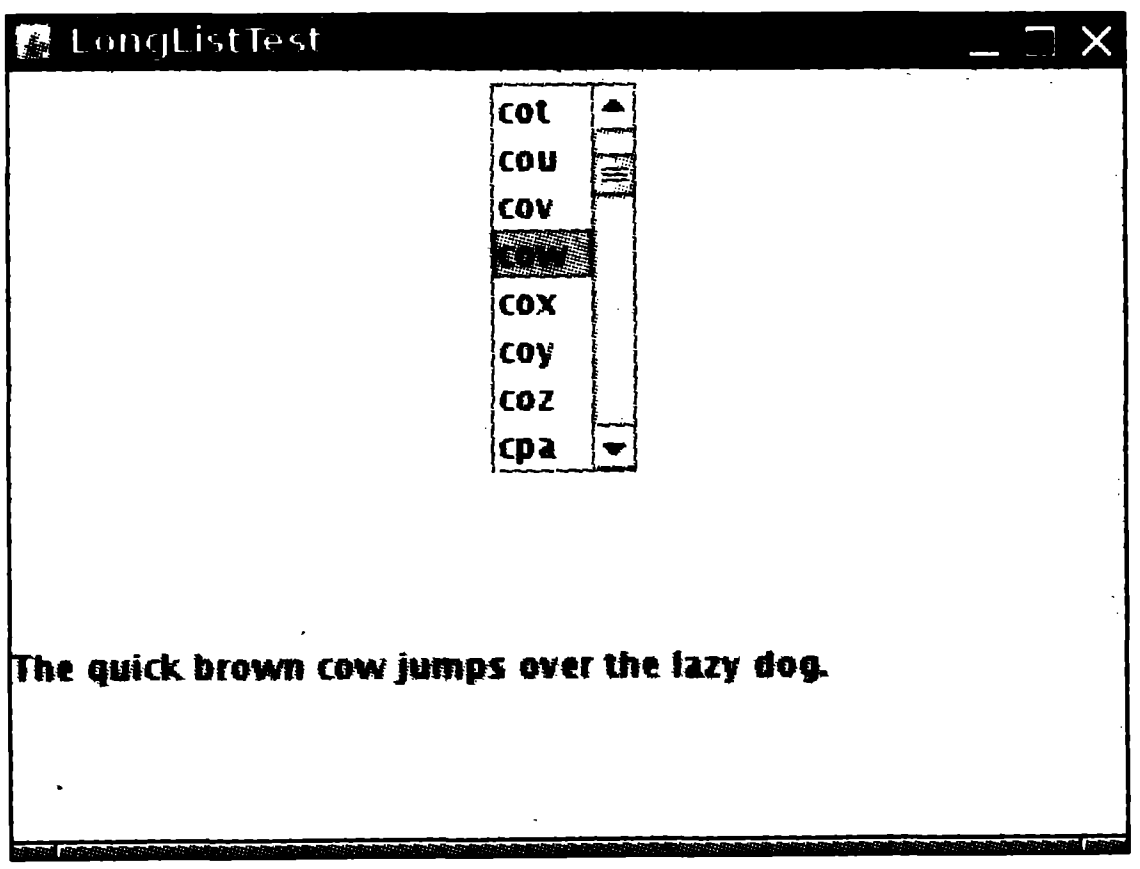


Рис. 6.3. Пример выбора варианта из очень длинного списка

Оказывается, что составить такой список довольно просто. Рутинные операции ввода и удаления приемников событий уже реализованы в классе `AbstractListModel`, который можно расширить конкретным классом. Остается только предоставить методы `getSize()` и `getElementAt()`, как показано ниже.

```
class WordListModel extends AbstractListModel<String>
{
  public WordListModel(int n) { length = n; }
  public int getSize() { return (int) Math.pow(26, length); }
  public String getElementAt(int n)
  {
    // вычислить n-ю символьную строку
    . . .
  }
  . . .
}
```

Подробнее о вычислении *n*-й символьной строки можно узнать из листинга 6.3. Имея теперь модель, можно составить список, который можно прокручивать, просматривая и выбирая элементы, предоставляемые данной моделью, как показано в приведенном ниже фрагменте кода. Основная идея состоит в том, что символьные строки не сохраняются, а вычисляются, причем формируются только те символьные строки, которые требуются пользователю в данный момент.

```
JList<String> wordList = new JList<>(new WordListModel(3));
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane scrollPane = new JScrollPane(wordList);
```

Необходимо сделать еще одну установку: уведомить компонент составления списков о том, что все элементы списка имеют фиксированную высоту и ширину. Установить размеры ячейки проще всего, определив *значение прототипной ячейки* следующим образом:

```
wordList.setPrototypeCellValue("www");
```

Значение прототипной ячейки используется для определения размеров всех ячеек. (В данном случае используется символьная строка `"www"`, потому что в большинстве шрифтов буква `w` оказывается самой широкой из всех прописных букв английского алфавита.) С другой стороны, размеры ячейки можно задать явным образом:

```
wordList.setFixedCellWidth(50);
wordList.setFixedCellHeight(15);
```

Если не установить значение прототипной ячейки или фиксированные размеры конкретной ячейки, ее высота и ширина будут вычисляться отдельно для каждого элемента списка, на что потребуется немало времени. В листинге 6.2 приведен исходный код класса фрейма из рассматриваемого здесь примера программы.

На практике очень длинные списки применяются крайне редко, потому что пользователям не совсем удобно искать и выбирать в них требуемые элементы. Кроме того, набор выбранных пользователем элементов имеет достаточно малый объем и может быть сохранен непосредственно в компоненте `JList`. Такой механизм избавляет от необходимости обращаться к модели списка отдельно от самого компонента. С другой стороны, класс `JList` оказывается совместимым с классами `JTable` и `JTree`.

Листинг 6.2. Исходный код из файла `longList/LongListFrame.java`

```
1  package longList;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.event.*;
6
7  /**
8   * Этот фрейм содержит длинный список слов и метку, отображающую
9   * предложение, составляемое из слов, выбираемых из данного списка
10 */
11 public class LongListFrame extends JFrame
12 {
13   private JList<String> wordList;
14   private JLabel label;
15   private String prefix = "The quick brown ";
16   private String suffix = " jumps over the lazy dog.";
```

```
17
18   public LongListFrame()
19   {
20     wordList = new JList<String>(new WordListModel(3));
21     wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
22     wordList.setPrototypeCellValue("www");
23     JScrollPane scrollPane = new JScrollPane(wordList);
24
25     JPanel p = new JPanel();
26     p.add(scrollPane);
27     wordList.addListSelectionListener(new ListSelectionListener()
28       {
29         public void valueChanged(ListSelectionEvent evt)
30         {
31           setSubject(wordList.getSelectedValue());
32         }
33       });
34
35     Container contentPane = getContentPane();
36     contentPane.add(p, BorderLayout.NORTH);
37     label = new JLabel(prefix + suffix);
38     contentPane.add(label, BorderLayout.CENTER);
39     setSubject("fox");
40     pack();
41   }
42   /**
43    * Устанавливает в метке новый субъект
44    * @param word Новый субъект, перепрыгивающий через ленивую собаку
45   */
46   public void setSubject(String word)
47   {
48     StringBuilder text = new StringBuilder(prefix);
49     text.append(word);
50     text.append(suffix);
51     label.setText(text.toString());
52   }
53 }
```

Листинг 6.3. Исходный код из файла `longList/WordListModel.java`

```
1  package longList;
2
3  import javax.swing.*;
4
5  /**
6   * Модель, динамически формирующая слова из n букв
7  */
8  public class WordListModel extends AbstractListModel<String>
9  {
10   private int length;
11   public static final char FIRST = 'a';
12   public static final char LAST = 'z';
13
14   /**
15     * Конструирует модель
16     * @param n Длина слова
17   */
```

```
18   public WordListModel(int n)
19   {
20     length = n;
21   }
22
23   public int getSize()
24   {
25     return (int) Math.pow(LAST - FIRST + 1, length);
26   }
27
28   public String getElementAt(int n)
29   {
30     StringBuilder r = new StringBuilder();
31     for (int i = 0; i < length; i++)
32     {
33       char c = (char) (FIRST + n % (LAST - FIRST + 1));
34       r.insert(0, c);
35       n = n / (LAST - FIRST + 1);
36     }
37     return r.toString();
38   }
39 }
```

javax.swing.JList<E> 1.2

- `JList(ListModel<E> dataModel)`
- Составляет список, в котором отображаются элементы из указанной модели.
- `E getPrototypeCellValue()`
- `void setPrototypeCellValue(E newValue)`
- Получают или устанавливают значение прототипной ячейки, используемое для определения ширины и высоты каждой ячейки в списке. По умолчанию принимается пустое значение `null`, в результате чего размеры вычисляются отдельно для каждого элемента списка.
- `void setFixedCellWidth(int width)`
- `void setFixedCellHeight(int height)`
- Если значение параметра *width* или *height* больше нуля, эти методы определяют ширину или высоту (в пикселях) всех ячеек в списке. По умолчанию параметр *width* или *height* принимает значение **-1**, которое означает определение размеров каждой ячейки.

javax.swing.ListModel<E> 1.2

- `int getSize()`
- Возвращает количество элементов в модели.
- `E getElementAt(int position)`
- Возвращает из модели элемент, находящийся на указанной позиции.

Ввод и удаление значений

Непосредственно редактировать набор значений в списке нельзя. Для этого нужно сначала обеспечить доступ к *модели* и только затем выполнять операции удаления и добавления элементов в список. Однако это проще сказать, чем сделать. Итак, сначала нужно получить ссылку на модель списка следующим образом:

```
ListModel<String> model = list.getModel();
```

Но это ничего не дает. Ведь, как упоминалось ранее, в интерфейсе `ListModel` отсутствуют методы добавления и удаления элементов из списка. Основное назначение модели состоит в том, чтобы исключить потребность в *хранении* элементов. Попробуем поступить иначе. Один из конструкторов класса `JList` получает в качестве параметра вектор объектов, как показано ниже.

```
Vector<String> values = new Vector<String>();
values.addElement("quick");
values.addElement("brown");
. . .
JList<String> list = new JList<>(values);
```

Теперь можно редактировать вектор, добавляя и удаляя из него элементы. Но списку ничего не известно о подобных изменениях, поэтому он никак не отреагирует на них. В частности, список не будет обновляться при добавлении новых элементов. Следовательно, данный конструктор не приносит особой пользы.

Вместо этого следует создать объект типа `DefaultListModel`, заполнить его нужными значениями и связать со списком, как показано в приведенном ниже фрагменте кода. Класс `DefaultListModel` реализует интерфейс `ListModel` и управляет коллекцией объектов.

```
DefaultListModel<String> model = new DefaultListModel<>();
model.addElement("quick");
model.addElement("brown");
. . .
JList<String> list = new JList<>(model);
```

Теперь в объект `model` можно добавлять значения и удалять их из него, как следует из приведенного ниже фрагмента кода. Этот объект будет уведомлять список о происходящих изменениях, а список, в свою очередь, изменять отображаемые данные.

```
model.removeElement("quick");
model.addElement("slow");
```

Исторически сложилось так, что методы из класса `DeafultListModel` отличаются своими именами от методов из классов коллекций. Для внутреннего хранения значений в исходной модели списка типа `DeafultListModel` используется вектор.

ВНИМАНИЕ! Некоторые конструкторы класса `JList` создают список из массива или вектора объектов или символьных строк. На первый взгляд, эти конструкторы используют класс `DeafultListModel` для хранения значений элементов. Но это совсем не так. Они создают обычную модель, в которой отсутствует уведомление об изменении содержимого. Ниже приводится исходный код конструктора для создания списка типа `JList` из объекта типа `Vector`.

```
public JList(final Vector<? extends E> listData)
{
  this (new AbstractListModel<E>()
  {
```

```
    public int getSize() { return listData.size(); }
    public E getElementAt(int i) { return listData.elementAt(i); }
  });
}
```

Это означает, что если вектор изменится после создания списка, то в списке могут вперемешку присутствовать старые и новые значения до тех пор, пока он не будет полностью перерисован. (Ключевое слово `final` в приведенном выше конструкторе не накладывает запрет на изменение самого вектора, а только запрещает изменять значение ссылки на объект типа `listData`. Это ключевое слово требуется, потому что объект типа `listData` используется во внутреннем классе.)

javax.swing.JList<E> 1.2

- `ListModel<E> getModel()`

  Возвращает модель для данного списка.

javax.swing.DefaultListModel<E> 1.2

- `void addElement(E obj)`
- Добавляет объект в конце модели.
- `boolean removeElement(Object obj)`
- Удаляет из модели первый экземпляр объекта и возвращает логическое значение `true`, если такой объект обнаружен, а иначе — логическое значение `false`.

Воспроизведение значений

Все рассмотренные до сих пор списки содержали символьные строки. Но список можно также составить из пиктограмм, передав ему массив или вектор объектов типа `Icon`. Более того, список можно составить из любых объектов и воспроизвести их любым требующимся способом.

И хотя класс `JList` позволяет воспроизвести символьные строки и пиктограммы в списке автоматически, для воспроизведения произвольных объектов необходимо установить в объекте типа `JList` *средство воспроизведения ячеек из списка*. В качестве такого средства может выступать экземпляр любого класса, реализующего интерфейс `ListCellRenderer`:

```
interface ListCellRenderer<E>
{
  Component getListCellRendererComponent(JList<? extends E> list,
  E value, int index, boolean isSelected, boolean cellHasFocus);
}
```

Метод, объявленный в данном интерфейсе, вызывается для каждой ячейки. Он возвращает компонент, предназначенный для воспроизведения содержимого ячейки. Этот компонент размещается там, где предполагается воспроизвести ячейку. Для того чтобы реализовать средство воспроизведения ячеек из списка, проще всего создать класс, расширяющий класс `JComponent`. Пример такого класса приведен ниже.

```
class MyCellRenderer extends JComponent implements ListCellRenderer<Тип>
{
  public Component getListCellRendererComponent(JList<? extends Тип> list,
    Тип value, int index, boolean isSelected, boolean cellHasFocus)
  {
    сохранить данные, необходимые для воспроизведения
    и определения размеров ячеек
    return this;
  }
  public void paintComponent(Graphics g)
  {
    здесь следует код для воспроизведения ячеек
  }
  public Dimension getPreferredSize()
  {
     здесь следует код для определения размеров ячеек
  }
  поля экземпляра
}
```

В листинге 6.4 приведен пример программы для графического отображения выбираемых из списка шрифтов (рис. 6.4). Для вывода названия каждого шрифта вызывается метод `paintComponent()`. При этом необходимо учитывать цвета воспроизводимого списка. Требуемые сведения о цвете переднего и заднего планов получаются с помощью методов `getForeground()/getBackground()` и `getSelectionForeground()/getSelection Background()` из класса `JList`. В методе `getPrefferedSize()` определяются типографские размеры воспроизводимой символьной строки способом, подробно описанным в главе 7 первого тома данной книги.

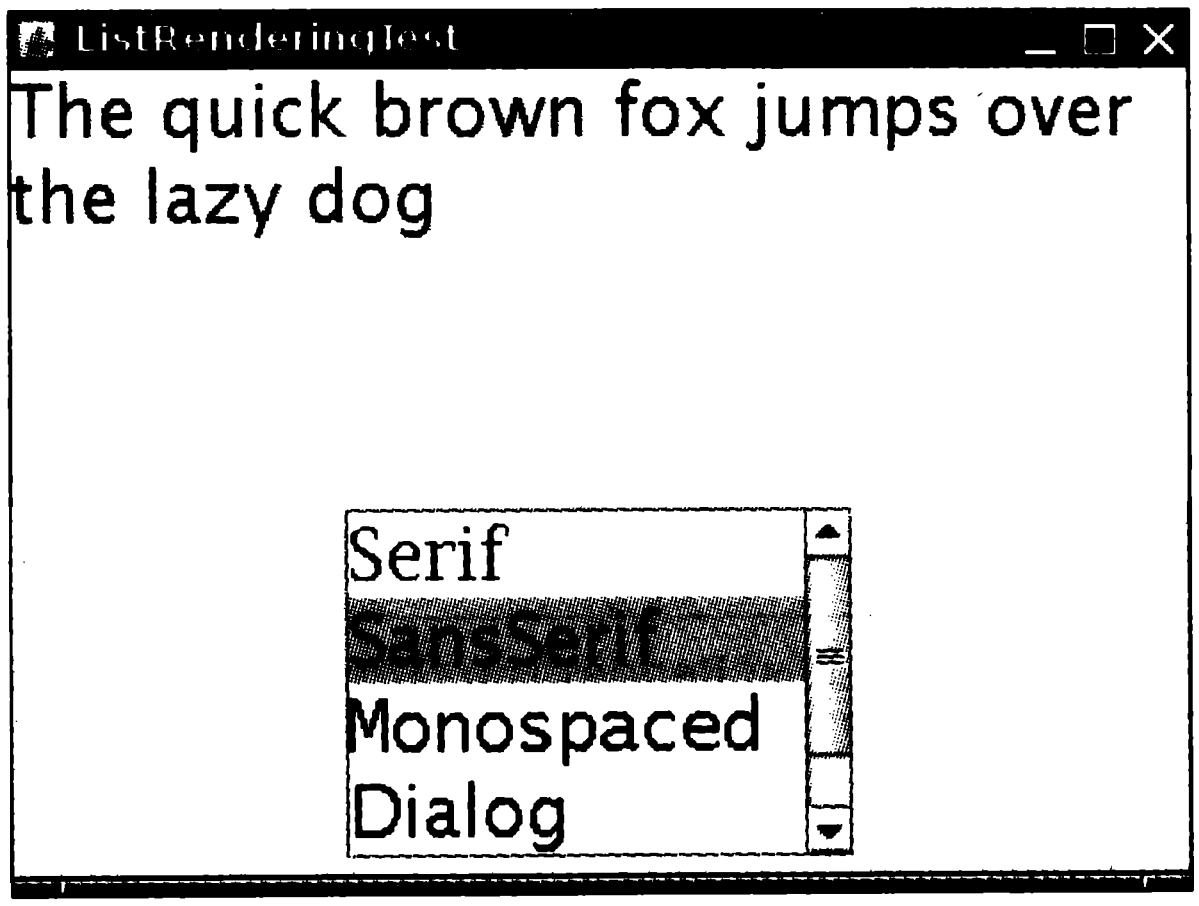


Рис. 6.4. Пример списка с воспроизведением ячеек разными шрифтами

Для установки средства воспроизведения ячеек из списка достаточно вызвать приведенный ниже метод `setCellRenderer()`. В итоге все ячейки будут выводиться из списка с помощью специального установленного средства их воспроизведения.

```
fontList.setCellRenrderer(new FontCellRenderer);
```

Существует более простой способ создания специальных средств воспроизведения ячеек из списка. Так, если изображение содержит текст, пиктограмму, а возможно, и меняющийся цвет, то для его воспроизведения можно обойтись соответственно настроенными средствами класса `JLabel`. Например, для выделения названия шрифта этим же шрифтом подходит следующее средство воспроизведения:

```
class FontCellRenderer extends JLabel implements ListCellRenderer<Font>
{
  public Component getListCellRendererComponent(JList<? extends Font> list,
    Font value, int index, boolean isSelected, boolean cellHasFocus)
  {
    Font font = (Font) value;
    setText(font.getFamily());
    setFont(font);
    setOpaque(true);
    setBackground(isSelected ? list.getSelectionBackground() :
       list.getBackground());
    setForeground(isSelected ? list.getSelectionForeground() :
       list.getForeground());
    return this;
  }
}
```

В данном случае не пришлось самостоятельно создавать методы `paintComponent()` или `getPrefferedSize()`, поскольку они уже реализованы в классе `JLabel`. Поэтому остается лишь задать параметры метки, т.е. указать текст, шрифт и цвет. Такой код оказывается удобным в тех случаях, когда какой-нибудь существующий компонент (в данном случае `JLabel`) поддерживает все функциональные возможности, требующиеся для воспроизведения содержимого ячеек из списка.

В рассматриваемом здесь примере программы можно было бы воспользоваться компонентом `JLabel`, но ее исходный код специально написан обобщенным, чтобы в него можно было легко внести изменения, если в ячейках из списка потребуется нарисовать что-нибудь другое.

НА ЗАМЕТКУ! Создавать новый компонент при каждом вызове метода `getListCellRendererComponent()` вряд ли оправдано. Так, если пользователю приходится прокручивать список, состоящий из большого количества элементов, то всякий раз будет создаваться новый компонент. Более эффективное и надежное решение состоит в перенастройке существующего компонента.

Листинг 6.4. Исходный код из файла `listRendering/FontCellRenderer.java`

```
1  package listRendering;
2
3  import java.awt.*;
4  import javax.swing.*;
5  /**
6   * Средство воспроизведения названий шрифтов самими шрифтами,
7   * выбираемыми в виде объектов типа Font из ячеек списка
8  */
9  public class FontCellRenderer extends JComponent
10        implements ListCellRenderer<Font>
11 {
12   private Font font;
```

```
13   private Color background;
14   private Color foreground;
15
16   public Component getListCellRendererComponent(
17       JList<? extends Font> list, Font value, int index,
18       boolean isSelected, boolean cellHasFocus)
19   {
20     font = value;
21     background = isSelected ? list.getSelectionBackground() :
22         list.getBackground();
23     foreground = isSelected ? list.getSelectionForeground() :
24         list.getForeground();
25     return this;
26   }
27
28   public void paintComponent(Graphics g)
29   {
30     String text = font.getFamily();
31     FontMetrics fm = g.getFontMetrics(font);
32     g.setColor(background);
33     g.fillRect(0, 0, getWidth(), getHeight());
34     g.setColor(foreground);
35     g.setFont(font);
36     g.drawString(text, 0, fm.getAscent());
37   }
38
39   public Dimension getPreferredSize()
40   {
41     String text = font.getFamily();
42     Graphics g = getGraphics();
43     FontMetrics fm = g.getFontMetrics(font);
44     return new Dimension(fm.stringWidth(text), fm.getHeight());
45   }
46 }
```

javax.swing.JList<E> 1.2

- `Color getBackground()`

  Возвращает цвет фона для ячеек, невыбранных из списка.

- `Color getSelectionBackground()`

  Возвращает цвет фона для ячеек, выбранных из списка.

- `Color getForeground()`

  Возвращает цвет переднего плана для ячеек, невыбранных из списка.

- `Color getSelectionForeground()`

  Возвращает цвет переднего плана для ячеек, выбранных из списка.

- `void setCellRenderer(ListCellRenderer<? super E> cellRenderer)`

  Устанавливает средство для воспроизведения ячеек из списка.

javax.swing.ListCellRenderer<E> 1.2

- `Component getListCellRendererComponent(JList<? extends E> list, E item, int index, boolean isSelected, boolean hasFocus)`

  Возвращает компонент, метод `paint()` которого воспроизводит содержимое ячеек из списка. Если размеры ячеек не фиксированы, этот компонент должен также реализовать метод `getPrefferedSize()`.

Параметры:		
	list	Список, ячейки которого воспроизводятся
	item	Воспроизводимый элемент списка
	index	Индекс, обозначающий место для хранения элемента в модели
	isSelected	Принимает логическое значение `true`, если указанная ячейка выбрана
	hasFocus	Принимает логическое значение `true`, если указанная ячейка обладает фокусом ввода

Таблицы

Компонент `JTable` служит для отображения таблицы в виде двухмерной сетки объектов. Таблицы широко применяются при построении ГПИ, поэтому разработчики библиотеки Swing уделили должное внимание созданию данного компонента. Компонент `JTable` имеет очень сложную структуру, но она, в отличие от других компонентов Swing, практически скрыта от пользователей. С помощью всего нескольких строк кода можно создать полноценную таблицу. Но для составления специальных таблиц придется написать дополнительный код, оформить их особый внешний вид и задать конкретное их поведение в разрабатываемой прикладной программе.

В этом разделе показывается, как создаются простые таблицы, как организуется взаимодействие пользователей с ними и как они чаще всего настраиваются. Аналогично другим сложным компонентам Swing, все особенности обращения с таблицами невозможно подробно описать в одном разделе. Поэтому для углубленного изучения данного вопроса рекомендуется следующая дополнительная литература: *Graphic Java™: Mastering the JFC, Volume II: Swing, 3rd Edition* Дэвида М. Гери (David M. Geary; издательство Prentice Hall, 1999 г.) или *Swing: Advanced Programming* Кима Топли (Kim Topley; издательство Prentice Hall, 1999 г.).

Простая таблица

Подобно компоненту `JList`, компонент `JTable` не хранит свои данные, а получает их из *модели таблицы*. Класс `JTable` содержит конструктор, который заключает двухмерный массив объектов в оболочку модели, используемой по умолчанию.

Именно такой способ применяется в рассматриваемом здесь первом примере создания таблиц. Модели таблиц более подробно обсуждаются далее в этой главе.

На рис. 6.5 приведена типичная таблица с описаниями свойств планет Солнечной системы. Следует иметь в виду, что в столбце `Gaseous` указывается логическое значение `true`, если планета состоит в основном из водорода и гелия, а иначе — логическое значение `false`. Значения в столбце `Color` пока еще не имеют какого-то определенного смысла, но этот столбец понадобится в следующих примерах.

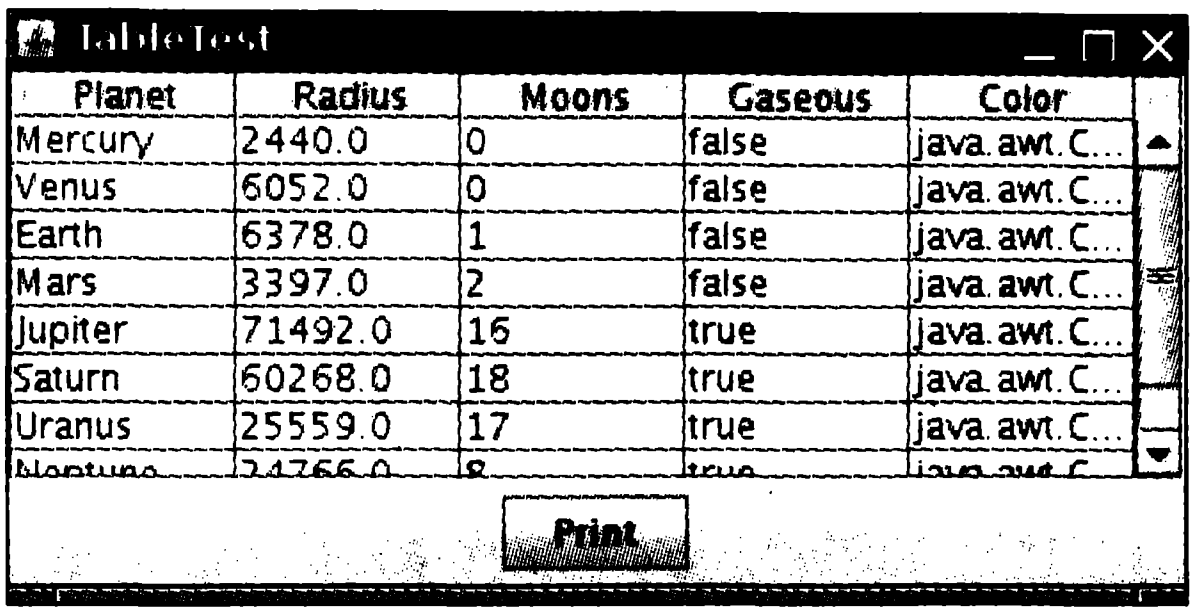


Рис. 6.5. Пример простой таблицы

Как следует из приведенного ниже фрагмента кода, взятого из листинга 6.5, где представлен исходный код рассматриваемого здесь примера программы, данные таблицы хранятся в виде двухмерного массива значений типа `Object`.

```
Object[][] cells =
{
  { "Mercury", 2440.0, 0, false, Color.YELLOW },
  { "Venus", 6052.0, 0, false, Color.YELLOW },
  . . .
}
```

НА ЗАМЕТКУ! В данном случае используются преимущества автоматического преобразования примитивных типов. Так, значения во втором, в третьем и четвертом столбцах преобразуются в объекты типа `Double`, `Integer` и `Boolean`.

Для отображения каждого объекта таблица вызывает метод `toString()`. Поэтому цвета в последнем столбце представлены в виде строк `java.awt.Color[r=...,g=...,b=...]`.

Сначала имена столбцов задаются в отдельном массиве символьных строк следующим образом:

```
String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color" };
```

Затем из массивов ячеек и имен столбцов составляется таблица, как показано ниже.

```
JTable table = new JTable(cells, columnNames);
```

Таблицу можно также снабдить полосами прокрутки. Для этого достаточно заключить ее в оболочку компонента `JScrollPane` следующим образом:

```
JScrollPane pane = new JScrollPane(table);
```

НА ЗАМЕТКУ! Следует иметь в виду, что компонент `JTable`, в отличие от компонента `JList`, не является обобщенным. И для этого имеются веские основания. Если в списке предполагается наличие однотипных элементов, то в таблице зачастую находятся разнотипные элементы. В рассматриваемом здесь примере названия планет представлены символьными строками, цвет — строками типа `java.awt.Color[r=..., g=...,b=...]` и т.д.

Как ни странно, столь просто полученная таблица обладает довольно богатыми функциональными возможностями. Так, если сильно уменьшить масштаб отображения таблицы, автоматически появляется полоса прокрутки. Примечательно, что при прокрутке таблицы имена столбцов не исчезают из виду!

Если щелкнуть кнопкой мыши на одном из столбцов и перетащить его влево или вправо, весь столбец визуально отделяется от других столбцов небольшим промежутком (рис. 6.6). Перетаскиваемый столбец можно опустить на любом другом месте. Такое переупорядочение столбцов возможно *только* в представлении таблицы и никак не влияет на модель данных.

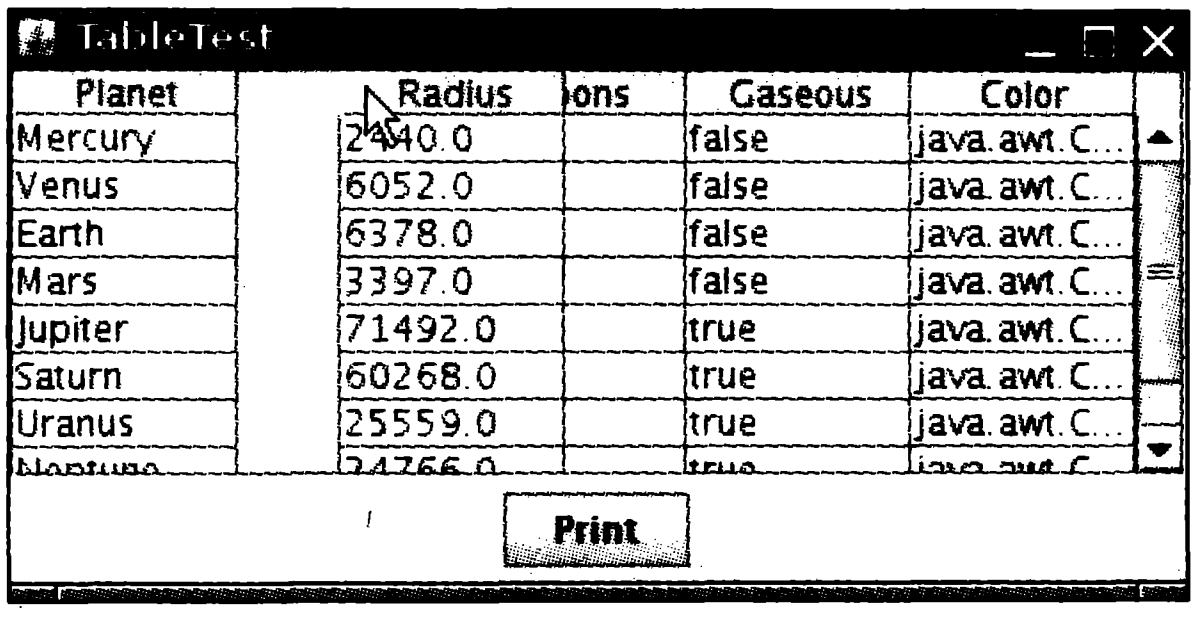


Рис. 6.6. Перемещение столбца в таблице

Для того чтобы изменить размеры столбца, наведите курсор на границу раздела двух столбцов. Как только указатель мыши примет вид двойной стрелки, перетащите границу раздела столбцов в нужное место (рис. 6.7).

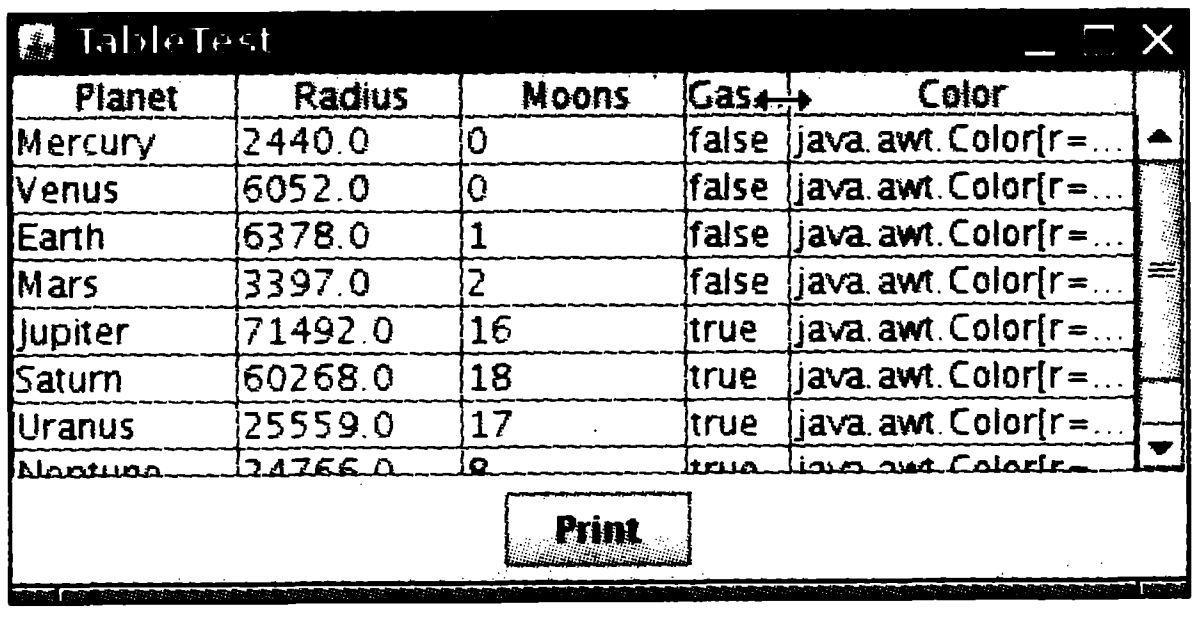


Рис. 6.7. Изменение размеров столбца в таблице

Для того чтобы выбрать строку в таблице, достаточно щелкнуть на любом месте в этой строке. Выбранные строки таблицы выделяются другим цветом. А обработка событий выбора из таблицы более подробно обсуждается далее в этом разделе. В рассматриваемом здесь примере допускается редактирование ячеек таблицы, но

внесенные в них изменения не сохраняются. Поэтому при разработке собственной прикладной программы вам придется выбирать одно из двух: вообще запретить редактирование в ячейках таблицы или организовать обработку событий редактирования и обновление модели таблицы. Подробнее об этом — несколько позже.

И наконец, если щелкнуть на заголовке столбца, строки таблицы будут автоматически отсортированы. Если щелкнуть еще раз, сортировка будет произведена в обратном порядке. Такое поведение активизируется при вызове следующего метода:

```
able.setAutoCreateRowSorter(true);
```

Для вывода таблицы на печать вызывается приведенный ниже метод. В итоге появляется диалоговое окно печати, откуда таблица выводится на печатающее устройство. Параметры вывода на печать, доступные в этом окне, рассматриваются в главе 7.

```
table.print()
```

НА ЗАМЕТКУ! Если в рассматриваемом здесь примере программы изменить фрейм из класса `TableTest` таким образом, чтобы по высоте он был выше таблицы, то под таблицей появится серая область. В отличие от компонентов `JList` и `JTree`, таблица, составляемая с помощью компонента `JTable`, не заполняет область просмотра на панели прокрутки. Поэтому это может вызвать определенные затруднения, если требуется организовать поддержку перетаскивания. (Подробнее об этом речь пойдет в главе 7.) В таком случае нужно вызвать следующий метод:

```
table.setFillsViewportHeight(true);
```

Листинг 6.5. Исходный код из файла `table/TableTest.java`

```
1  package table;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.beans.*;
6  import javax.swing.*;
7
8  /**
9   * В этой программе демонстрируется порядок отображения простой таблицы
10  * @version 1.12 2012-06-09
11  * @author Cay Horstmann
12 */
13 public class TableTest
14 {
15   public static void main(String[] args)
16   {
17     EventQueue.invokeLater(new Runnable()
18       {
19         public void run()
20         {
21           JFrame frame = new PlanetTableFrame();
22           frame.setTitle("TableTest");
23           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24           frame.setVisible(true);
25         }
26       });
27   }
28 }
29
```

```
30 /**
31  * Этот фрейм содержит таблицу с данными о планетах
32 */
33 class PlanetTableFrame extends JFrame
34 {
35   private String[] columnNames =
36       { "Planet", "Radius", "Moons", "Gaseous", "Color" };
37   private Object[][] cells =
38       { { "Mercury", 2440.0, 0, false, Color.YELLOW },
39       { "Venus", 6052.0, 0, false, Color.YELLOW },
40       { "Earth", 6378.0, 1, false, Color.BLUE },
41       { "Mars", 3397.0, 2, false, Color.RED },
42       { "Jupiter", 71492.0, 16, true, Color.ORANGE },
43       { "Saturn", 60268.0, 18, true, Color.ORANGE },
44       { "Uranus", 25559.0, 17, true, Color.BLUE },
45       { "Neptune", 24766.0, 8, true, Color.BLUE },
46       { "Pluto", 1137.0, 1, false, Color.BLACK } };
47
48   public PlanetTableFrame()
49   {
50     final JTable table = new JTable(cells, columnNames);
51     table.setAutoCreateRowSorter(true);
52     add(table, BorderLayout.CENTER);
53     JButton printButton = new JButton("Print");
54     printButton.addActionListener(EventHandler.create(
55         ActionListener.class, table, "print"));
56     JPanel buttonPanel = new JPanel();
57     buttonPanel.add(printButton);
58     add(buttonPanel, BorderLayout.SOUTH);
59     pack();
60   }
61 }
```

javax.swing.JTable 1.2

- `JTable(Object[][] entries, Object[] columnNames)`

  Составляет таблицу с моделью, используемой по умолчанию.

- `void print()` **5.0**

  Отображает диалоговое окно печати и выводит таблицу на печать.

- `boolean getAutoCreateRowSorter()` **6**
- `void setAutoCreateRowSorter(boolean newValue)` **6**

  Получают или устанавливают свойство `autoCreateRowSorter`. По умолчанию оно принимает логическое значение `false`. Если установлено именно это логическое значение, то при любом изменении модели таблицы будет автоматически устанавливаться сортировщик строк, выбираемый по умолчанию.

- `boolean getFillsViewportHeight()` **6**
- `void setFillsViewportHeight(boolean newValue)` **6**

  Получают или устанавливают свойство `fillsViewportHeight`. По умолчанию оно принимает логическое значение `false`. Если установлено именно это логическое значение, то таблица всегда будет заполнять объемлющую область просмотра.

Модели таблиц

Описанные в предыдущем разделе данные таблицы хранились в виде двухмерного массива. Но такой способ в прикладном коде обычно не применяется. Вместо вывода данных в массив для их отображения в табличном виде рекомендуется реализовать собственную модель таблицы.

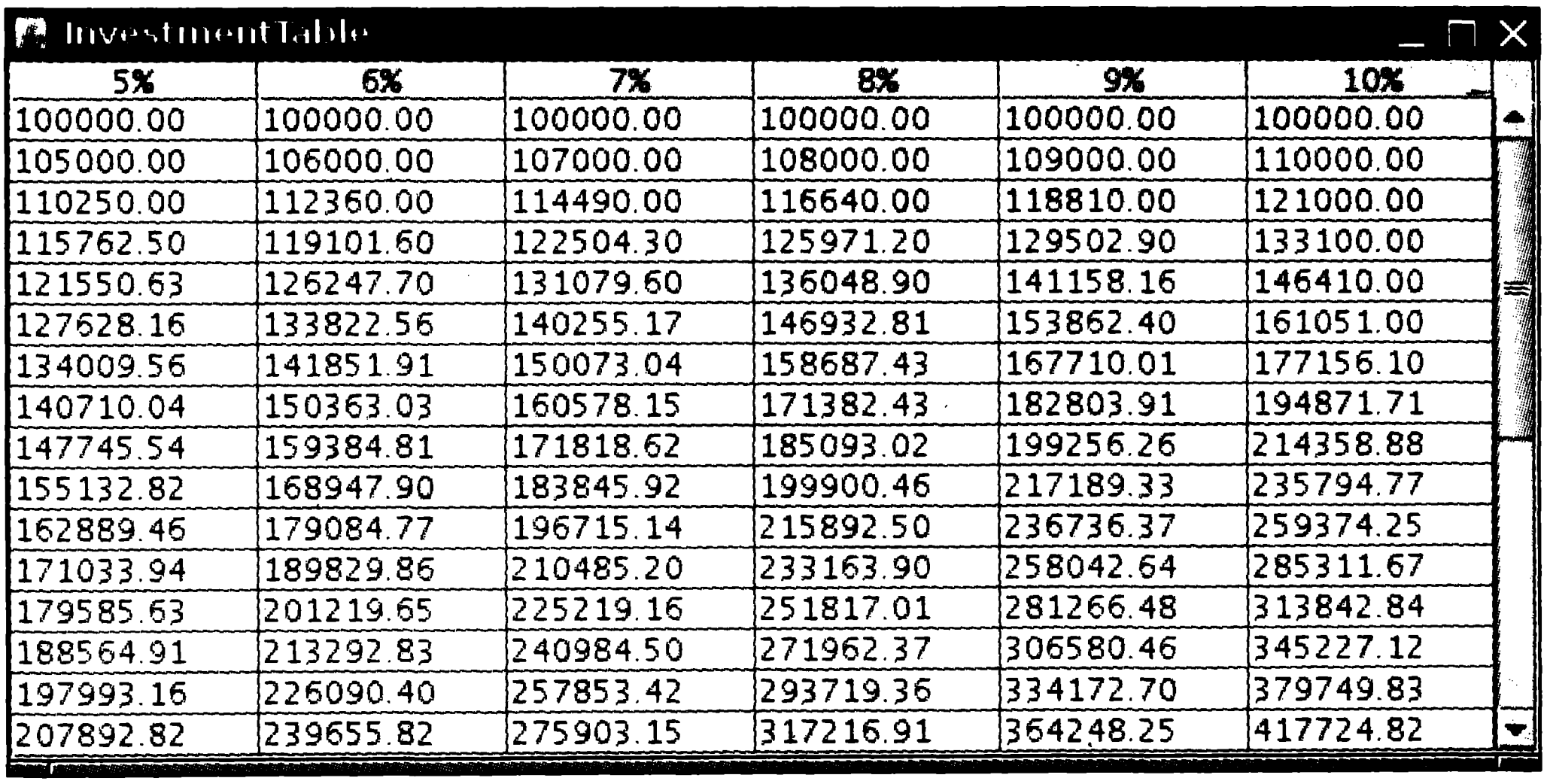
Реализовать модель таблицы совсем не трудно, поскольку для этой цели предусмотрен отдельный класс `AbstractTableModel` с большинством требующихся методов. Остается лишь предоставить следующие три метода:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

Метод `getValueAt()` можно реализовать несколькими способами. Так, если требуется отобразить набор строк типа `RowSet`, содержащий результат запроса к базе данных, то для этого подойдет следующий метод:

```
public Object getValueAt(int r, int c)
{
  try
  {
    rowSet.absolute(r + 1);
    return rowSet.getObject(c + 1);
  }
  catch (SQLException e)
  {
    e.printStackTrace();
    return null;
  }
}
```

Следующий пример программы еще проще. В ней составляется таблица из рассчитанных значений, а именно: роста инвестиций при разных учетных ставках (рис. 6.8).



5%	6%	7%	8%	9%	10%
100000.00	100000.00	100000.00	100000.00	100000.00	100000.00
105000.00	106000.00	107000.00	108000.00	109000.00	110000.00
110250.00	112360.00	114490.00	116640.00	118810.00	121000.00
115762.50	119101.60	122504.30	125971.20	129502.90	133100.00
121550.63	126247.70	131079.60	136048.90	141158.16	146410.00
127628.16	133822.56	140255.17	146932.81	153862.40	161051.00
134009.56	141851.91	150073.04	158687.43	167710.01	177156.10
140710.04	150363.03	160578.15	171382.43	182803.91	194871.71
147745.54	159384.81	171818.62	185093.02	199256.26	214358.88
155132.82	168947.90	183845.92	199900.46	217189.33	235794.77
162889.46	179084.77	196715.14	215892.50	236736.37	259374.25
171033.94	189829.86	210485.20	233163.90	258042.64	285311.67
179585.63	201219.65	225219.16	251817.01	281266.48	313842.84
188564.91	213292.83	240984.50	271962.37	306580.46	345227.12
197993.16	226090.40	257853.42	293719.36	334172.70	379749.83
207892.82	239655.82	275903.15	317216.91	364248.25	417724.82

Рис. 6.8. Пример таблицы с данными о росте инвестиций при разных учетных ставках

В данном примере метод `getValueAt()` выполняет расчет соответствующего значения и форматирует его, как показано ниже.

```
public Object getValueAt(int r, int c)
{
  double rate = (c + minRate) / 100.0;
  int nperiods = r;
  double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
  return String.format("%.2f", futureBalance);
}
```

А методы `getRowCount()` и `getColumnCount()` просто возвращают количество строк и столбцов соответственно:

```
public int getRowCount() { return years; }
public int getColumnCount() { return maxRate - minRate + 1; }
```

Если заголовки столбцов не заданы явно, то метод `getColumnName()` из класса `AbstractTableModel` по умолчанию использует для них имена `A`, `B`, `C` и т.д. Для изменения этих имен нужно переопределить метод `getColumnName()`. В данном примере в качестве заголовка столбца используется учетная ставка, как показано ниже. Весь исходный код данного примера программы приведен в листинге 6.6.

```
public String getColumnName(int c) { return (c + minRate) + "%"; }
```

Листинг 6.6. Исходный код из файла `tableModel/InvestmentTable.java`

```
1  package tableModel;
2
3  import java.awt.*;
4
5  import javax.swing.*;
6  import javax.swing.table.*;
7
8  /**
9   * В этой программе демонстрируется построение таблицы по ее модели
10  * @version 1.02 2007-08-01
11  * @author Cay Horstmann
12 */
13 public class InvestmentTable
14 {
15   public static void main(String[] args)
16   {
17      EventQueue.invokeLater(new Runnable()
18        {
19          public void run()
20          {
21            JFrame frame = new InvestmentTableFrame();
22            frame.setTitle("InvestmentTable");
23            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24            frame.setVisible(true);
25          }
26        });
27   }
28 }
29
30 /**
31  * Этот фрейм содержит таблицу капиталовложений
```

```
32 */
33 class InvestmentTableFrame extends JFrame
34 {
35   public InvestmentTableFrame()
36   {
37     TableModel model = new InvestmentTableModel(30, 5, 10);
38     JTable table = new JTable(model);
39     add(table);
40     pack();
41   }
42 }
43
44 /**
45  * В этой модели таблицы рассчитывается содержимое ячеек таблицы
46  * всякий раз, когда оно запрашивается. В таблице представлен рост
47  * капиталовложений в течение ряда лет при разных учетных ставках
48 */
49 class InvestmentTableModel extends AbstractTableModel
50 {
51   private static double INITIAL_BALANCE = 100000.0;
52
53   private int years;
54   private int minRate;
55   private int maxRate;
56
57   /**
58    * Конструирует модель таблицы капиталовложений
59    * @param y Количество лет
60    * @param r1 Наинизшая учетная ставка для составления таблицы
61    * @param r2 Наивысшая учетная ставка для составления таблицы
62   */
63   public InvestmentTableModel(int y, int r1, int r2)
64   {
65     years = y;
66     minRate = r1;
67     maxRate = r2;
68   }
69
70   public int getRowCount()
71   {
72     return years;
73   }
74
75   public int getColumnCount()
76   {
77     return maxRate - minRate + 1;
78   }
79
80   public Object getValueAt(int r, int c)
81   {
82     double rate = (c + minRate) / 100.0;
83     int nperiods = r;
84     double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
85     return String.format("%.2f", futureBalance);
86   }
87
88   public String getColumnName(int c)
89   {
```

```
90      return (c + minRate) + "%";
91    }
92 }
```

javax.swing.table.TableModel 1.2

- `int getRowCount()`
- `int getColumnCount()`

  Получают количество строк и столбцов в модели таблицы.

- `Object getValueAt(int row, int column)`

  Получает значение из указанных строки и столбца.

- `void setValueAt(Object newValue, int row, int column)`

  Устанавливает новое значение в указанных строке и столбце.

- `boolean isCellEditable(int row, int column)`

  Возвращает логическое значение `true`, если ячейка в указанных строке и столбце доступна для редактирования.

- `String getColumnName(int column)`

  Получает заголовок столбца.

Обращение со строками и столбцами таблицы

В этом разделе рассматриваются способы обращения со строками и столбцами таблицы. Таблица, составленная средствами Swing, имеет несимметричную структуру, допускающую выполнение разных операций над строками и столбцами. Дело в том, что компонент библиотеки Swing, реализующий таблицу, предназначен для отображения строк одинаковой структуры, например, данных из таблиц базы данных, а не объектов в виде произвольной двухмерной сетки. Такая асимметрия демонстрируется в данном разделе на конкретных примерах.

Классы столбцов таблицы

В следующем примере снова рассматривается таблица с данными о планетах, но теперь особое внимание уделяется типам данных в столбцах. В частности, приведенный ниже метод из модели таблицы возвращает класс с описанием типа столбца.

```
Class<?> getColumnClass(int columnIndex)
```

Эти сведения используются в классе `JTable` для выбора средства воспроизведения, подходящего для конкретного класса столбца.

В табл. 6.1 перечислены действия, выполняемые по умолчанию, для воспроизведения столбцов по их типам (и соответствующим классам). Так, в столбцах таблицы на рис. 6.9 показаны флажки и изображения. Авторы выражают искреннюю благодарность Джиму Эвинсу (Jim Evins) за любезно представленные изображения планет (`http://www.snaught.com/JimsCoolIcons/Planets`). Чтобы воспроизвести другие типы, можно установить специальное средство воспроизведения, как поясняется ниже, в разделе "Воспроизведение и редактирование ячеек таблицы".

Таблица 6.1. Действия, выполняемые по умолчанию, для воспроизведения столбцов по их типам

Тип	Как воспроизводится
`Boolean`	Флажок
`Icon`	Изображение
`Object`	Символьная строка

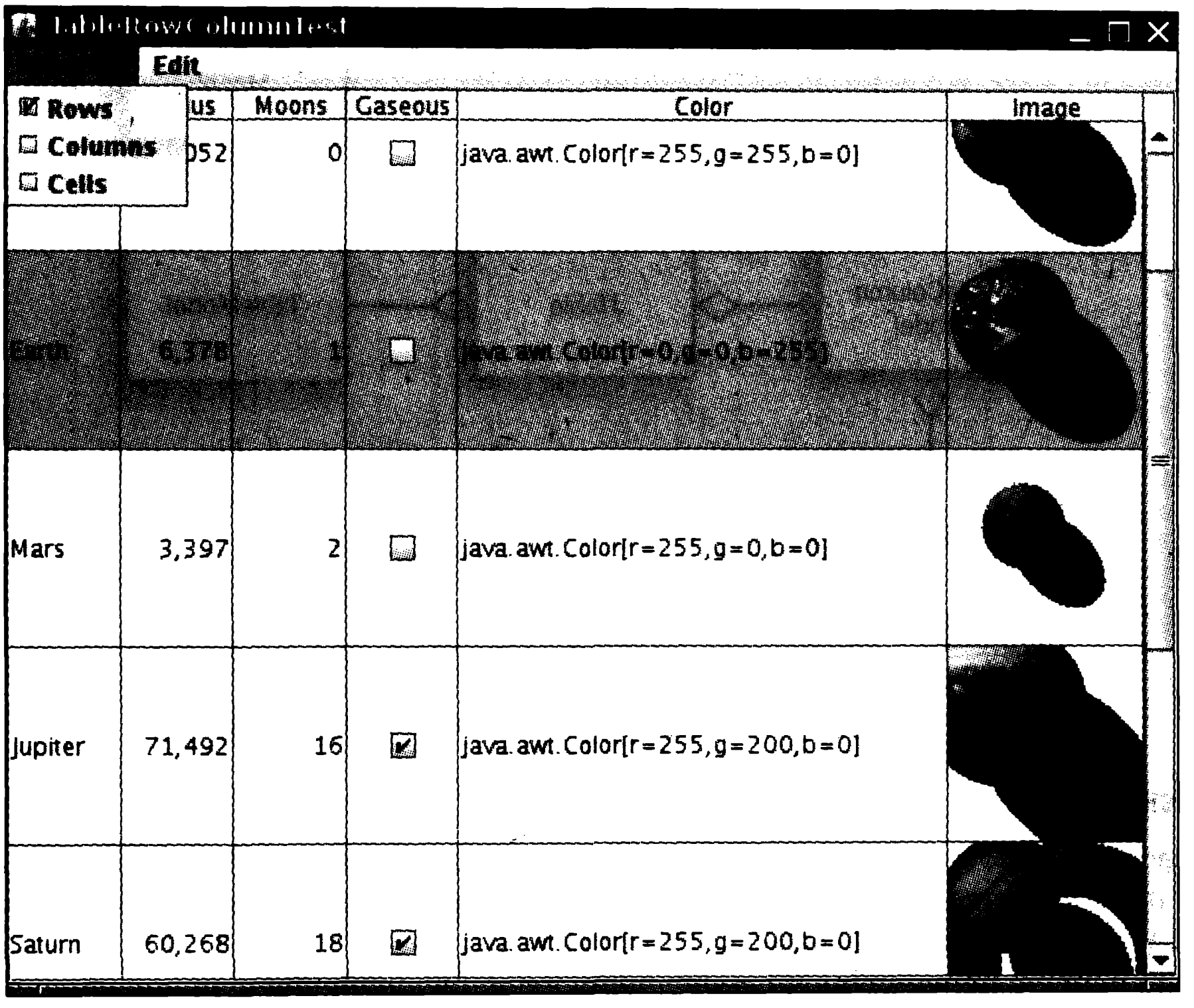


Рис. 6.9. Пример таблицы с данными о планетах, в столбцах которой воспроизводятся флажки и изображения планет

Доступ к столбцам таблицы

Компонент `JTable` сохраняет сведения обо всех столбцах таблицы в объектах типа `TableColumn`, а объект типа `TableColumnModel` манипулирует столбцами. (На рис. 6.10 схематически представлены отношения между наиболее важными классами таблиц.) Если столбцы таблицы не предполагается перемещать или вставлять динамически, то обращаться к модели столбцов таблицы (и соответствующему классу) придется нечасто. А чаще всего обращаться к этой модели требуется для получения объекта типа `TableColumn`, как показано ниже.

```
int columnIndex = . . .;
TableColumn column = table.getColumnModel().getColumn(columnIndex);
```

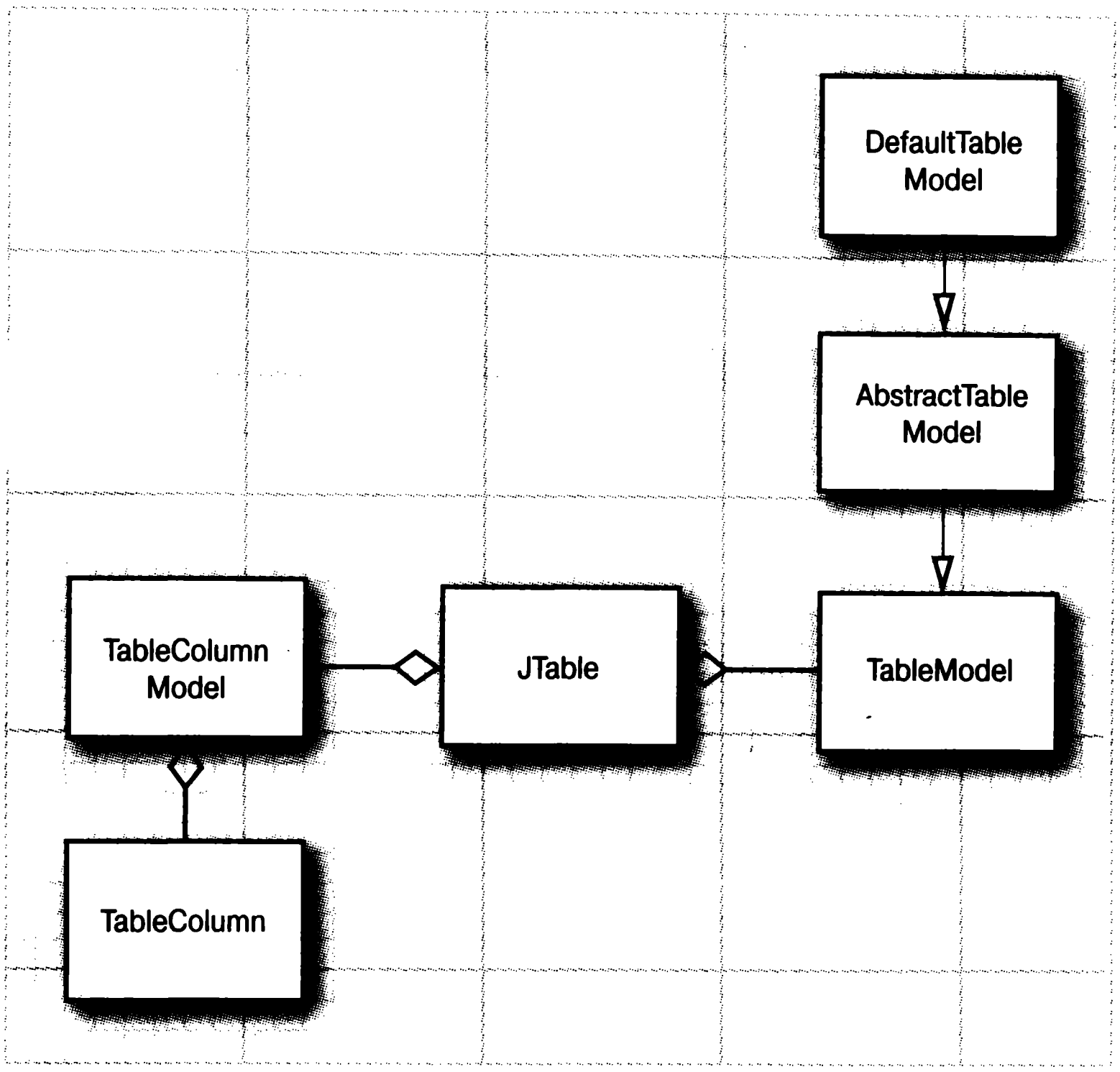


Рис. 6.10. Отношения между табличными классами

Изменение размеров столбцов таблицы

Класс `TableColumn` позволяет изменять размеры столбцов. В частности, для указания предпочтительной, минимальной и максимальной ширины столбца используются следующие методы:

```
void setPreferredWidth(int width)
void setMinWidth(int width)
void setMaxWidth(int width)
```

Полученная информация используется компонентом составления таблиц для размещения столбцов. Для того чтобы разрешить или запретить пользователю изменять размеры столбцов, достаточно вызвать метод `void setResizable (boolean resizable)`. А для того чтобы изменить размеры столбца программным путем, следует вызвать метод `void setWidth(int width)`.

Если изменяются размеры одного столбца, то по умолчанию общие размеры таблицы остаются прежними. Это, конечно, приводит к увеличению или уменьшению размеров остальных столбцов. По умолчанию подобные изменения распространяются на все столбцы, расположенные справа от того столбца, размеры которого изменяются. Благодаря этому нужные размеры для всех столбцов можно указывать слева направо.

В табл. 6.2 перечислены режимы изменения размеров таблицы при изменении размеров столбца. Их можно указать с помощью метода `void setAutoResizeMode (int mode)` из класса `JTable`.

Таблица 6.2. Режимы изменения размеров таблицы

Режим	Поведение
`AUTO_RESIZE_OFF`	Размеры остальных столбцов остаются прежними, а размеры таблицы изменяются
`AUTO_RESIZE_NEXT_COLUMN`	Изменяются только размеры следующего столбца
`AUTO_RESIZE_SUBSEQUENT_COLUMNS`	Равномерно изменяются размеры всех последующих столбцов (выбирается по умолчанию)
`UTO_RESIZE_LAST_COLUMN`	Изменяются размеры только последнего столбца
`AUTO_RESIZE_ALL_COLUMNS`	Размеры всех столбцов таблицы изменяются равномерно (это не самый лучший вариант, так как он не позволяет пользователю подгонять многие столбцы под нужные размеры)

Изменение размеров строк таблицы

Изменение высоты строк таблицы производится непосредственно в классе `JTable`. Если высота ячеек выше заданной по умолчанию, то ее придется указать явно, как показано ниже.

```
table.setRowHeight(height);
```

По умолчанию все строки таблицы имеют одинаковую высоту, но данную установку можно изменить с помощью метода `setRowHeight()` следующим образом:

```
table.setRowHeight(row, height);
```

Конкретная высота строки задается перечисленными выше методами за вычетом величины междустрочного интервала. По умолчанию величина этого интервала равна **1**, но можно указать любое другое значение следующим образом:

```
table.setRowMargin(margin);
```

Выбор строк, столбцов и ячеек таблицы

В зависимости от заданного режима пользователь может выбирать строки, столбцы и отдельные ячейки таблицы. По умолчанию допускается выбор строки, т.е. после щелчка кнопкой мыши в одной из ячеек будет выбрана вся строка, как было показано на рис. 6.9. Для отмены режима выбора строк следует вызвать метод

```
table.setRowSelectionAllowed(false)
```

В режиме выбора строк можно разрешить пользователю выбирать только одну строку, несколько смежных строк или несколько несмежных строк. Для этого нужно получить *модель выбора* и вызвать ее метод `setSelectionMode()` следующим образом:

```
table.getSelectionModel().setSelectionMode(mode);
```

В качестве параметра `mode` можно указать следующие значения:

```
ListSelectionModel.SINGLE_SELECTION
ListSelectionModel.SINGLE_INTERVAL_SELECTION
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
```

По умолчанию режим выбора столбцов отключен. Его можно включить, вызвав метод

```
table.setColumnSelectionAllowed(true)
```

Одновременное включение режимов выбора строк и столбцов равнозначно включению режима выбора ячеек (рис. 6.11). Эти режимы можно указать явно, вызвав следующий метод:

```
table.setCellSelectionEnabled(true)
```

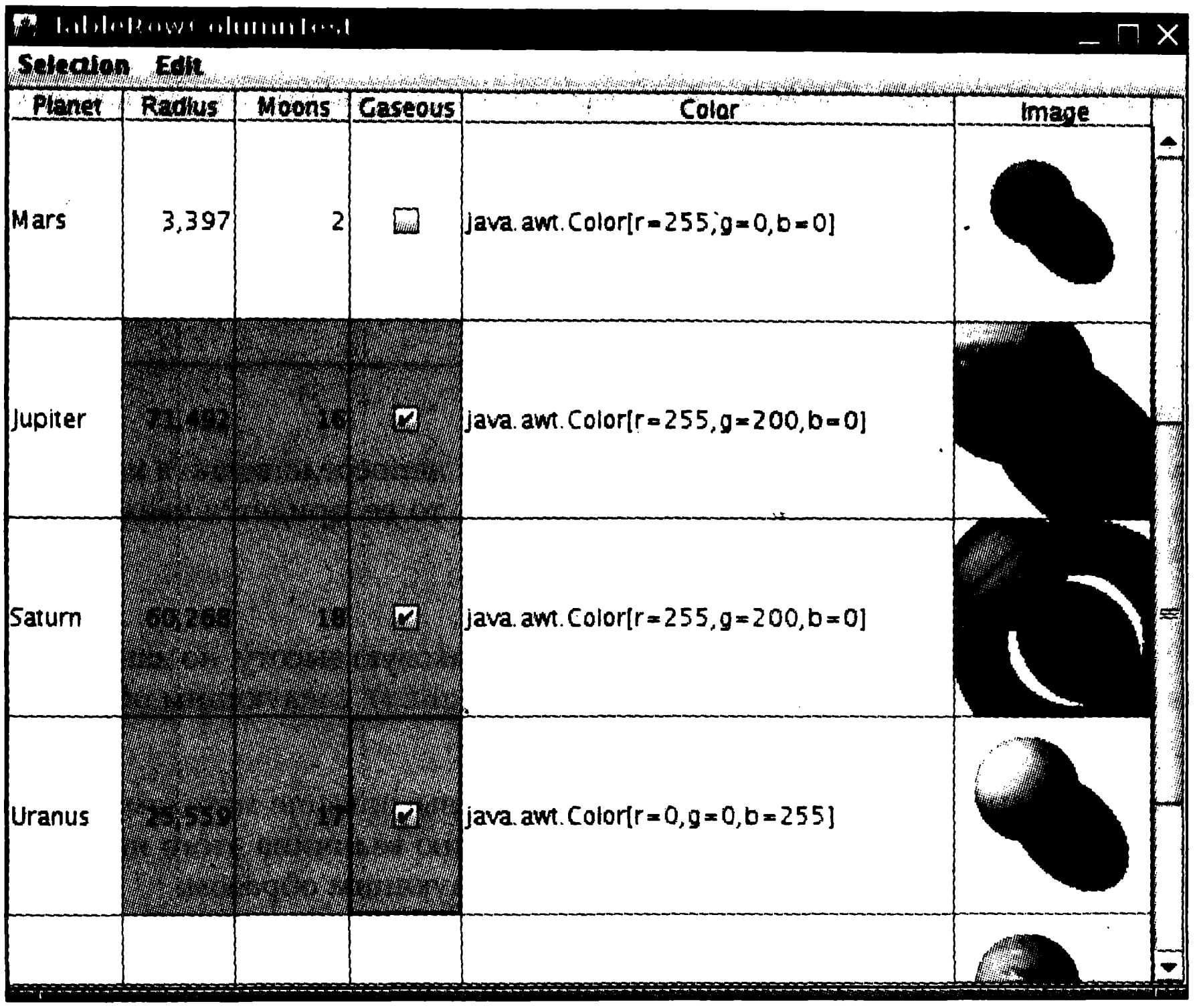


Рис. 6.11. Выбор ряда ячеек

Чтобы посмотреть, каким образом выбор ячеек происходит на практике, запустите на выполнение пример программы, исходный код которой представлен в листинге 6.7. Активизируйте режим выбора строки, столбца или ячейки из меню Selection (Выбор) и понаблюдайте за тем, как изменяется поведение таблицы в данном режиме.

С помощью методов `getSelectedRows()` и `getSelectedColumns()` можно узнать, какие именно строки и столбцы были выбраны. Оба метода возвращают массив `int[]` индексов выбранных элементов. Следует иметь в виду, что значения индексов берутся из представления таблицы, а не из базовой модели таблицы. Попытайтесь выбрать строки и столбцы, а затем перетащите столбцы в разные места и отсортируйте строки, щелкая на заголовках столбцов. Выбрав из меню пункт Print Selection (Выбор для печати), посмотрите, какие именно строки и столбцы отображаются как выбранные. Если же требуется преобразовать значения индексов таблицы в значения индексов модели таблицы, то для этой цели можно воспользоваться методами `convertRowIndexToModel()` и `convertColumnIndexToModel()` из класса `JTable`.

Сортировка строк таблицы

Как было показано в рассмотренном выше первом примере таблицы, компонент можно без особого труда дополнить функцией сортировки строк таблицы, вызвав метод `setAutoCreateRowSorter()`. Но для того чтобы получить больше возможностей для управления сортировкой, в компоненте `JTable` нужно установить и настроить объект типа `TableRowSorter<M>`. Параметр типа `M` обозначает тип модели; им должен быть подтип интерфейса `TableModel`, как показано ниже.

Некоторые столбцы должны быть исключены из сортировки (например, столбец с изображениями в таблице с данными о планетах). Для того чтобы исключить отдельные столбцы таблицы из сортировки, достаточно вызвать следующий метод:

```
sorter.setSortable(IMAGE_COLUMN, false);
```

Для каждого столбца можно также установить специальное средство сравнения. В рассматриваемом здесь примере предполагается отсортировать цвета в столбце `Color`, отдавая предпочтение синему и зеленому цвету над красным. Если щелкнуть на столбце `Color`, планеты синего цвета окажутся внизу таблицы. Такой результат сортировки по цвету достигается с помощью следующего средства сравнения:

```
sorter.setComparator(COLOR_COLUMN, new Comparator<Color>()
   {
     public int compare(Color c1, Color c2)
     {
       int d = c1.getBlue() - c2.getBlue();
       if (d != 0) return d;
       d = c1.getGreen() - c2.getGreen();
       if (d != 0) return d;
       return c1.getRed() - c2.getRed();
     }
})
```

Если не установить и не указать специальное средство сравнения, то сортировка будет произведена в следующем порядке.

1. Если класс столбца относится к типу `String`, то по умолчанию используется средство сортировки, возвращаемое в виде сортирующего объекта из метода `Collator.getInstance()`. Это средство сортирует символьные строки в соответствии с текущими региональными настройками. (Подробнее о региональных настройках и средствах сортировки см. в главе 5.)
2. Если класс столбца реализует интерфейс `Comparable`, то используется метод `compareTo()`.
3. Если в качестве средства сравнения установлен преобразователь строк таблицы типа `TableStringConverter`, то сортировку символьных строк, возвращаемых методом `toString()` этого преобразователя, необходимо выполнять с помощью средства сортировки, выбираемого по умолчанию. Для того чтобы воспользоваться именно таким способом сортировки, нужно определить преобразователь строк таблицы следующим образом:

```
sorter.setStringConverter(new TableStringConverter()
{
  public String toString(TableModel model, int row, int column)
  {
    Object value = model.getValueAt(row, column);
```

```
            преобразовать объект value в символьную строку и возвратить ее
        }
    });
```

4. В противном случае вызывается метод `toString()` со значениями в ячейках, которые сортируются с помощью средства сортировки, выбираемого по умолчанию.

Фильтрация строк таблицы

Помимо сортировки строк таблицы, класс `TableRowSorter` позволяет избирательно скрывать строки. Этот процесс называется *фильтрацией*. Чтобы активизировать режим фильтрации, следует установить соответствующий фильтр типа `RowFilter`. Например, для того чтобы отобрать все строки таблицы, содержащие хотя бы один спутник планеты, достаточно вызвать следующий метод:

```
sorter.setRowFilter(
    RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN));
```

Здесь используется предопределенный фильтр чисел. Чтобы создать такой фильтр, понадобятся следующие средства.

- Вид сравнения (одна из констант `EQUAL`, `NOT_EQUAL`, `AFTER` или `BEFORE`).
- Объект подкласса, производного от класса `Number` (например, `Integer` или `Double`). Допускаются только те объекты, которые имеют тот же класс, что и у данного объекта типа `Number`.
- Ни одного значения или несколько значений индекса столбца. Если эти значения не заданы, поиск будет производиться по всем столбцам.

Аналогичным образом в статическом методе `RowFilter.dateFilter()` создается фильтр дат. Отличие заключается лишь в том, что вместо объекта типа `Number` задается объект типа `Date`. И наконец, в статическом методе `RowFilter.regexFilter()` создается фильтр, осуществляющий поиск символьных строк, совпадающих с регулярным выражением. Например, в следующей строке кода отбираются только те планеты, название которых не оканчивается на `"s"`. (Подробнее о регулярных выражениях см. в главе 1.)

```
sorter.setRowFilter(RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN));
```

Кроме того, фильтры можно применять в различных сочетаниях с помощью методов `andFilter()`, `orFilter()` и `notFilter()`. Так, если требуется отобрать планеты, названия которых не оканчиваются на `"s"` и которые имеют как минимум один спутник, можно применить фильтры в следующем сочетании:

```
sorter.setRowFilter(RowFilter.andFilter(Arrays.asList(
    RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN),
    RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN))));
```

ВНИМАНИЕ! Обратите внимание на то, что в методах `andFilter()` и `orFilter()` употребляется не переменное число аргументов, а единственный параметр типа `Iterable`.

Чтобы реализовать собственный фильтр, необходимо предоставить объект подкласса, производного от класса `RowFilter`, и реализовать метод `include()` с целью указать те строки таблицы, которые требуется отобрать и отобразить. Сделать это нетрудно, хотя и не так просто в силу обобщенного характера класса `RowFilter`.

У класса RowFilter<M, I> имеются два параметра типа, обозначающие типы модели и идентификатора строк таблицы. При обращении с таблицами модель всегда относится к подтипу TableModel, а идентификатор — к типу Integer. (Когда-нибудь остальные компоненты будут также поддерживать фильтрацию строк, как в таблицах. Например, для отбора строк в компоненте JTree может потребоваться класс RowFilter<TreeModel, TreePath>.) Фильтр строк таблицы должен реализовывать следующий метод:

```
public boolean include(RowFilter.Entry<? extends M, ? extends I> entry)
```

В классе RowFilter предоставляются методы для получения модели, идентификатора строк таблицы и значения по заданному индексу. Таким образом, фильтрацию можно производить как по идентификатору строк таблицы, так и по их содержимому. Например, с помощью следующего фильтра отображается каждая вторая строка таблицы:

```
RowFilter<TableModel, Integer> filter =
        new RowFilter<TableModel, Integer>()
{
  public boolean include(
        Entry<? extends TableModel, ? extends Integer> entry)
  {
    return entry.getIdentifier() % 2 == 0;
  }
};
```

Если же требуется отобрать только те планеты, которые содержат четное количество спутников, то вместо приведенного выше фильтра можно попытаться применить следующий фильтр:

```
((Integer) entry.getValue(MOONS_COLUMN)) % 2 == 0
```

В рассматриваемом здесь примере программы пользователю разрешается скрывать произвольные строки таблицы. Индексы скрытых строк сохраняются в наборе строк. А в фильтр строк включаются все строки таблицы, индекс которых отсутствует в данном наборе.

Такой механизм фильтрации не предназначен для применения фильтров, критерии которых изменяются с течением времени. В рассматриваемом здесь примере программы вызов приведенного ниже метода повторяется всякий раз, когда изменяется набор скрытых строк таблицы. Как только фильтр установлен, он сразу же применяется.

```
sorter.setRowFilter(filter);
```

Сокрытие и показ столбцов таблицы

Как было показано в предыдущем подразделе, строки таблицы можно отфильтровывать по их содержимому или же их идентификатору. А для того чтобы скрыть столбцы, понадобится совершенно другой механизм.

Метод removeColumn() из класса JTable позволяет удалить столбец, определяемый параметром TableColumn, из представления таблицы, т.е. скрыть его от пользователя, оставив в составе модели. Ниже показано, каким образом конкретный объект, описывающий столбец таблицы, извлекается из модели таблицы по известному номеру, получаемому, например, с помощью метода getSelectedColumns().

```
TableColumnModel columnModel = table.getColumnModel();
TableColumn column = columnModel.getColumn(i);
table.removeColumn(column);
```

Если запомнить этот объект, то впоследствии его можно ввести обратно в модель таблицы следующим образом:

```
table.addColumn(column);
```

Этот метод добавляет столбец в конец таблицы. Если же столбец требуется расположить в каком-нибудь другом месте, то для его перемещения на это место следует вызвать метод `moveColumn()`.

Кроме того, создав объект типа `TableColumn`, можно сформировать новый столбец, который соответствует индексу столбца в модели таблицы, как показано ниже. Таким образом, в таблице можно создать несколько столбцов, которые будут представлять один и тот же столбец в модели.

```
table.addColumn(new TableColumn(modelColumnIndex));
```

В рассматриваемом здесь примере программы демонстрируется выбор и фильтрация строк и столбцов таблицы. Исходный код этой программы представлен в листинге 6.7.

Листинг 6.7. Исходный код из файла `tableRowColumn/PlanetTableFrame.java`

```
1   package tableRowColumn;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.util.*;
6   import javax.swing.*;
7   import javax.swing.table.*;
8
9   /**
10   * Этот фрейм содержит таблицы с данными о планетах
11  */
12  public class PlanetTableFrame extends JFrame
13  {
14    private static final int DEFAULT_WIDTH = 600;
15    private static final int DEFAULT_HEIGHT = 500;
16
17    public static final int COLOR_COLUMN = 4;
18    public static final int IMAGE_COLUMN = 5;
19
20    private JTable table;
21    private HashSet<Integer> removedRowIndices;
22    private ArrayList<TableColumn> removedColumns;
23    private JCheckBoxMenuItem rowsItem;
24    private JCheckBoxMenuItem columnsItem;
25    private JCheckBoxMenuItem cellsItem;
26
27    private String[] columnNames =
28       { "Planet", "Radius", "Moons", "Gaseous", "Color", "Image" };
29
30    private Object[][] cells = {
31       { "Mercury", 2440.0, 0, false, Color.YELLOW,
32         new ImageIcon(getClass().getResource("Mercury.gif")) },
33       { "Venus", 6052.0, 0, false, Color.YELLOW,
```

```
34          new ImageIcon(getClass().getResource("Venus.gif")) },
35        { "Earth", 6378.0, 1, false, Color.BLUE,
36          new ImageIcon(getClass().getResource("Earth.gif")) },
37        { "Mars", 3397.0, 2, false, Color.RED,
38          new ImageIcon(getClass().getResource("Mars.gif")) },
39        { "Jupiter", 71492.0, 16, true, Color.ORANGE,
40          new ImageIcon(getClass().getResource("Jupiter.gif")) },
41        { "Saturn", 60268.0, 18, true, Color.ORANGE,
42          new ImageIcon(getClass().getResource("Saturn.gif")) },
43        { "Uranus", 25559.0, 17, true, Color.BLUE,
44          new ImageIcon(getClass().getResource("Uranus.gif")) },
45        { "Neptune", 24766.0, 8, true, Color.BLUE,
46          new ImageIcon(getClass().getResource("Neptune.gif")) },
47        { "Pluto", 1137.0, 1, false, Color.BLACK,
48          new ImageIcon(getClass().getResource("Pluto.gif")) } };
49
50    public PlanetTableFrame()
51    {
52      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
53
54      TableModel model = new DefaultTableModel(cells, columnNames)
55        {
56          public Class<?> getColumnClass(int c)
57          {
58            return cells[0][c].getClass();
59          }
60        };
61
62      table = new JTable(model);
63
64      table.setRowHeight(100);
65      table.getColumnModel().getColumn(COLOR_COLUMN).setMinWidth(250);
66      table.getColumnModel().getColumn(IMAGE_COLUMN).setMinWidth(100);
67
68      final TableRowSorter<TableModel> sorter =
69          new TableRowSorter<>(model);
70      table.setRowSorter(sorter);
71      sorter.setComparator(COLOR_COLUMN, new Comparator<Color>()
72        {
73          public int compare(Color c1, Color c2)
74          {
75            int d = c1.getBlue() - c2.getBlue();
76            if (d != 0) return d;
77            d = c1.getGreen() - c2.getGreen();
78            if (d != 0) return d;
79            return c1.getRed() - c2.getRed();
80          }
81        });
82      sorter.setSortable(IMAGE_COLUMN, false);
83      add(new JScrollPane(table), BorderLayout.CENTER);
84
85      removedRowIndices = new HashSet<>();
86      removedColumns = new ArrayList<>();
87
88      final RowFilter<TableModel, Integer> filter =
89          new RowFilter<TableModel, Integer>()
90      {
91        public boolean include(
92            Entry<? extends TableModel, ? extends Integer> entry)
```

```
93          {
94            return !removedRowIndices.contains(entry.getIdentifier());
95          }
96        };
97
98        // создать меню
99
100       JMenuBar menuBar = new JMenuBar();
101       setJMenuBar(menuBar);
102
103       JMenu selectionMenu = new JMenu("Selection");
104       menuBar.add(selectionMenu);
105
106       rowsItem = new JCheckBoxMenuItem("Rows");
107       columnsItem = new JCheckBoxMenuItem("Columns");
108       cellsItem = new JCheckBoxMenuItem("Cells");
109
110       rowsItem.setSelected(table.getRowSelectionAllowed());
111       columnsItem.setSelected(table.getColumnSelectionAllowed());
112       cellsItem.setSelected(table.getCellSelectionEnabled());
113
114       rowsItem.addActionListener(new ActionListener()
115         {
116           public void actionPerformed(ActionEvent event)
117           {
118             table.clearSelection();
119             table.setRowSelectionAllowed(rowsItem.isSelected());
120             updateCheckboxMenuItems();
121           }
122         });
123       selectionMenu.add(rowsItem);
124
125       columnsItem.addActionListener(new ActionListener()
126         {
127           public void actionPerformed(ActionEvent event)
128           {
129             table.clearSelection();
130             table.setColumnSelectionAllowed(columnsItem.isSelected());
131             updateCheckboxMenuItems();
132           }
133         });
134       selectionMenu.add(columnsItem);
135
136       cellsItem.addActionListener(new ActionListener()
137         {
138           public void actionPerformed(ActionEvent event)
139           {
140             table.clearSelection();
141             table.setCellSelectionEnabled(cellsItem.isSelected());
142             updateCheckboxMenuItems();
143           }
144         });
145       selectionMenu.add(cellsItem);
146
147       JMenu tableMenu = new JMenu("Edit");
148       menuBar.add(tableMenu);
149
150       JMenuItem hideColumnsItem = new JMenuItem("Hide Columns");
151       hideColumnsItem.addActionListener(new ActionListener()
```

```
152         {
153           public void actionPerformed(ActionEvent event)
154           {
155             int[] selected = table.getSelectedColumns();
156             TableColumnModel columnModel = table.getColumnModel();
157
158             // удалить столбцы из представления таблицы, начиная с
159             // последнего индекса, но не затрагивая номера столбцов
160
161             for (int i = selected.length - 1; i >= 0; i--)
162             {
163               TableColumn column = columnModel.getColumn(selected[i]);
164               table.removeColumn(column);
165
166               // сохранить удаленные столбцы для последующего отображения
167
168               removedColumns.add(column);
169             }
170           }
171         });
172       tableMenu.add(hideColumnsItem);
173
174       JMenuItem showColumnsItem = new JMenuItem("Show Columns");
175       showColumnsItem.addActionListener(new ActionListener()
176         {
177           public void actionPerformed(ActionEvent event)
178           {
179             // восстановить все удаленные столбцы
180             for (TableColumn tc : removedColumns)
181             table.addColumn(tc);
182             removedColumns.clear();
183           }
184         });
185       tableMenu.add(showColumnsItem);
186
187       JMenuItem hideRowsItem = new JMenuItem("Hide Rows");
188
189       hideRowsItem.addActionListener(new ActionListener()
190         {
191           public void actionPerformed(ActionEvent event)
192           {
193             int[] selected = table.getSelectedRows();
194             for (int i : selected)
195               removedRowIndices.add(table.convertRowIndexToModel(i));
196             sorter.setRowFilter(filter);
197           }
198         });
199       tableMenu.add(hideRowsItem);
200
201       JMenuItem showRowsItem = new JMenuItem("Show Rows");
202       showRowsItem.addActionListener(new ActionListener()
203         {
204           public void actionPerformed(ActionEvent event)
205           {
206             removedRowIndices.clear();
207             sorter.setRowFilter(filter);
208           }
209         });
210       tableMenu.add(showRowsItem);
```

```
211
212     JMenuItem printSelectionItem = new JMenuItem("Print Selection");
213     printSelectionItem.addActionListener(new ActionListener()
214       {
215         public void actionPerformed(ActionEvent event)
216         {
217           int[] selected = table.getSelectedRows();
218           System.out.println(
219               "Selected rows: " + Arrays.toString(selected));
220           selected = table.getSelectedColumns();
221           System.out.println(
222               "Selected columns: " + Arrays.toString(selected));
223         }
224       });
225     tableMenu.add(printSelectionItem);
226   }
227   private void updateCheckboxMenuItems()
228   {
229     rowsItem.setSelected(table.getRowSelectionAllowed());
230     columnsItem.setSelected(table.getColumnSelectionAllowed());
231     cellsItem.setSelected(table.getCellSelectionEnabled());
232   }
233 }
```

javax.swing.table.TableModel 1.2

- `Class getColumnClass(int columnIndex)`

  Возвращает класс для определения типа значений в указанном столбце. Эти сведения используются для сортировки и воспроизведения значений в указанном столбце.

javax.swing.JTable 1.2

- `TableColumnModel getColumnModel()`

  Получает модель столбца, описывающую расположение столбцов в таблице.

- `void setAutoResizeMode(int mode)`

  Устанавливает режим автоматического изменения размеров столбцов таблицы.

  Параметры: *mode* Одно из значений следующих констант:

  AUTO_RESIZE_OFF,

  AUTO_RESIZE_NEXT_COLUMN,

  AUTO_RESIZE_SUBSEQUENT_COLUMNS,

  AUTO_RESIZE_LAST_COLUMN,

  AUTO_RESIZE_ALL_COLUMNS

- `int getRowMargin()`
- `void setRowMargin(int margin)`

  Получают или устанавливают ширину интервала между ячейками смежных строк таблицы.

- `int getRowHeight()`
- `void setRowHeight(int height)`

  Получают или устанавливают высоту всех строк таблицы по умолчанию.
- `int getRowHeight(int row)`
- `void setRowHeight(int row, int height)`

  Получают или устанавливают высоту указанной строки таблицы.
- `ListSelectionModel getSelectionModel()`

  Возвращает модель выбора списка. Эта модель требуется для того, чтобы сделать выбор между строкой, столбцом и ячейкой.
- `boolean getRowSelectionAllowed()`
- `void setRowSelectionAllowed(boolean b)`

  Получают или устанавливают свойство `rowSelectionAllowed`. Если оно принимает логическое значение `true`, то строки выбираются из таблицы, если щелкнуть на ее ячейках.
- `boolean getColumnSelectionAllowed()`
- `void setColumnSelectionAllowed(boolean b)`

  Получают или устанавливают свойство `columnSelectionAllowed`. Если оно принимает логическое значение `true`, то столбцы выбираются из таблицы, если щелкнуть на ее ячейках.
- `boolean getCellSelectionEnabled()`

  Возвращает логическое значение `true`, если оба свойства, `rowSelectionAllowed` и `columnSelectionAllowed`, принимают логическое значение `true`.
- `void setCellSelectionEnabled(boolean b)`

  Присваивает обоим свойствам, `rowSelectionAllowed` и `columnSelectionAllowed`, значение параметра *b*.
- `void addColumn(TableColumn column)`

  Добавляет столбец, становящийся последним в представлении таблицы.
- `void moveColumn(int from, int to)`

  Перемещает столбец, находящийся в таблице по индексу `from`, на новое место по индексу *to*. Эта операция затрагивает только представление таблицы.
- `void removeColumn(TableColumn column)`

  Удаляет указанный столбец из представления таблицы.
- `int convertRowIndexToModel(int index)` **6**
- `int convertColumnIndexToModel(int index)`

  Возвращают индекс строки или столбца в модели по указанному индексу. Значение возвращаемого индекса отличается от значения параметра *index*, если производится сортировка или фильтрация строк таблицы или же если столбцы перемещаются или удаляются из таблицы.
- `void setRowSorter(RowSorter<? extends TableModel> sorter)`

  Устанавливает средство сортировки строк таблицы.

javax.swing.table.TableColumnModel 1.2

- `TableColumn getColumn(int index)`

  Возвращает объект, описывающий столбец по указанному индексу в представлении таблицы.

javax.swing.table.TableColumn 1.2

- `TableColumn(int modelColumnIndex)`

  Создает столбец таблицы для представления столбца в модели таблицы по указанному индексу.

- `void setPreferredWidth(int width)`
- `void setMinWidth(int width)`
- `void setMaxWidth(int width)`

  Задают предпочтительную, минимальную или максимальную ширину столбца равной значению параметра *width*.

- `void setWidth(int width)`

  Задает конкретную ширину столбца равной значению параметра *width*.

- `void setResizable(boolean b)`

  Если параметр *b* принимает логическое значение `true`, допускается изменять размеры столбца.

javax.swing.ListSelectionModel 1.2

- `void setSelectionMode(int mode)`

  Задает режим выбора.

  Параметры: *mode* Одно из значений следующих констант: `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION` или `MULTIPLE_INTERVAL_SELECTION`

javax.swing.DefaultRowSorter<M, I> 6

- `void setComparator(int column, Comparator<?> comparator)`

  Устанавливает средство для сравнения со значением в указанном столбце.

- `void setSortable(int column, boolean enabled)`

  Разрешает или запрещает сортировку для указанного столбца.

- `void setRowFilter(RowFilter<? super M,? super I> filter)`

  Устанавливает фильтр строк таблицы.

javax.swing.table.TableRowSorter<M extends TableModel> 6

- `void setStringConverter(TableStringConverter` *stringConverter*`)`

  Устанавливает преобразователь для сортировки и фильтрации строк таблицы.

javax.swing.table.TableStringConverter<M extends TableModel> 6

- `abstract String toString(TableModel` *model*`, int` *row*`, int` *column*`)`

  Преобразует в символьную строку значение из указанного места в модели таблицы. Этот метод можно переопределить.

javax.swing.RowFilter<M, I> 6

- `boolean include(RowFilter.Entry<? extends M,? extends I>` *entry*`)`

  Определяет строки таблицы, которые остались после фильтрации. Этот метод можно переопределить.

- `static <M,I> RowFilter<M,I> numberFilter(RowFilter.ComparisonType` *type*`, Number` *number*`, int...`*indices*`)`
- `static <M,I> RowFilter<M,I> dateFilter(RowFilter.ComparisonType` *type*`, Date` *date*`, int...` *indices*`)`

  Возвращают фильтр, включающий строки, которые содержат значения, совпадающие со сравниваемым числом или датой. В качестве вида сравнения можно указать значение одной из следующих констант: `EQUAL`, `NOT_EQUAL`, `AFTER` или `BEFORE`. Если заданы любые индексы столбцов в модели, то поиск будет производиться только в этих столбцах, а иначе — во всех столбцах. Для фильтрации чисел класс, определяющий тип значений в ячейках таблицы, должен совпадать с классом параметра `number`.

- `static <M,I> RowFilter<M,I> regexFilter(String` *regex*`, int...` *indices*`)`

  Возвращает фильтр, включающий строки, которые содержат значения, совпадающие со сравниваемым регулярным выражением. Если заданы любые индексы столбцов в модели, то поиск будет производиться только в этих столбцах, а иначе — во всех столбцах. Следует иметь в виду, что метод `getStringValue()` из класса `RowFilter.Entry` возвращает совпавшую символьную строку.

- `static <M,I> RowFilter<M,I> andFilter(Iterable<? extends RowFilter<? super M,? super I>>` *filters*`)`
- `static <M,I> RowFilter<M,I> orFilter(Iterable<? extends RowFilter<? super M,? super I>>` *filters*`)`

  Возвращают фильтр, включающий записи, входящие во все фильтры или хотя бы в один из фильтров.

- `static <M,I> RowFilter<M,I> notFilter(RowFilter<M,I>` *filter*`)`

  Возвращает фильтр, включающий записи, не входящие в указанный фильтр.

javax.swing.RowFilter.Entry<M, I> 6

- `I getIdentifier()`

  Возвращает идентификатор данной записи в строке таблицы.

- `M getModel()`

  Возвращает модель данной записи в строке таблицы.

- `Object getValue(int index)`

  Возвращает значение, хранящееся по указанному индексу в данной строке таблицы.

- `int getValueCount()`

  Возвращает количество значений, хранящихся в данной строке таблицы.

- `String getStringValue(int index)`

  Возвращает значение, хранящееся по указанному индексу в данной строке таблицы и преобразованное в символьную строку. Средство сортировки строк типа `TableRowSorter` создает записи, для которых в данном методе вызывается преобразователь отсортированных результатов в символьные строки.

Воспроизведение и редактирование ячеек таблицы

Как было показано ранее в подразделе "Доступ к столбцам таблицы", тип столбца определяет способ воспроизведения ячеек таблицы. По умолчанию для типов `Boolean` и `Icon` предоставляются средства воспроизведения флажков или изображений. А для всех остальных типов приходится самостоятельно устанавливать специальное средство воспроизведения.

Средства воспроизведения ячеек в таблице подобны упоминавшимся ранее средствам воспроизведения ячеек в списке. Они реализуют интерфейс `TableCellRenderer` с приведенным ниже единственным методом. Этот метод вызывается всякий раз, когда требуется перерисовка таблицы. Он возвращает компонент, метод `paint()` которого служит для отображения содержимого ячейки.

```
Component getTableCellRendererComponent(JTable table, Object value,
    boolean isSelected, boolean hasFocus, int row, int column)
```

В таблице, представленной на рис. 6.12, содержатся ячейки типа `Color`. Средство воспроизведения просто возвращает панель с цветом фона в виде объекта цвета, хранящегося в данной ячейке. Требующийся цвет передается в качестве параметра `value` методу, устанавливающему цвет фона, как показано ниже.

```
class ColorTableCellRenderer extends JPanel implements TableCellRenderer
{
  public Component getTableCellRendererComponent(
        JTable table, Object value, boolean isSelected,
        boolean hasFocus, int row, int column)
  {
    setBackground((Color) value);
    if (hasFocus)
      setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
    else
      setBorder(null);
```

```
        return this;
    }
}
```

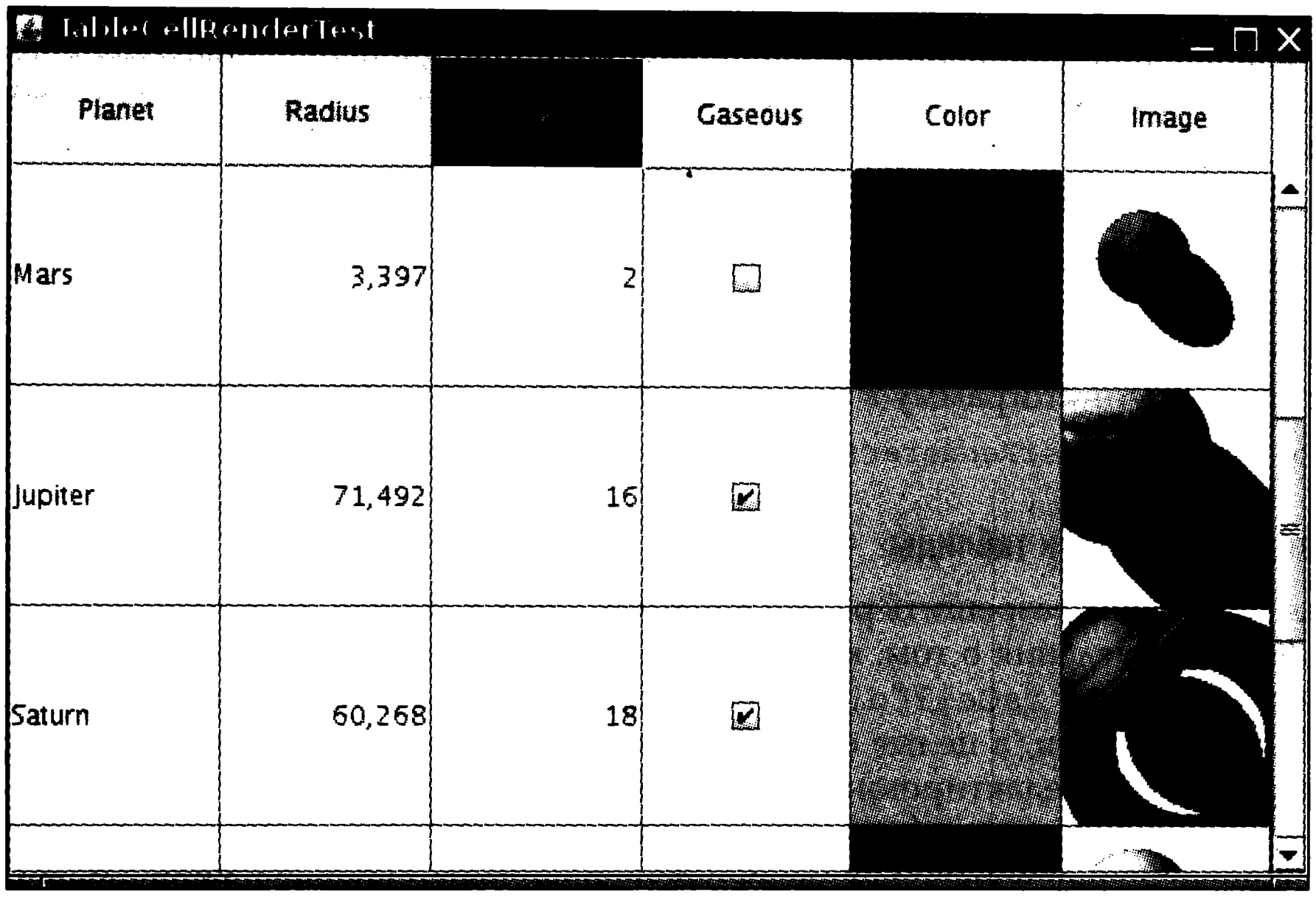


Рис. 6.12. Таблица со средствами воспроизведения ячеек

Нетрудно заметить, что при получении ячейкой фокуса ввода средство воспроизведения устанавливает рамку. (Для установки нужной рамки запрашивается объект типа `UIManager`. А для выбора подходящего ключа поиска пришлось обратиться к исходному коду класса `DefaultTableCellRenderer`.)

Как правило, для указания на то, что текущая ячейка таблицы выбрана, устанавливается походящий цвет фона. Здесь трудно посоветовать что-нибудь конкретное, поскольку не исключено, что фон будет плохо сочетаться с остальными цветами. Способ обозначения выбранной ячейки таблицы можно, в частности, позаимствовать из рассмотренного ранее примера программы `ListRenderingTest` из листинга 6.4.

СОВЕТ. Если конкретное средство воспроизведения должно просто выводить текстовую строку или пиктограмму, его можно построить в виде подкласса, расширяющего класс `DefaultTableCellRenderer`. Тогда средства, реализованные в суперклассе, возьмут на себя ответственность за воспроизведение ячейки в состоянии выбора или обладания фокусом ввода.

Таблице необходимо каким-то образом указать, что для воспроизведения всех объектов типа `Color` следует использовать данное средство. Для этого предусмотрен метод `setDefaultRenderer()` из класса `JTable`. Ему передаются объект типа `Class` и требующееся средство воспроизведения следующим образом:

```
table.setDefaultRenderer(Color.class, new ColorTableCellRenderer());
```

В итоге указанное средство будет использоваться для воспроизведения всех объектов данного типа. Если средство воспроизведения требуется выбрать

по какому-нибудь другому критерию, для этого придется создать подкласс, производный от класса `JTable`, а также переопределить метод `getCellRenderer()`.

Воспроизведение заголовков

Чтобы воспроизвести пиктограмму или иное изображение в заголовке столбца таблицы, необходимо установить соответствующее значение для этого заголовка следующим образом:

```
moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
```

Но ведь заголовок столбца таблицы не настолько логически развит, чтобы самостоятельно выбирать подходящее средство воспроизведения по указанному значению. Поэтому данное средство придется установить вручную. Например, для того чтобы показать пиктограмму в заголовке столбца, нужно вызвать следующий метод:

```
moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIcon.class));
```

Редактирование ячеек таблицы

Для того чтобы разрешить редактирование ячеек таблицы, нужно получить из модели таблицы сведения о том, какие именно ячейки можно редактировать. Для этой цели служит метод `isCellEditable()`. Чаще всего редактируемыми объявляются не отдельные ячейки, а целые столбцы. Например, в приведенном ниже фрагменте кода разрешается редактирование данных в четырех столбцах.

```
public boolean isCellEditable(int r, int c)
{
  return c == PLANET_COLUMN || c == MOONS_COLUMN ||
         c == GASEOUS_COLUMN || c ==   COLOR_COLUMN;
}
```

НА ЗАМЕТКУ! В классе `AbstractTableModel` имеется метод `isCellEditable()`, возвращающий логическое значение `false`, а в классе `DefaultTableModel` этот метод переопределен и по умолчанию возвращает логическое значение `true`.

В примере программы, исходный код которой представлен в листингах 6.8–6.11, можно устанавливать и сбрасывать флажки непосредственно в ячейках столбца `Gaseous` таблицы с данными о планетах. В этой программе можно также выбирать нужное количество спутников планет из комбинированных списков в ячейках столбца `Moons`, как показано на рис. 6.13. Несколько ниже будет показано, как организовать редактор ячейки таблицы на основании комбинированного списка. И, наконец, если щелкнуть на любой ячейке в первом столбце `Planet` рассматриваемой здесь таблицы, то эта ячейка получит фокус ввода, а следовательно, в ней можно ввести или отредактировать названия планеты.

В данном примере программы демонстрируется применение трех вариантов класса `DefaultCellEditor` для редактирования ячеек типа `JTextField`, `JCheckBox` и `JComboBox`. Класс `JTable` автоматически устанавливает редактор флажков для ячеек типа `Boolean`, а также редактор текстовых полей для всех редактируемых ячеек, у которых отсутствует собственное средство воспроизведения. В текстовых полях имеется возможность редактировать символьные строки, которые получаются в результате вызова метода `toString()` со значением, возвращаемым методом `getValueAt()` из модели таблицы.

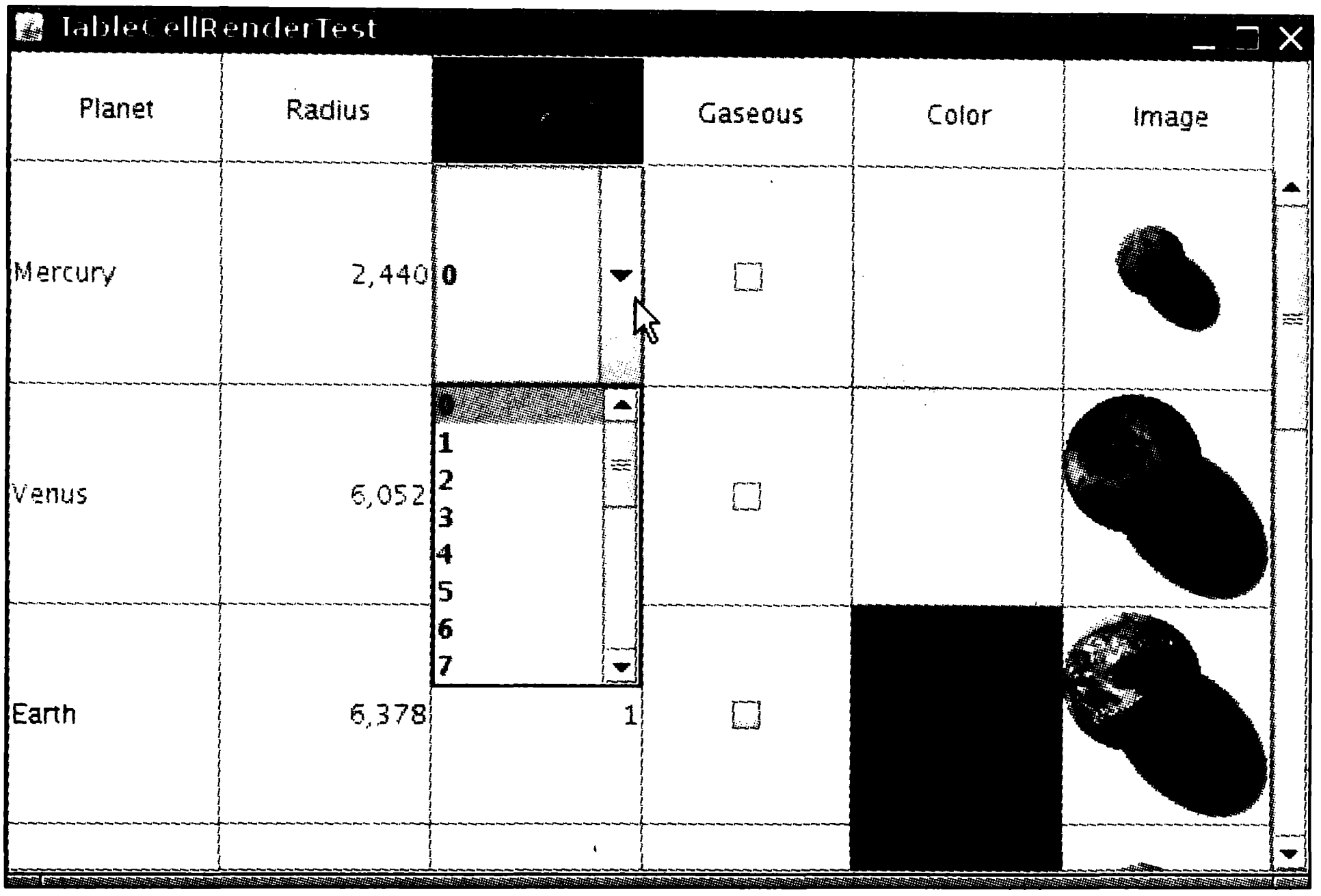


Рис. 6.13. Редактирование ячеек таблицы

По завершении редактирования в результате вызова метода `getCellEditorValue()` отредактированное значение извлекается из соответствующего редактора. Этот метод должен возвратить значение правильного типа (т.е. того типа, который возвращается методом `getColumnType()` из модели).

Редактор ячейки таблицы на основании комбинированного списка придется создать вручную, потому что компоненту `JTable` неизвестно, какие именно значения могут подойти для отдельного вида ячейки. Так, в столбце `Moons` требуется организовать выбор любого значения в пределах от **0** до **20**. Ниже приведен соответствующий код для инициализации этими значениями комбинированного списка.

```
JComboBox moonCombo = new JComboBox();
for (int i = 0; i <= 20; i++)
  moonCombo.addItem(i);
```

Чтобы создать объект класса `DefaultCellEditor` для такого типа данных, нужно передать комбинированный список конструктору этого класса следующим образом:

```
TableCellEditor moonEditor = new DefaultCellEditor(moonCombo);
```

Затем следует установить редактор, который, в отличие от средства воспроизведения цвета в столбце `Color`, не должен зависеть от *типа* объекта. Его совсем не обязательно использовать для всех объектов типа `Integer`, а достаточно установить только в отдельном столбце следующим образом:

```
moonColumn.setCellEditor(moonEditor);
```

Специальные редакторы

Запустите еще раз на выполнение рассматриваемую здесь программу и щелкните кнопкой мыши на ячейке столбца `Color`. На экране появится диалоговое окно *селектора цвета* для данной планеты. Выберите нужный цвет и щелкните на кнопке OK. В итоге цвет ячейки изменится на выбранный (рис. 6.14).

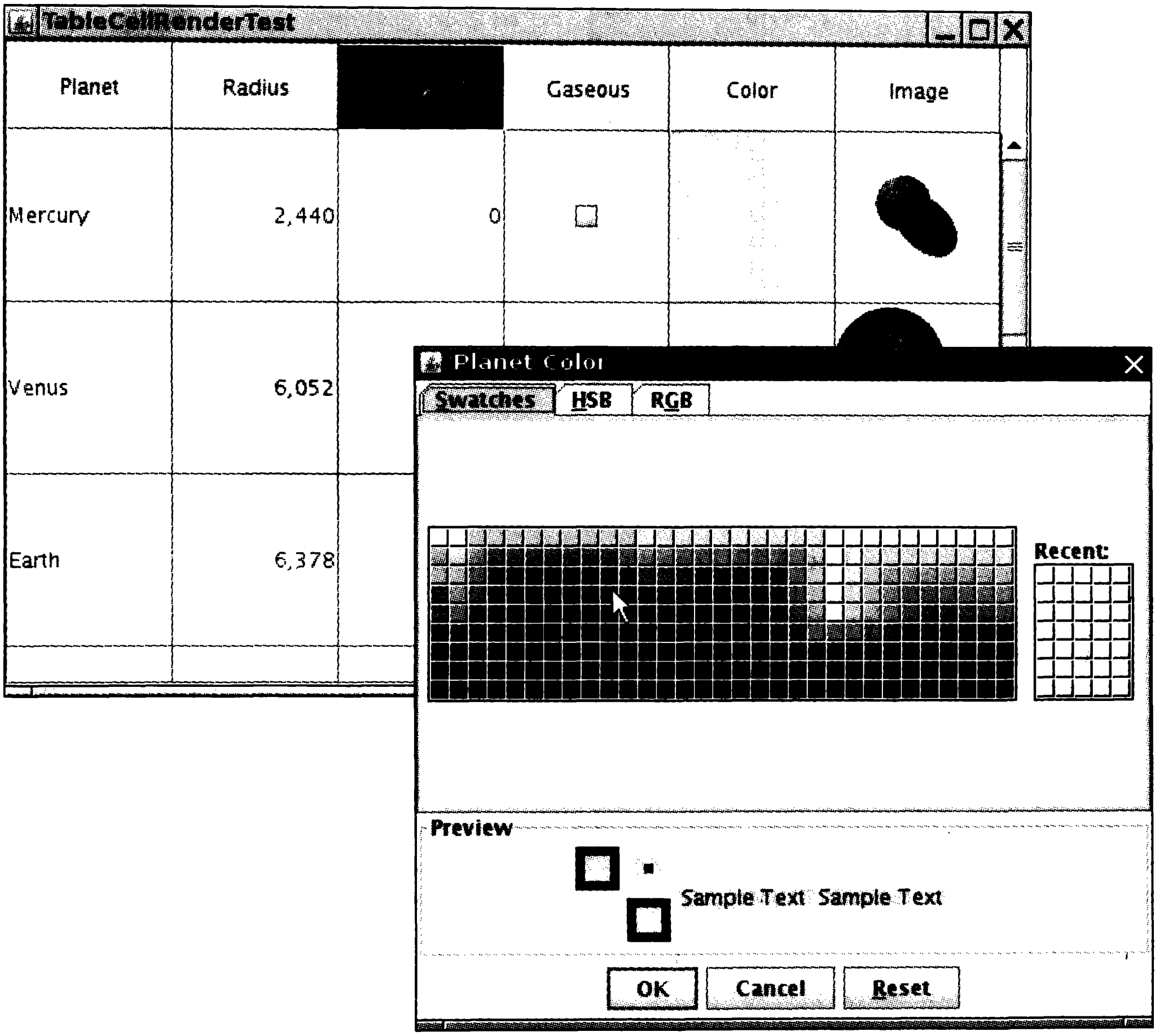


Рис. 6.14. Выбор цвета ячейки таблицы

Редактор цвета ячеек таблицы является не стандартным, а специальным, т.е. определяется и реализуется разработчиком прикладной программы. Для создания специального редактора ячеек таблицы следует реализовать интерфейс `TableCellEditor`. Впрочем, пользоваться этим интерфейсом не очень удобно, поэтому с версии JDK 1.3 для обработки событий предоставляется класс `AbstractClassEditor`.

Метод `getTableCellEditorComponent()` из интерфейса `TableCellEditor` запрашивает компонент разрешения на воспроизведение ячейки таблицы. Он действует таким же образом, как и метод `getTableCellRendererComponent()` из интерфейса `TableCellRenderer`, за исключением того, что у него отсутствует параметр `focus`. Дело в том, что при редактировании ячейки таблицы предполагается, что эта

ячейка обязательно имеет фокус ввода. На время редактирования компонент редактора временно *заменяет* средство воспроизведения. В данном случае возвращается пустая панель, для которой цвет не устанавливается. Это явно указывает пользователю на то, что в данный момент ячейка редактируется.

Далее требуется организовать отображение соответствующего редактора, как только пользователь щелкнет на ячейке таблицы. Компонент `JTable` вызывает редактор в ответ на соответствующее событие (например, щелчок кнопкой мыши). Для приема всех событий, которые могут инициировать редактирование, в классе `AbstractCellEditor` предусмотрен приведенный ниже метод. Но если переопределить этот метод таким образом, чтобы он возвращал логическое значение `false`, то компонент редактора не будет установлен в таблице.

```
public boolean isCellEditable(EventObject anEvent)
{
  return true;
}
```

После установки компонента редактора вызывается метод `shouldSelectCell()` — предположительно, с тем же самым событием. В этом методе происходит инициализация процесса редактирования, например, отображается окно внешнего редактора:

```
public boolean shouldSelectCell(EventObject anEvent)
{
  colorDialog.setVisible(true);
  return true;
}
```

Если пользователь откажется от редактирования ячейки таблицы, вызывается метод `cancelCellEditing()`, а если он щелкнет на другой ячейке таблицы, — метод `stopCellEditing()`. В обоих случаях следует закрыть диалоговое окно соответствующего редактора. При вызове метода `stopCellEditing()` в таблице может появиться частично отредактированное значение. Если такое значение является допустимым, следует возвратить логическое значение `true`. Любое значение, выбранное в окне селектора цвета, будет допустимым, но при редактировании других данных следует непременно убедиться в их достоверности.

Кроме того, из суперкласса следует вызвать методы, отвечающие за инициирование соответствующих событий. В противном случае редактирование не удастся отменить должным образом. В приведенном ниже фрагменте кода показано, каким образом отменяется редактирование.

```
public void cancelCellEditing()
{
  colorDialog.setVisible(false);
  super.cancelCellEditing();
}
```

И наконец, необходимо определить метод, возвращающий значение, получающееся в результате редактирования, как показано ниже.

```
public Object getCellEditorValue()
{
  return colorChooser.getColor();
}
```

Таким образом, к специальному редактору ячеек таблицы предъявляются следующие требования.

1. Он должен расширять класс `AbstractCellEditor` и реализовывать интерфейс `TableCellEditor`.
2. В нем должен быть определен метод `getTableCellEditorComponent()`, предназначенный для предоставления компонента редактора. Это может быть фиктивный компонент, если редактор предполагается отображать в отдельном диалоговом окне, или же компонент для редактирования *непосредственно* в ячейке, как, например, комбинированный список или текстовое поле.
3. В нем должны быть определены методы `shouldSelectCell()`, `stopCellEditing()` и `cancelCellEditing()` для управления запуском, завершением и отменой процесса редактирования. Для уведомления обработчиков событий из суперкласса должны быть вызваны методы `stopCellEditing()` и `cancelCellEditing()`.
4. В нем должен быть определен метод `getCellEditorValue()`, возвращающий значение, получаемое в результате редактирования.

Когда пользователь завершит редактирование, следует вызвать метод `stopCellEditing()` или `cancelCellEditing()`. Так, при создании диалогового окна селектора цвета устанавливаются приведенные ниже методы обратного вызова, уведомляющие о подтверждении или отмене результатов редактирования и инициирующие соответствующие события.

```
colorDialog =
  JColorChooser.createDialog(null, "Planet Color", false, colorChooser,
    EventHandler.create(ActionListener.class, this, "stopCellEditing"),
    EventHandler.create(ActionListener.class, this, "cancelCellEditing"));
```

На этом реализация специального редактора завершается. Теперь вы знаете, как сделать ячейку таблицы редактируемой и установить ее редактор. Остается открытым только один вопрос: как обновить модель с учетом отредактированного значения? По завершении редактирования компонент `JTable` вызывает следующий метод из модели таблицы:

```
void setValueAt(Object value, int r, int c)
```

Для сохранения нового значения этот метод придется переопределить. Параметр `value` теперь будет обозначать объект, возвращаемый редактором ячейки таблицы. В определении редактора ячейки таблицы задается тип объекта, который возвращается методом `getCellEditorValue()`. Так, при использовании класса `DefaultCellEditor` возможны три варианта указания типа для этого значения. Для редактора флажка это тип `boolean`, для текстового поля — тип `String`, а для комбинированного списка — тип объекта, выбираемого пользователем.

Если объект `value` имеет другой тип, его необходимо привести к нужному типу. Чаще всего это происходит при редактировании чисел в текстовом поле. В данном примере комбинированный список содержит объекты типа `Integer`, поэтому никакого приведения типов не требуется.

Листинг 6.8. Исходный код из файла `tableCellRender/TableCellRenderFrame.java`

```
1  package tableCellRender;
2
3  import java.awt.*;
4  import javax.swing.*;
```

```
5  import javax.swing.table.*;
6
7  /**
8   * Этот фрейм содержит таблицу с данными о планетах
9  */
10 public class TableCellRenderFrame extends JFrame
11 {
12   private static final int DEFAULT_WIDTH = 600;
13   private static final int DEFAULT_HEIGHT = 400;
14
15   public TableCellRenderFrame()
16   {
17     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19     TableModel model = new PlanetTableModel();
20     JTable table = new JTable(model);
21     table.setRowSelectionAllowed(false);
22
23     // установить средства воспроизведения и
24     // редактирования ячеек таблицы
25     table.setDefaultRenderer(
26        Color.class, new ColorTableCellRenderer());
27     table.setDefaultEditor(Color.class, new ColorTableCellEditor());
28
29     JComboBox<Integer> moonCombo = new JComboBox<>();
30     for (int i = 0; i <= 20; i++)
31        moonCombo.addItem(i);
32
33     TableColumnModel columnModel = table.getColumnModel();
34     TableColumn moonColumn =
35         columnModel.getColumn(PlanetTableModel.MOONS_COLUMN);
36     moonColumn.setCellEditor(new DefaultCellEditor(moonCombo));
37     moonColumn.setHeaderRenderer(
38        table.getDefaultRenderer(ImageIcon.class));
39     moonColumn.setHeaderValue(
40        new ImageIcon(getClass().getResource("Moons.gif")));
41
42     // показать таблицу
43     table.setRowHeight(100);
44     add(new JScrollPane(table), BorderLayout.CENTER);
45   }
46 }
```

Листинг 6.9. Исходный код из файла `tableCellRender/PlanetTableModel.java`

```
1  package tableCellRender;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.table.*;
6
7  /**
8   * Модель таблицы планет, определяющая значения, свойства
9   * воспроизведения и редактирования данных о планетах
10 */
11 public class PlanetTableModel extends AbstractTableModel
12 {
```

```
13   public static final int PLANET_COLUMN = 0;
14   public static final int MOONS_COLUMN = 2;
15   public static final int GASEOUS_COLUMN = 3;
16   public static final int COLOR_COLUMN = 4;
17
18   private Object[][] cells = {
19      { "Mercury", 2440.0, 0, false, Color.YELLOW,
20        new ImageIcon(getClass().getResource("Mercury.gif")) },
21      { "Venus", 6052.0, 0, false, Color.YELLOW,
22        new ImageIcon(getClass().getResource("Venus.gif")) },
23      { "Earth", 6378.0, 1, false, Color.BLUE,
24        new ImageIcon(getClass().getResource("Earth.gif")) },
25      { "Mars", 3397.0, 2, false, Color.RED,
26        new ImageIcon(getClass().getResource("Mars.gif")) },
27      { "Jupiter", 71492.0, 16, true, Color.ORANGE,
28        new ImageIcon(getClass().getResource("Jupiter.gif")) },
29      { "Saturn", 60268.0, 18, true, Color.ORANGE,
30        new ImageIcon(getClass().getResource("Saturn.gif")) },
31      { "Uranus", 25559.0, 17, true, Color.BLUE,
32        new ImageIcon(getClass().getResource("Uranus.gif")) },
33      { "Neptune", 24766.0, 8, true, Color.BLUE,
34        new ImageIcon(getClass().getResource("Neptune.gif")) },
35      { "Pluto", 1137.0, 1, false, Color.BLACK,
36        new ImageIcon(getClass().getResource("Pluto.gif")) } };
37
38   private String[] columnNames =
39      { "Planet", "Radius", "Moons", "Gaseous", "Color", "Image" };
40
41   public String getColumnName(int c)
42   {
43     return columnNames[c];
44   }
45
46   public Class<?> getColumnClass(int c)
47   {
48     return cells[0][c].getClass();
49   }
50
51   public int getColumnCount()
52   {
53     return cells[0].length;
54   }
55
56   public int getRowCount()
57   {
58     return cells.length;
59   }
60
61   public Object getValueAt(int r, int c)
62   {
63     return cells[r][c];
64   }
65
66   public void setValueAt(Object obj, int r, int c)
67   {
68     cells[r][c] = obj;
69   }
70   public boolean isCellEditable(int r, int c)
```

```
71    {
72      return c == PLANET_COLUMN || c == MOONS_COLUMN ||
73             c == GASEOUS_COLUMN || c == COLOR_COLUMN;
74    }
75 }
```

Листинг 6.10. Исходный код из файла `tableCellRender/ColorTableCellRenderer.java`

```
1  package tableCellRender;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.table.*;
6
7  /**
8   * Это средство воспроизведения отображает цвет
9   * в виде панели с заданным цветом
10 */
11 public class ColorTableCellRenderer extends
12       JPanel implements TableCellRenderer
13 {
14   public Component getTableCellRendererComponent(
15      JTable table, Object value, boolean isSelected,
16      boolean hasFocus, int row, int column)
17   {
18     setBackground((Color) value);
19     if (hasFocus) setBorder(
20     UIManager.getBorder("Table.focusCellHighlightBorder"));
21     else setBorder(null);
22     return this;
23   }
24 }
```

Листинг 6.11. Исходный код из файла `tableCellRender/ColorTableCellEditor.java`

```
1  package tableCellRender;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.beans.*;
6  import java.util.*;
7  import javax.swing.*;
8  import javax.swing.table.*;
9
10 /**
11  * Этот редактор открывает диалоговое окно селектора цвета
12  * для редактирования значения цвета в выбранной ячейке таблицы
13 */
14 public class ColorTableCellEditor extends
15     AbstractCellEditor implements TableCellEditor
16 {
17   private JColorChooser colorChooser;
18   private JDialog colorDialog;
19   private JPanel panel;
20   public ColorTableCellEditor()
21   {
```

```
22      panel = new JPanel();
23      // подготовить диалоговое окно выбора цвета
24
25      colorChooser = new JColorChooser();
26      colorDialog = JColorChooser.createDialog(
27         null, "Planet Color", false, colorChooser,
29         EventHandler.create(
30            ActionListener.class, this, "stopCellEditing"),
31         EventHandler.create(
32            ActionListener.class, this, "cancelCellEditing"));
33   }
34
35   public Component getTableCellEditorComponent(
36      JTable table, Object value, boolean isSelected,
37      int row, int column)
38   {
39      // Именно здесь получается текущее значение цвета,
40      // сохраняемое в диалоговом окне на тот случай,
41      // если пользователь начнет редактирование
42      colorChooser.setColor((Color) value);
43      return panel;
44   }
45
46   public boolean shouldSelectCell(EventObject anEvent)
47   {
48      // начать редактирование
49      colorDialog.setVisible(true);
50
51      // уведомить вызывающую часть программы о том, что
52      // эту ячейку разрешается выбрать
53      return true;
54   }
55
56   public void cancelCellEditing()
57   {
58      // редактирование отменено - скрыть диалоговое окно
59      colorDialog.setVisible(false);
60      super.cancelCellEditing();
61   }
62
63   public boolean stopCellEditing()
64   {
65      // редактирование завершено - скрыть диалоговое окно
66      colorDialog.setVisible(false);
67      super.stopCellEditing();
68
69      // уведомить вызывающую часть программы о том, что
70      // данное значение цвета разрешается использовать
71      return true;
72   }
73   public Object getCellEditorValue()
74   {
75      return colorChooser.getColor();
76   }
77 }
```

javax.swing.JTable 1.2

- `TableCellRenderer getDefaultRenderer(Class<?> type)`

  Получает средство воспроизведения, выбираемое по умолчанию для указанного типа ячейки таблицы.

- `TableCellEditor getDefaultEditor(Class<?> type)`

  Получает редактор, выбираемый по умолчанию для указанного типа ячейки таблицы.

javax.swing.table.TableCellRenderer 1.2

- `Component getTableCellRendererComponent(JTable table, Object value, boolean selected, boolean hasFocus, int row, int column)`

Возвращает компонент, метод `paint()` которого вызывается для воспроизведения ячейки таблицы.

Параметры:		
	table	Таблица, содержащая воспроизводимые ячейки
	value	Воспроизводимая ячейка
	selected	Принимает логическое значение `true`, если текущая ячейка выбрана
	hasFocus	Принимает логическое значение `true`, если текущая ячейка обладает фокусом ввода
	row, *column*	Номер строки и столбца с воспроизводимой ячейкой

javax.swing.table.TableColumn 1.2

- `void setCellEditor(TableCellEditor editor)`
- `void setCellRenderer(TableCellRenderer renderer)`

  Устанавливает редактор и средство воспроизведения для всех ячеек в указанном столбце.

- `void setHeaderRenderer(TableCellRenderer renderer)`

  Устанавливает средство воспроизведения для ячейки с заголовком в указанном столбце.

- `void setHeaderValue(Object value)`

  Устанавливает значение, которое должно быть отображено в виде заголовка в данном столбце.

javax.swing.DefaultCellEditor 1.2

- `DefaultCellEditor(JComboBox comboBox)`

  Создает редактор, предоставляющий комбинированный список для выбора значений ячеек таблицы.

javax.swing.table.TableCellEditor 1.2

- `Component getTableCellEditorComponent(JTable` *`table`*`, Object` *`value`*`, boolean` *`selected`*`, int` *`row`*`, int` *`column`*`)`

  Возвращает компонент, метод `paint()` которого вызывается для воспроизведения ячейки таблицы.

  Параметры:

`table`	Таблица, содержащая воспроизводимые ячейки
`value`	Воспроизводимая ячейка
`selected`	Принимает логическое значение `true`, если текущая ячейка выбрана
`row`, *`column`*	Номер строки и столбца с воспроизводимой ячейкой

javax.swing.CellEditor 1.2

- `boolean isCellEditable(EventObject` *`event`*`)`

  Возвращает логическое значение `true`, если событие пригодно для запуска процесса редактирования данной ячейки.

- `boolean shouldSelectCell(EventObject` *`anEvent`*`)`

  Запускает процесс редактирования. Возвращает, как правило, логическое значение `true`, если редактируемая ячейка должна быть *выбрана*. Если же требуется, чтобы содержимое выбранной ячейки не изменилось в результате редактирования, следует возвратить логическое значение `false`.

- `void cancelCellEditing()`

  Отменяет процесс редактирования. Его результаты можно проигнорировать.

- `boolean stopCellEditing()`

  Останавливает процесс редактирования с намерением использовать его результаты. Возвращает логическое значение `true`, если значение, получаемое в результате редактирования, оказывается допустимым.

- `Object getCellEditorValue()`

  Возвращает результаты редактирования.

- `void addCellEditorListener(CellEditorListener` *`l`*`)`
- `void removeCellEditorListener(CellEditorListener` *`l`*`)`

  Вводят или удаляют обязательный приемник событий, наступающих при редактировании ячеек таблицы.

Деревья

Каждому пользователю компьютера, обращающегося с иерархической файловой системой, знакомы древовидные представления файлов и каталогов. Это, конечно, лишь один из многих примеров применения древовидной структуры. В повседневной жизни такую же структуру образует система административно-территориального деления страны на штаты, области и города (рис. 6.15).

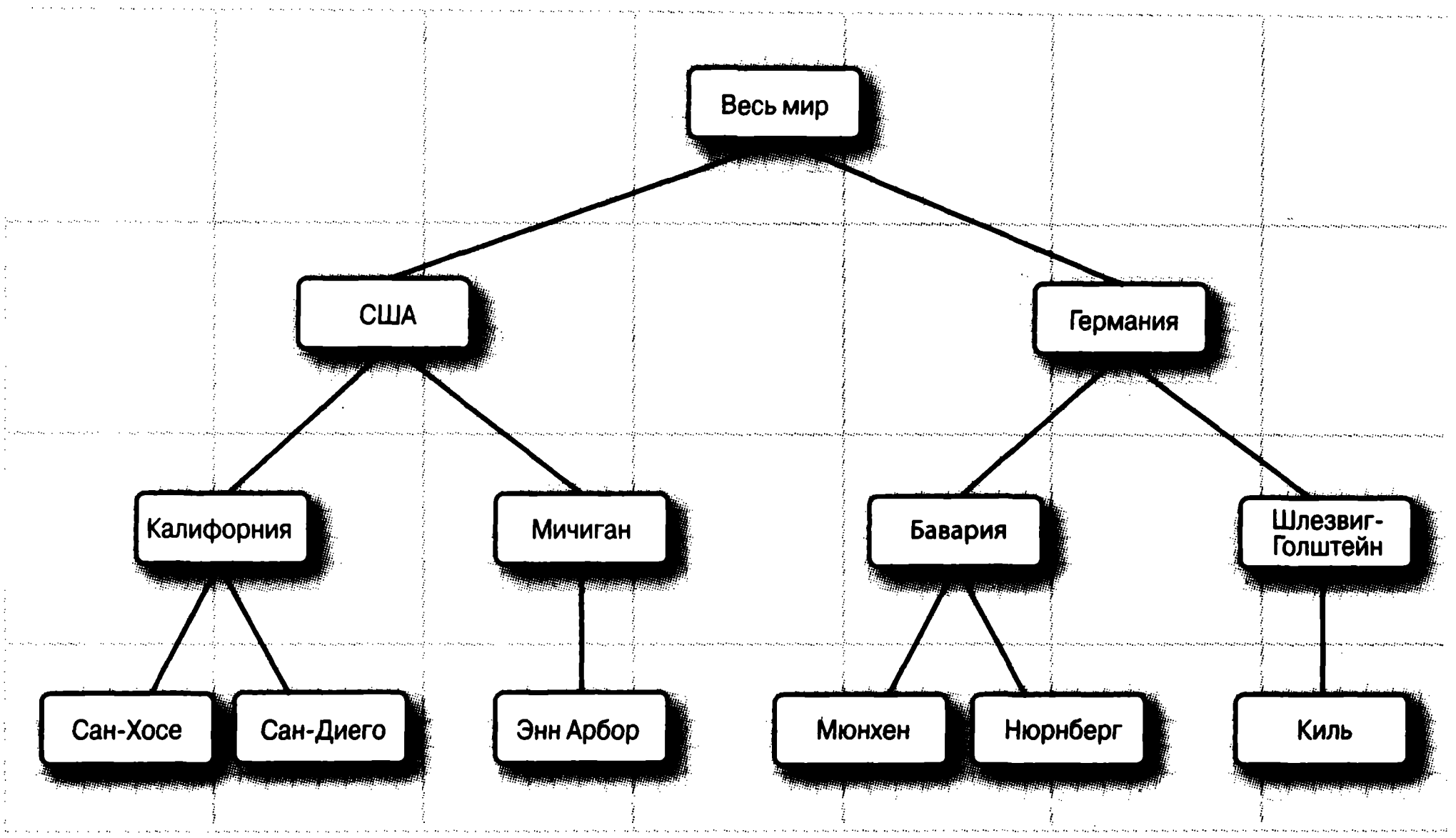


Рис. 6.15. Древовидная система административно-территориального деления страны на штаты, области и города

Программирующим на Java нередко приходится писать код для отображения подобных структур. И для этого в библиотеке Swing предусмотрен класс `JTree`. Вместе со вспомогательными классами он берет на себя все хлопоты по компоновке древовидной структуры и обработке запросов пользователей на развертывание и свертывание узлов дерева. В этом разделе показывается, как пользоваться средствами, доступными в классе `Jtree`, для построения древовидных структур.

Подобно другим сложным компонентам Swing, здесь рассматриваются только самые общие и наиболее распространенные приемы обращения с деревьями. А для углубленного изучения данного вопроса рекомендуется упоминавшаяся ранее дополнительная литература: *Graphic Java™: Mastering the JFC, Volume II: Swing, 3rd Edition* Дэвида М. Гери или *Swing: Advanced Programming* Кима Топли.

Приступая к рассмотрению деревьев, необходимо сначала пояснить характерную для них терминологию (рис. 6.16). Прежде всего, дерево состоит из *узлов*. Каждый узел может быть *листом* (т.е. *краевым узлом*) или *иметь дочерние узлы*. У каждого узла, кроме корневого, имеется только один *родительский узел*. А у всего дерева в целом имеется только один *корневой узел*. Ряд деревьев с собственными корневыми узлами называется *лесом*.

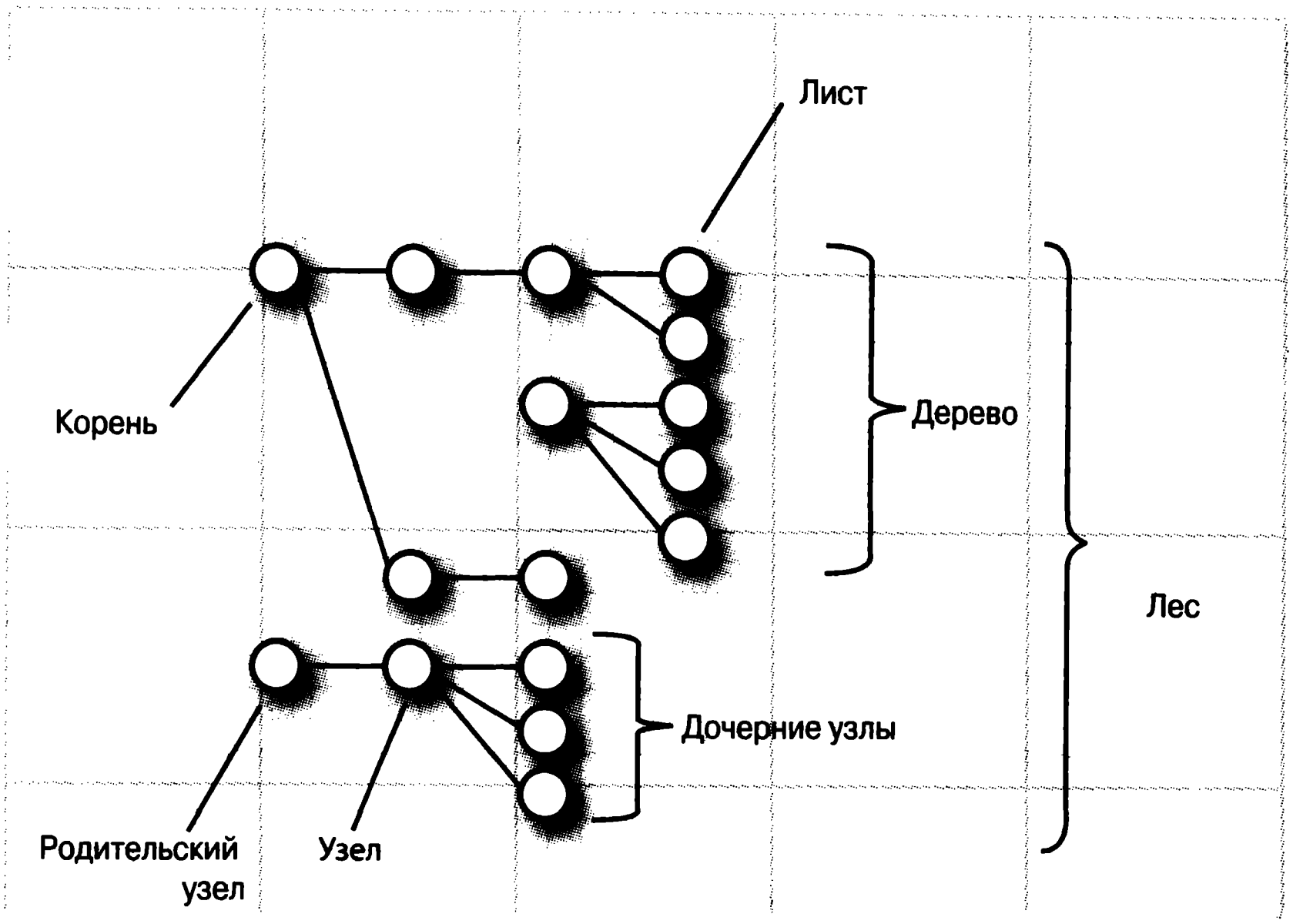


Рис. 6.16. Терминология, употребляемая для описания древовидных структур

Простые деревья

В первом несложном примере программы будет построено дерево лишь с несколькими узлами, как показано далее на рис. 6.18. Подобно многим другим компонентам библиотеки Swing, программирующий на Java создает модель иерархических данных, а компонент `JTree` автоматически отображает их. Для создания объекта типа `JTree` его конструктору передается модель дерева следующим образом:

```
TreeModel model = . . .;
JTree tree = new JTree(model);
```

НА ЗАМЕТКУ! Некоторые конструкторы создают деревья из коллекции составляющих элементов, как показано ниже.

```
JTree(Object[] nodes)
JTree(Vector<?> nodes)
JTree(Hashtable<?, ?> nodes) // значения становятся узлами
```

Такие конструкторы практически бесполезны, потому что они создают только лес деревьев, каждое из которых содержит единственный узел. А третий конструктор из тех, что приведены выше, и вовсе бесполезен, поскольку узлы передаются ему в произвольном порядке, который определяется хеш-кодами ключей.

Как же получить модель дерева? С помощью класса, реализующего интерфейс `TreeModel`, можно создать собственную модель. Такой способ будет рассмотрен далее в этой главе, а до тех пор воспользуемся моделью дерева типа `DefaultTreeModel`, предоставляемой в библиотеке Swing по умолчанию. Для построения используемой по умолчанию модели дерева нужно предоставить конструктору корневой узел следующим образом:

```
TreeNode root = . . .;
DefaultTreeModel model = new DefaultTreeModel(root);
```

где TreeNode — это еще один интерфейс. Используемую по умолчанию модель можно заполнить экземплярами любого класса, реализующего данный интерфейс. В данном случае применяется уже готовый класс узлов DefaultMutableTreeNode из библиотеки Swing. Как показано на рис. 6.17, этот класс реализует интерфейс MutableTreeNode, который является дочерним по отношению к интерфейсу TreeNode.

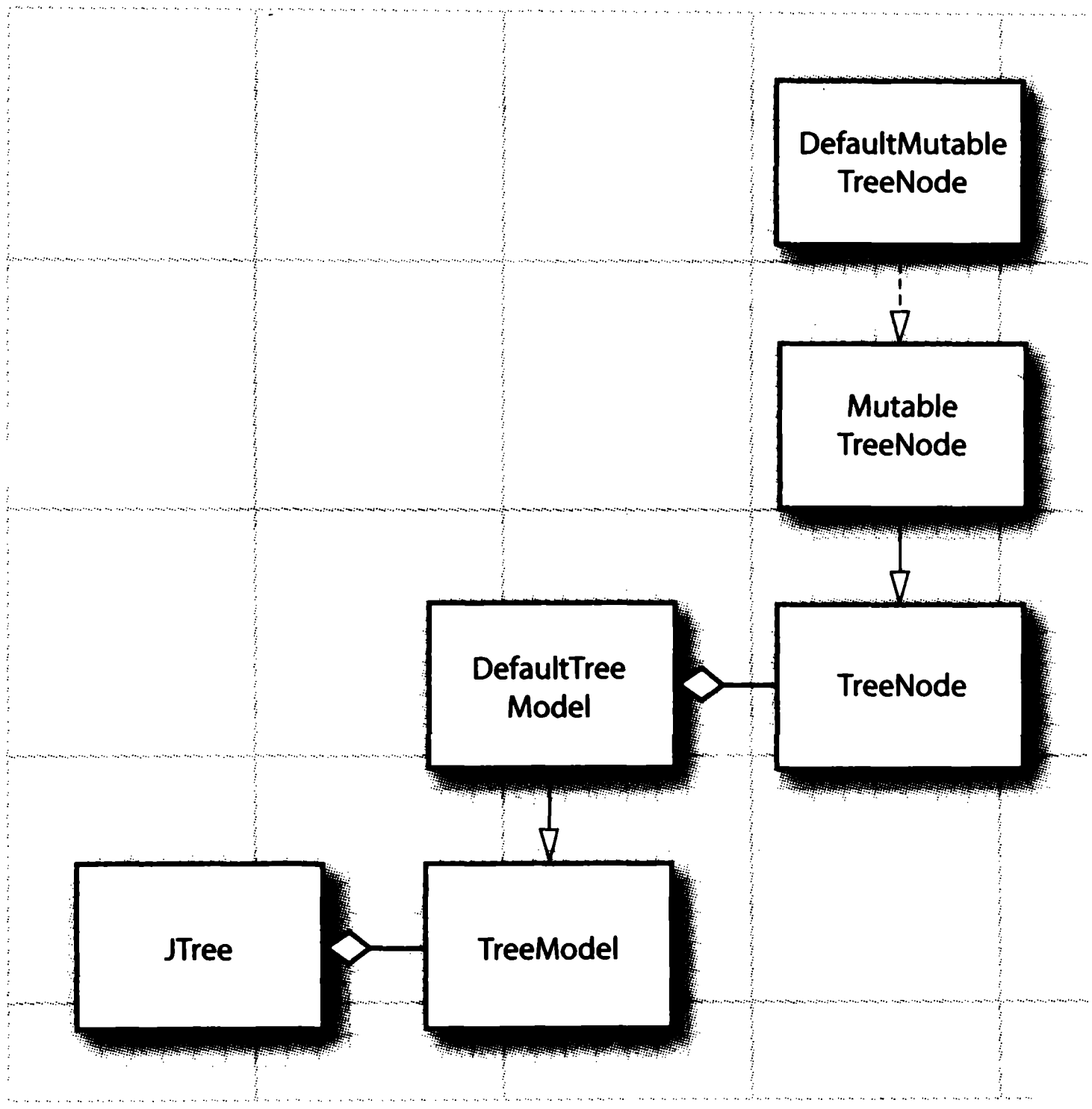


Рис. 6.17. Иерархия наследования интерфейсов и классов, используемых при построении деревьев

Изменяемый узел дерева по умолчанию (т.е. экземпляр класса DefaultMutableTreeNode) содержит *пользовательский объект*. Эти объекты воспроизводятся для всех узлов. Если не задано специальное средство воспроизведения, то в дереве отображается символьная строка, получаемая в результате выполнения метода toString().

В первом рассматриваемом здесь примере построения деревьев в качестве пользовательских объектов применяются символьные строки, хотя на практике древовидные структуры обычно заполняются более сложными объектами. Например, для отображения дерева каталогов в качестве его узлов имеет смысл использовать объекты типа File. Пользовательский объект можно указать сразу в конструкторе или в дальнейшем с помощью метода setUserObject(), как показано ниже.

```
DefaultMutableTreeNode node = new DefaultMutableTreeNode("Texas");
. . .
node.setUserObject("California");
```

Затем между узлами следует установить отношения "родительский–дочерний". Для этого нужно ввести дочерние узлы с помощью метода `add()`, начиная с корневого узла, как показано в приведенном ниже фрагменте кода. Получаемое в итоге простое дерево приведено на рис. 6.18.

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
root.add(country);
DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
country.add(state);
```

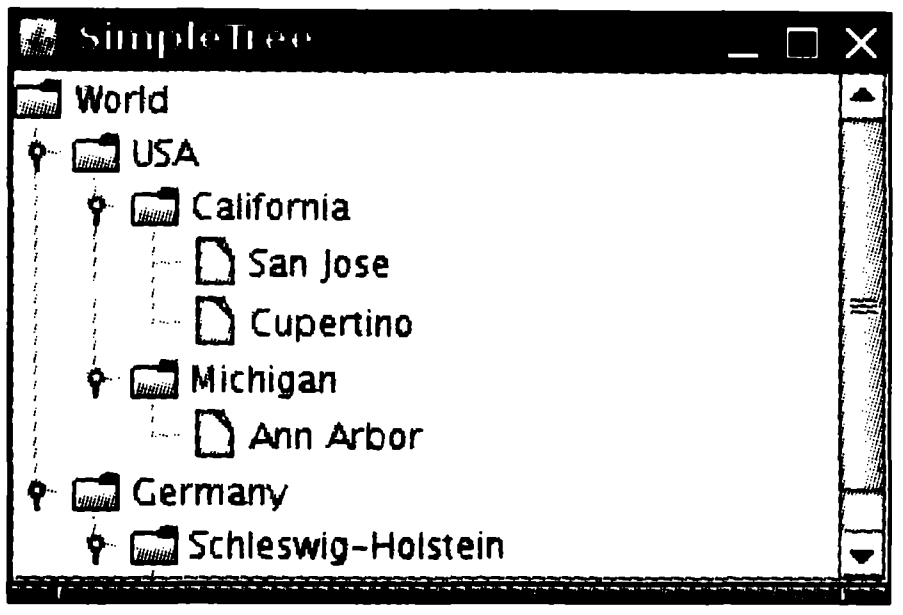


Рис. 6.18. Простое дерево

Подобным образом следует связать все узлы, а затем построить модель дерева типа `DeafultTreeNode` с корневым узлом, а на основе этой модели — само дерево с помощью компонента `JTree`:

```
DefaultTreeModel treeModel = new DefaultTreeModel(root);
JTree tree = new JTree(treeModel);
```

Можно поступить еще проще, передав корневой узел конструктору класса `JTree`, который автоматически создаст модель дерева по умолчанию, как показано ниже. Весь исходный код рассматриваемого здесь примера программы, демонстрирующей построение простого дерева, приведен в листинге 6.12.

```
JTree tree = new JTree(root);
```

Листинг 6.12. Исходный код из файла `tree/SimpleTreeFrame.java`

```
1  package tree;
2
3  import javax.swing.*;
4  import javax.swing.tree.*;
5
6  /**
7   * Этот фрейм содержит простое дерево, отображающее построенную
8   * вручную модель дерева
9  */
10 public class SimpleTreeFrame extends JFrame
11 {
12   private static final int DEFAULT_WIDTH = 300;
13   private static final int DEFAULT_HEIGHT = 200;
14
15   public SimpleTreeFrame()
16   {
```

```
17      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19      // подготовить данные для модели дерева
20
21      DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
22      DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
23      root.add(country);
24      DefaultMutableTreeNode state =
25         new DefaultMutableTreeNode("California");
26      country.add(state);
27      DefaultMutableTreeNode city =
28         new DefaultMutableTreeNode("San Jose");
29      state.add(city);
30      city = new DefaultMutableTreeNode("Cupertino");
31      state.add(city);
32      state = new DefaultMutableTreeNode("Michigan");
33      country.add(state);
34      city = new DefaultMutableTreeNode("Ann Arbor");
35      state.add(city);
36      country = new DefaultMutableTreeNode("Germany");
37      root.add(country);
38      state = new DefaultMutableTreeNode("Schleswig-Holstein");
39      country.add(state);
40      city = new DefaultMutableTreeNode("Kiel");
41      state.add(city);
42
43      // построить дерево и разместить его на прокручиваемой панели
44
45      JTree tree = new JTree(root);
46      add(new JScrollPane(tree));
47   }
48 }
```

Дерево, отображаемое при выполнении данной программы, будет выглядеть так, как показано на рис. 6.19. В рабочем окне программы будут видны только корневой узел и его дочерние узлы. Для развертывания поддеревьев нужно щелкнуть мышью на пиктограмме кружка (*маркере* узла). Линия, соединяющаяся с кружком, направлена вправо, если поддерево свернуто, или вниз, если оно развернуто (рис. 6.20). Неизвестно, что имели в виду разработчики визуального стиля Metal, но, по-видимому, кружок, соединяемый с линией, обозначает дверную ручку, которую нужно повернуть вниз по часовой стрелке, чтобы "открыть" поддерево.

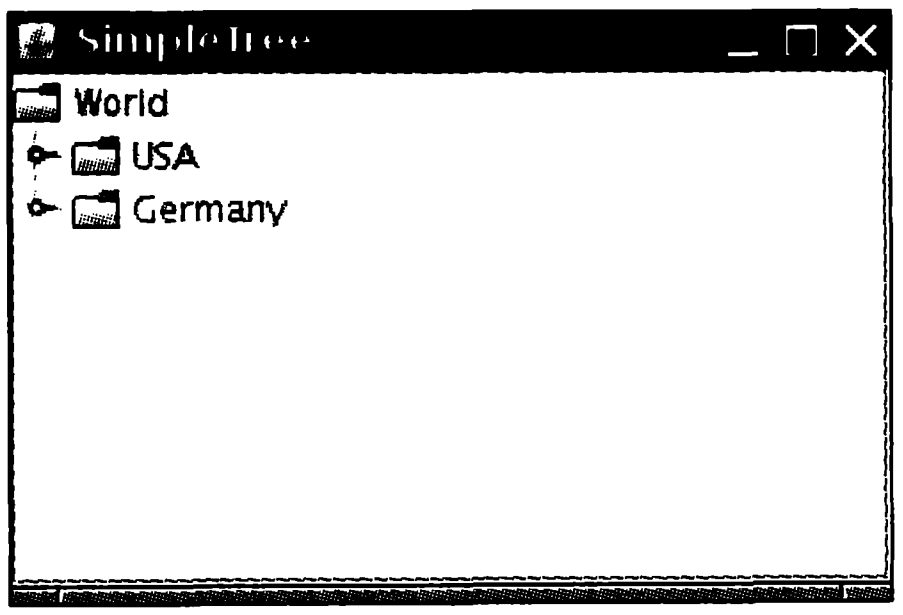


Рис. 6.19. Исходное состояние отображаемого дерева

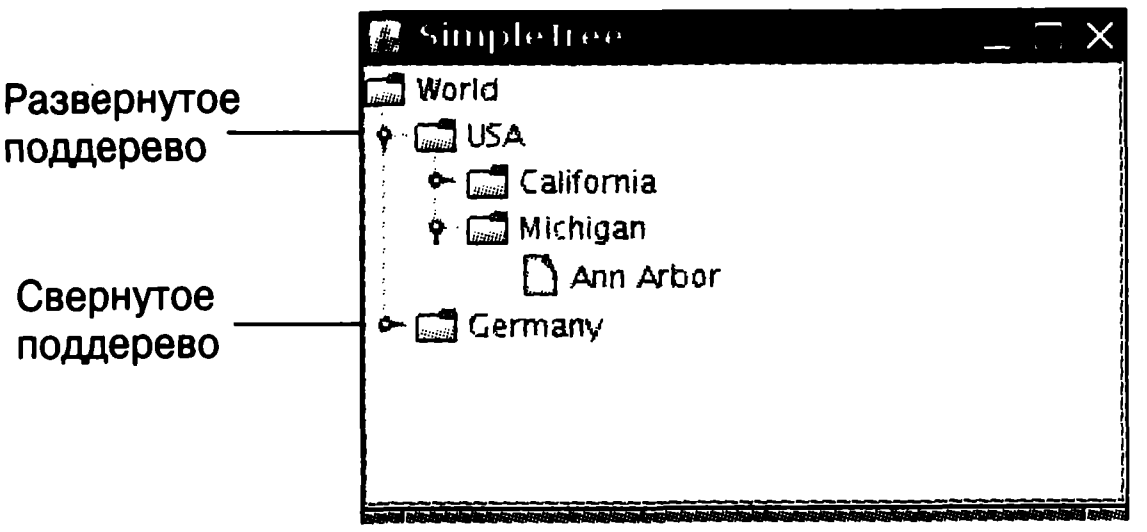


Рис. 6.20. Свернутые и развернутые поддеревья

НА ЗАМЕТКУ! Разумеется, внешний вид дерева зависит от выбранного визуального стиля. Приведенное выше описание относится к визуальному стилю Metal. А в визуальных стилях Windows и Motif для обозначения свернутого и развернутого дерева используются знаки + и – (рис. 6.21).

Чтобы скрыть линии, связывающие родительские и дочерние узлы, как показано на рис. 6.22, нужно указать значение `None` свойства `JTree.lineStyle` следующим образом:

```
tree.putClientProperty("JTree.lineStyle", "None");
```

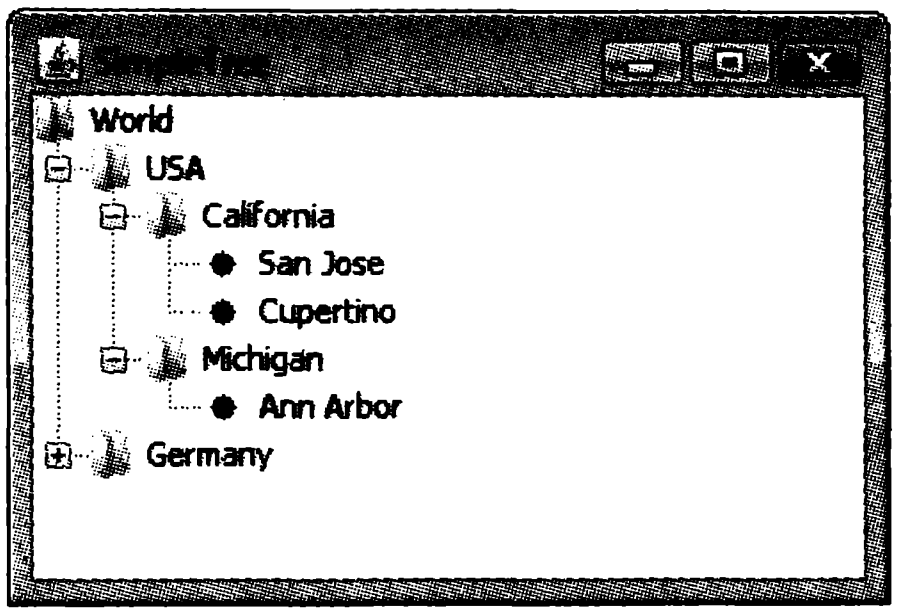


Рис. 6.21. Вид дерева, отображаемого в визуальном стиле Windows

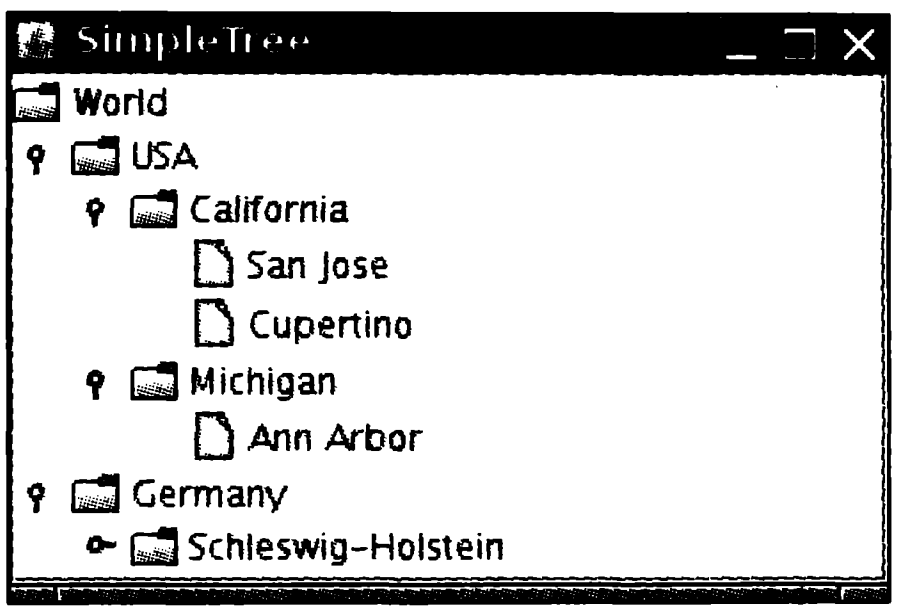


Рис. 6.22. Вид дерева без соединительных линий между узлами

А для отображения этих линий нужно указать значение `Angled` данного свойства, как показано ниже.

```
tree.putClientProperty("JTree.lineStyle", "Angled");
```

На рис. 6.23 показан еще один стиль представления структуры дерева с помощью горизонтальных линий. Такое дерево отображается с горизонтальными линиями, разделяющими только дочерние узлы корневого узла. Но трудно себе представить ситуацию, когда могла бы пригодиться такая древовидная структура.

По умолчанию у корневого узла отсутствует маркер для свертывания всего дерева. Если этот маркер требуется, то следует вызвать приведенный ниже метод. На рис. 6.24 представлен результат вызова данного метода. Теперь все дерево можно сворачивать и разворачивать полностью.

```
tree.setShowsRootHandles(true);
```

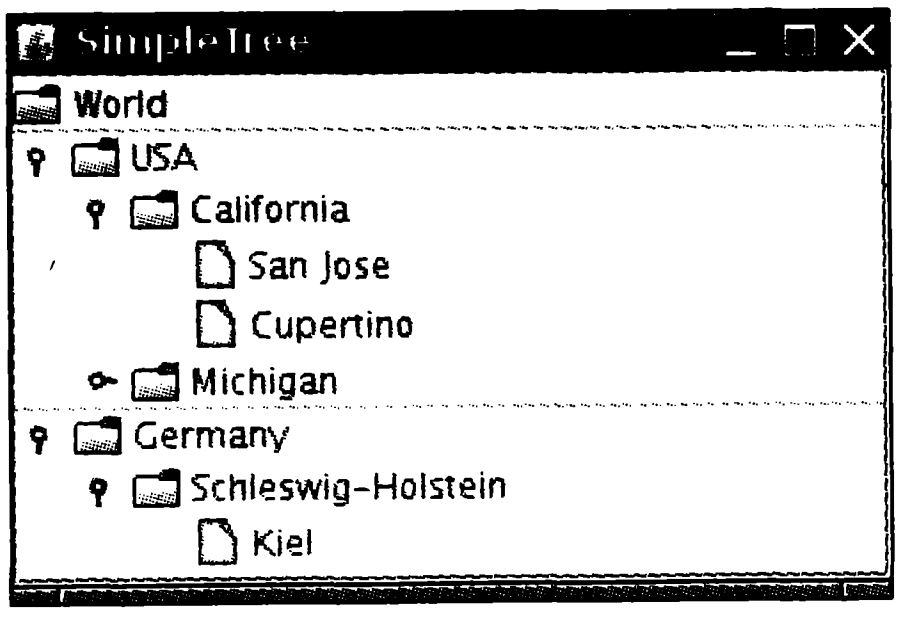


Рис. 6.23. Вид дерева с разделяющими горизонтальными линиями

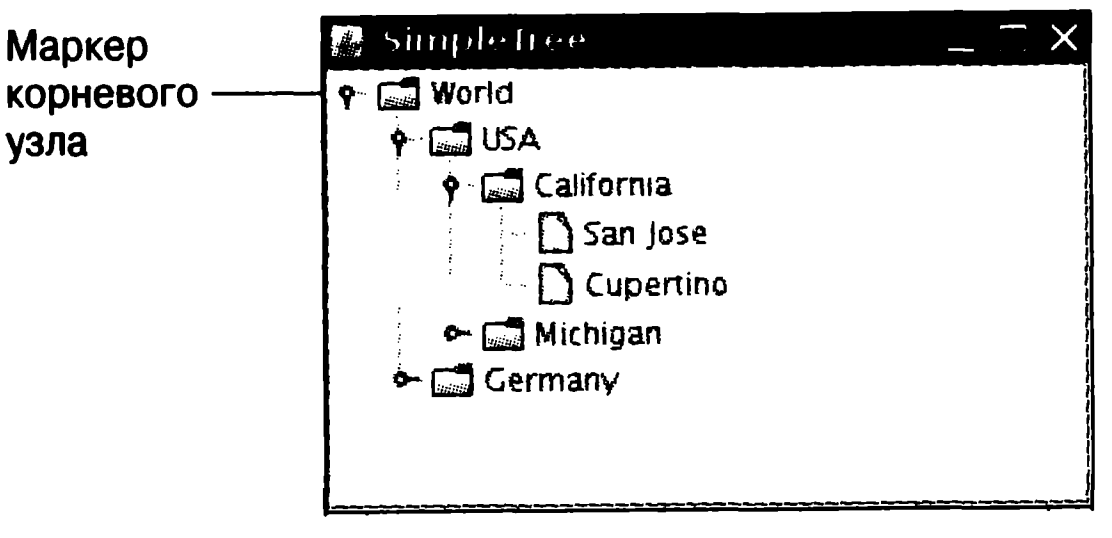


Рис. 6.24. Дерево с маркером корневого узла

Кроме того, корневой узел можно скрыть. Это может пригодиться, например, для отображения леса, т.е. ряда деревьев с собственными корневыми узлами. Все деревья следует объединить в дереве с общим корнем, а затем скрыть этот общий корень с помощью следующего метода:

```
tree.setRootVisible(false);
```

На рис. 6.25 показан пример леса с двумя деревьями, имеющими корневые узлы USA и Germany, объединенные в одном дереве со скрытым корневым узлом. Перейдем теперь от корня к листьям дерева. Для их отображения используется пиктограмма, приведенная на рис. 6.26.

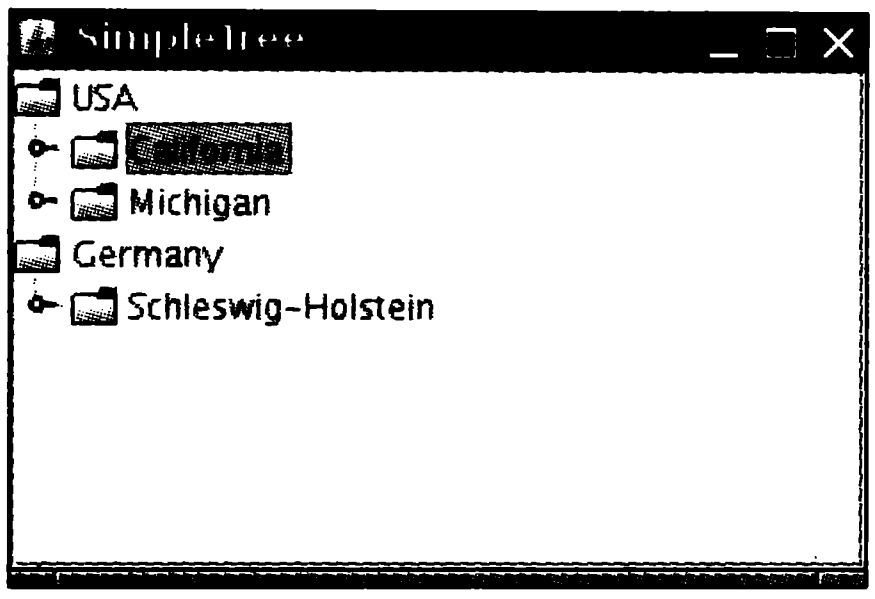


Рис. 6.25. Вид леса

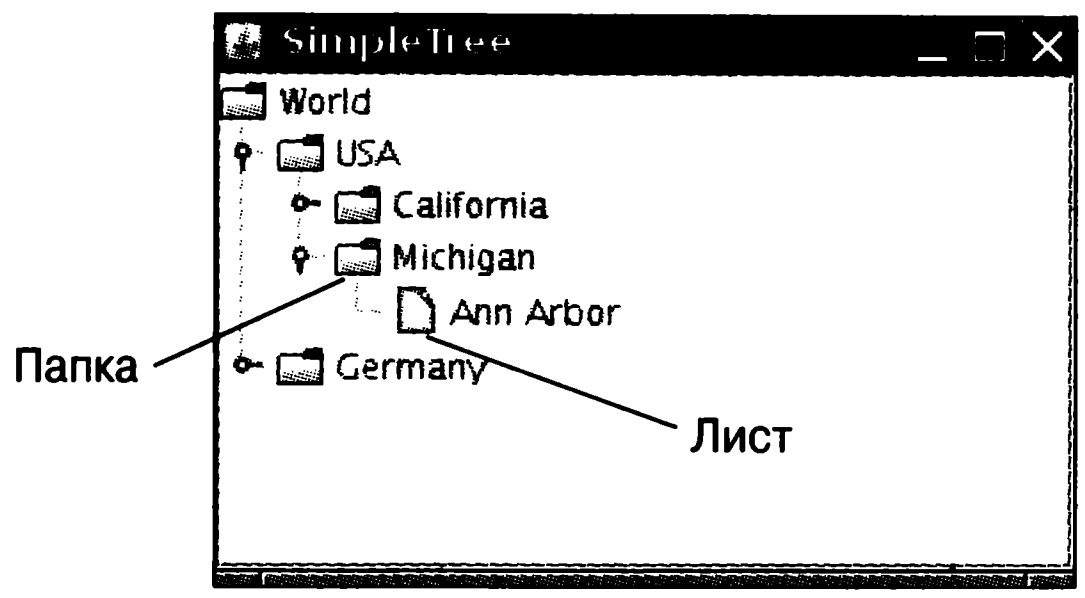


Рис. 6.26. Пиктограммы для отображения разнотипных узлов

Итак, каждый узел отображается отдельной пиктограммой. Для обозначения узлов дерева имеются три вида пиктограмм: листа бумаги, открытой и закрытой папки. Средству воспроизведения узлов должно быть известно, какой именно пиктограммой следует отображать каждый узел. По умолчанию решение принимается следующим образом: если метод `isLeaf()` возвращает логическое значение `true`, то используется пиктограмма листа бумаги, в противном случае — пиктограмма папки.

Метод `isLeaf()` из класса `DefaultMutableTreeNode` возвращает логическое значение `true`, если у данного узла отсутствуют дочерние узлы. Таким образом, узлы дерева с дочерними узлами будут отображаться в виде папок, а узлы дерева без дочерних узлов — в виде листов бумаги.

Но такой способ обозначения узлов дерева подходит не для всех случаев. Например, при добавлении узла `Montana` в дерево для отображения штата Монтана без указания городов этот штат будет обозначен пиктограммой листа бумаги. Но при этом будет нарушен сам принцип представления дерева, по которому такие пиктограммы служат для обозначения городов.

Компоненту `JTree` неизвестно, какие именно узлы являются листами, поэтому для выяснения этого факта он обращается к модели дерева. Если же узел без дочерних узлов не является листом в принципе, то для определения листов можно выбрать другой критерий, например, обратиться к свойству узла, определяющему допустимость в нем дочерних узлов. Так, если для некоторого узла дочерние узлы недопустимы, то нужно вызвать следующий метод:

```
node.setAllowsChildren(false);
```

Затем необходимо запросить модель дерева, чтобы она выяснила с помощью свойства допустимости дочерних узлов, является ли узел листом и следует ли

отображать его пиктограммой листа бумаги. Для этой цели нужно вызвать метод `setAsksAllowsChildren()` из класса `DefaultTreeModel` следующим образом:

```
model.setAsksAllowsChildren(true);
```

В таком случае те узлы, в которых допускается наличие дочерних узлов, будут обозначены пиктограммами папок, а те узлы, для которых не допускается наличие дочерних узлов, — пиктограммами листов бумаги. С другой стороны, задавая в конструкторе дерева корневой узел, можно также указать на необходимость запрашивать свойство допустимости дочерних узлов, как показано ниже.

```
JTree tree = new JTree(root, true); // узлы, в которых не допускаются
          // дочерние узлы обозначаются пиктограммами листов бумаги
```

javax.swing.JTree 1.2

- `JTree(TreeModel` *`model`*`)`

  Конструирует дерево из указанной модели.

- `JTree(TreeNode` *`root`*`)`

- `JTree(TreeNode` *`root`*`, boolean` *`asksAllowChildren`*`)`

  Конструируют дерево с заданной по умолчанию моделью, отображающее корневой узел и его дочерние узлы.

Параметры:		
	`root`	Корневой узел
	`asksAllowChildren`	Логическое значение `true` предписывает использовать свойство допустимости дочерних узлов, чтобы выяснить, является ли узел листом

- `void setShowsRootHandles(boolean` *`b`*`)`

  Если параметр *b* принимает логическое значение `true`, то в корневом узле дерева отображается маркер свертывания.

- `void setRootVisible(boolean` *`b`*`)`

  Если параметр *b* принимает логическое значение `true`, то корневой узел отображается, в противном случае — скрывается.

javax.swing.tree.TreeNode 1.2

- `boolean isLeaf()`

  Возвращает логическое значение `true`, если данный узел является листом.

- `boolean getAllowsChildren()`

  Возвращает логическое значение `true`, если данный узел может иметь дочерние узлы.

javax.swing.tree.MutableTreeNode 1.2

- `void setUserObject(Object` *`userObject`*`)`

  Задает пользовательский объект типа *`userObject`*, используемый для воспроизведения узла дерева.

javax.swing.tree.TreeModel 1.2

- `boolean isLeaf(Object` *`node`*`)`

  Возвращает логическое значение `true`, если узел *`node`* следует отобразить как лист дерева.

javax.swing.tree.DefaultTreeModel 1.2

- `void setAsksAllowsChildren(boolean` *`b`*`)`

  Если параметр *`b`* принимает значение `true`, то узлы отображаются как листья дерева, при условии, что метод `getAllowsChildren()` возвращает логическое значение `false`. В противном случае такой внешний вид узлов будет выбираться, если метод `isLeaf()` возвращает логическое значение `true`.

javax.swing.tree.DefaultMutableTreeNode 1.2

- `DefaultMutableTreeNode(Object` *`userObject`*`)`

  Создает изменяемый узел дерева с указанным пользовательским объектом.

- `void add(MutableTreeNode` *`child`*`)`

  Вводит узел как последний дочерний узел в данном узле дерева.

- `void setAllowsChildren(boolean` *`b`*`)`

  Если параметр *`b`* принимает логическое значение `true`, в данный узел могут быть введены дочерние узлы.

javax.swing.JComponent 1.2

- `void putClientProperty(Object` *`key`*`, Object` *`value`*`)`
- Вводит указанную пару "ключ-значение" в небольшую таблицу, которой управляет каждый компонент. Этот "запасной" механизм используется в некоторых компонентах библиотеки Swing для хранения специальных свойств, определяющих их внешний вид.

Редактирование деревьев и путей к деревьям

В следующем примере программы демонстрируются способы редактирования деревьев. На рис. 6.27 показан пользовательский интерфейс данной программы, где для создания нового узла под именем `New` предусмотрены кнопки Add Sibling

(Добавить равноправный узел) и Add Child (Добавить дочерний узел). Для удаления текущего выбранного узла предназначена кнопка Delete (Удалить).

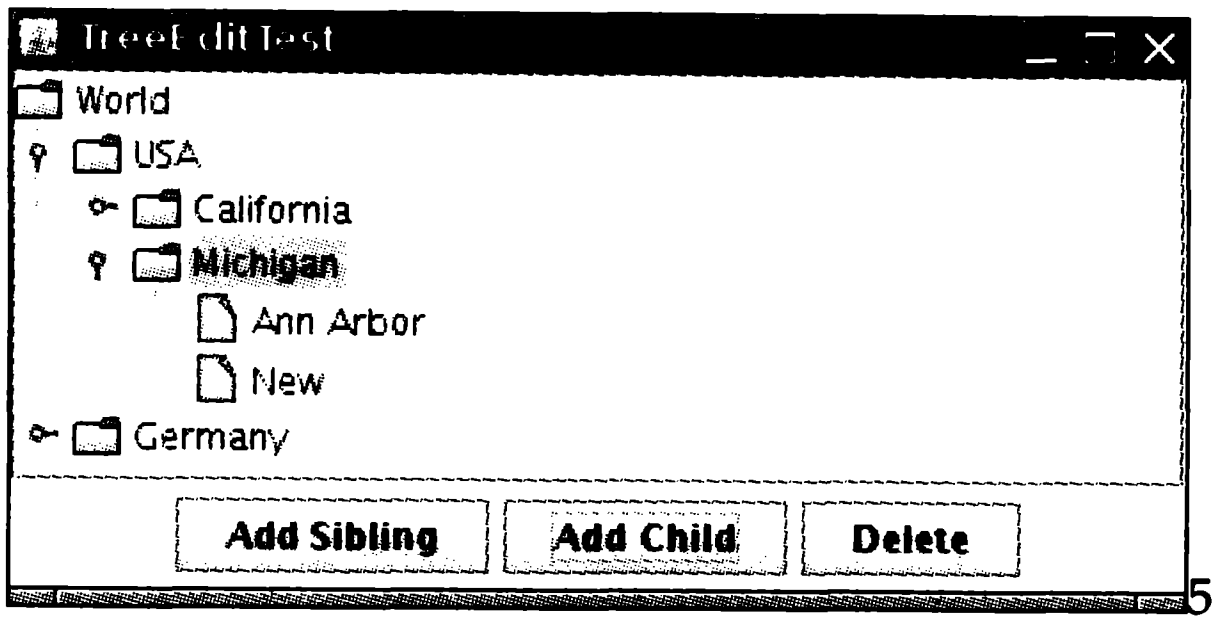


Рис. 6.27. Рабочее окно программы, демонстрирующей способы редактирования дерева

Для реализации такого поведения нужно найти текущий выбранный узел. В компоненте `JTree` предусмотрен интересный способ поиска такого узла на основе *пути к объекту*, называемого иначе *путем к дереву*. Такой путь начинается с корневого узла и состоит из последовательности дочерних узлов, а характерный его пример приведен на рис. 6.28.

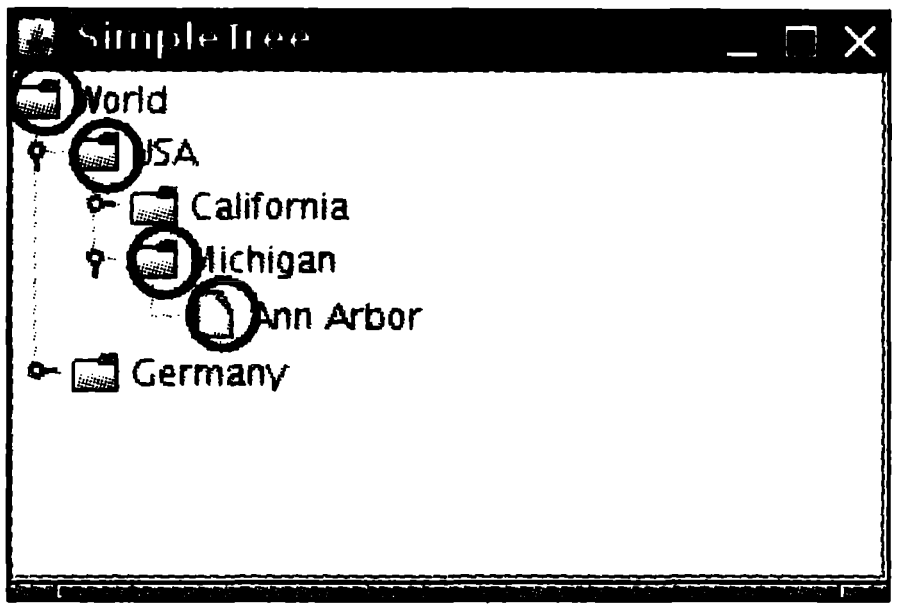


Рис. 6.28. Пример пути к дереву

Казалось бы, для поиска текущего узла лучше было бы воспользоваться интерфейсом `TreeNode` и методом `getParent()`. Но на самом деле компоненту `JTree` ничего неизвестно об интерфейсе `TreeNode`, поскольку он используется только для реализации в классе `DefaultTreeModel`, а не в интерфейсе `TreeModel`. Дерево может иметь модель, в узлах которой не реализуется интерфейс `TreeNode`. Например, в модели дерева с другими типами объектов могут вообще отсутствовать методы `getParent()` и `getChild()`. В таком случае для организации связей между узлами применяется модель дерева, но компоненту `JTree` ничего о них неизвестно. Поэтому в компоненте `JTree` предполагается всегда использовать полные пути к деревьям.

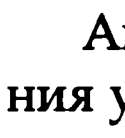
Класс `TreePath` управляет последовательностью ссылок на объекты типа `Object` (а не `TreeNode`) с помощью нескольких методов. Например, для получения ссылки на последний узел пути можно вызвать метод `getLastPathComponent()`. А для поиска текущего выбранного узла служит метод `getSelectionPath()` из класса `JTree`. Таким образом, зная путь к дереву (на основании объекта типа `TreePath`), можно получить ссылку на текущий выбранный узел следующим образом:

```
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode
    = (DefaultMutableTreeNode) selectionPath.getLastPathComponent();
```

Такой запрос выполняется очень часто, поэтому для него разработан приведенный ниже служебный метод.

```
DefaultMutableTreeNode selectedNode
    = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
```

Этот метод называется `getLastSelectedPathComponent()`, а не `getSelectedNode()`, поскольку самому дереву ничего неизвестно об узлах, а его модель оперирует только с путями к объектам.



НА ЗАМЕТКУ! Кроме путей к деревьям, для описания узлов дерева используются методы из класса `JTree`, которые принимают или возвращают целочисленный индекс, обозначающий позицию строки, т.е. номер строки, начиная с нуля, для узла во внутреннем представлении дерева. Такие номера имеют только видимые узлы, причем номер строки может меняться при свертывании, развертывании или изменении дерева. Поэтому номера, или позиции, строк не рекомендуется применять для доступа к узлам. У каждого метода из класса `JTree`, предназначенного для работы с позициями строк, имеется эквивалентный метод, оперирующий путями к деревьям.

Получив выбранный узел, можно приступать к его редактированию. Но добавить к нему дочерние узлы нельзя, просто вызвав метод `add()` следующим образом:

```
selectedNode.add(newNode); // Нельзя!
```

Если изменяется структура узлов, то изменения вносятся в модель дерева, а связанное с ним представление об этом не уведомляется. Следовательно, такое уведомление нужно послать самостоятельно. Но если для ввода нового узла вызвать метод `insertNodeInto()` из класса `DefaultTreeModel`, то такое уведомление будет отправлено представлению дерева автоматически:

```
model.insertNodeInto(newNode, selectedNode, selectedNode.getChildCount());
```

Аналогичным образом применяется метод `removeNodeFromParent()` для удаления узла и уведомления об обновлении представления дерева:

```
model.removeNodeFromParent(selectedNode);
```

Для изменения пользовательского объекта, но с сохранением структуры узла нужно вызвать метод `nodeChanged()` следующим образом:

```
odel.nodeChanged(changedNode);
```

Автоматическое уведомление является основным достоинством модели типа `DefaultTreeModel`. При создании собственных моделей деревьев такое уведомление приходится организовывать самостоятельно. Более подробно этот вопрос обсуждается в упоминавшей ранее книге *Swing: Advanced Programming* Кима Топли.

ВНИМАНИЕ! В состав класса `DeafultTreeModel` входит метод `reload()`, полностью перезагружающий модель дерева. Но метод `reload()` не следует вызывать только для обновления дерева после внесения нескольких изменений в нее. Дело в том, что при таком обновлении дерева все узлы за пределами дочерних узлов корневого узла снова будут свернуты. Такое поведение дерева может оказаться неудобным для пользователей, вынуждая их всякий раз развертывать дерево.

Если представление получает уведомление об изменении структуры узлов дерева, оно обновляет отображение узлов, не развертывая их для просмотра вновь добавленных узлов. Например, добавление в рассматриваемом здесь примере программы нового дочернего узла в свернутые узлы произойдет незаметно для пользователя. В таком случае придется специально организовать развертывание родительских узлов для отображения введенного нового дочернего узла. Для этого можно вызвать из класса `JTree` метод `makeVisible()`, принимающий путь, ведущий к отображаемому на экране узлу.

Таким образом, для отображения вновь введенного узла придется сформировать путь к нему от корневого узла дерева. Для получения этого пути сначала следует вызвать метод `getPathToRoot()` из класса `DeafultTreeModel`. Он возвращает массив `TreeArray[]` для всех узлов (от текущего до корневого), который далее передается конструктору класса `TreePath`. В приведенном ниже фрагменте когда показано, каким образом вновь введенный узел делается видимым.

```
TreeNode[] nodes = model.getPathToRoot(newNode);
TreePath path = new TreePath(nodes);
tree.makeVisible(path);
```

НА ЗАМЕТКУ! Как ни странно, класс `DefaultTreeModel` ведет себя так, как будто ему вообще ничего неизвестно о классе `TreePath`, хотя он и предназначается для взаимодействия с классом `JTree`. В то же время в классе `JTree` широко применяются пути и вообще не используются массивы узлов.

Допустим, дерево находится на прокручиваемой панели. После развертывания дерева новый узел все еще может быть за пределами текущего окна просмотра. Для перехода к новому узлу следует вместо метода `makeVisible()` вызвать метод `scrollPathToVisible()`, как показано ниже. Этот метод развертывает все узлы, указанные в заданном пути, прокручивая содержимое окна вплоть до последнего узла в конце пути (рис. 6.29).

```
tree.scrollPathToVisible(path);
```

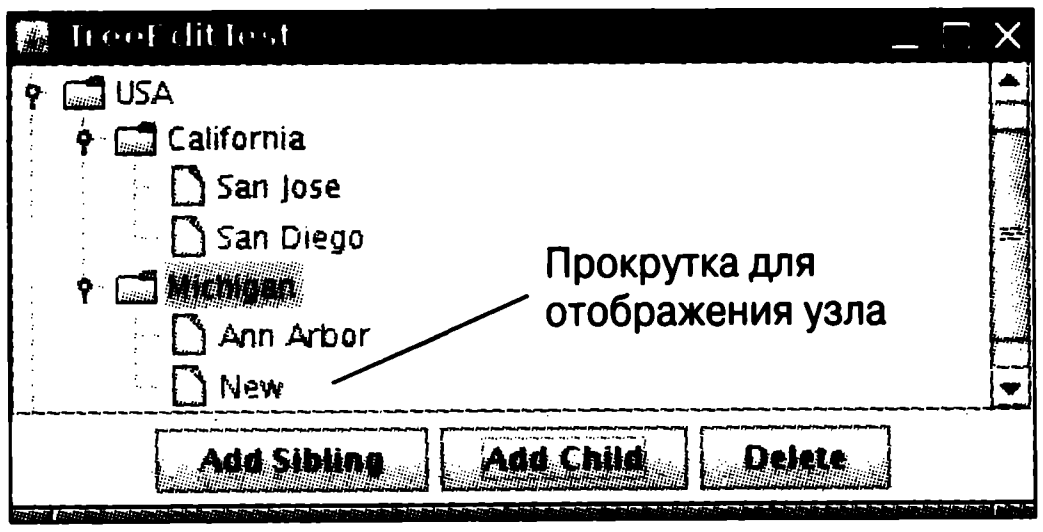


Рис. 6.29. Автоматическая прокрутка дерева на панели для просмотра нового узла

По умолчанию узлы дерева не редактируются, но если вызвать метод `setEditable()`, то пользователь сможет затем отредактировать любой узел дерева, дважды щелкнув на нем, внеся нужные правки и нажав клавишу <Enter>, чтобы зафиксировать результаты правки в данном узле.

После двойного щелчка мышью откроется показанное на рис. 6.30 окно редактора ячеек, вызываемого по умолчанию. Для его реализации служит класс `DefaultCellEditor`. А для редактирования узлов с иными объектами, кроме символьных строк, можно установить другие редакторы ячеек. Это делается аналогично установке редакторов ячеек таблицы, как пояснялось ранее в данной главе.

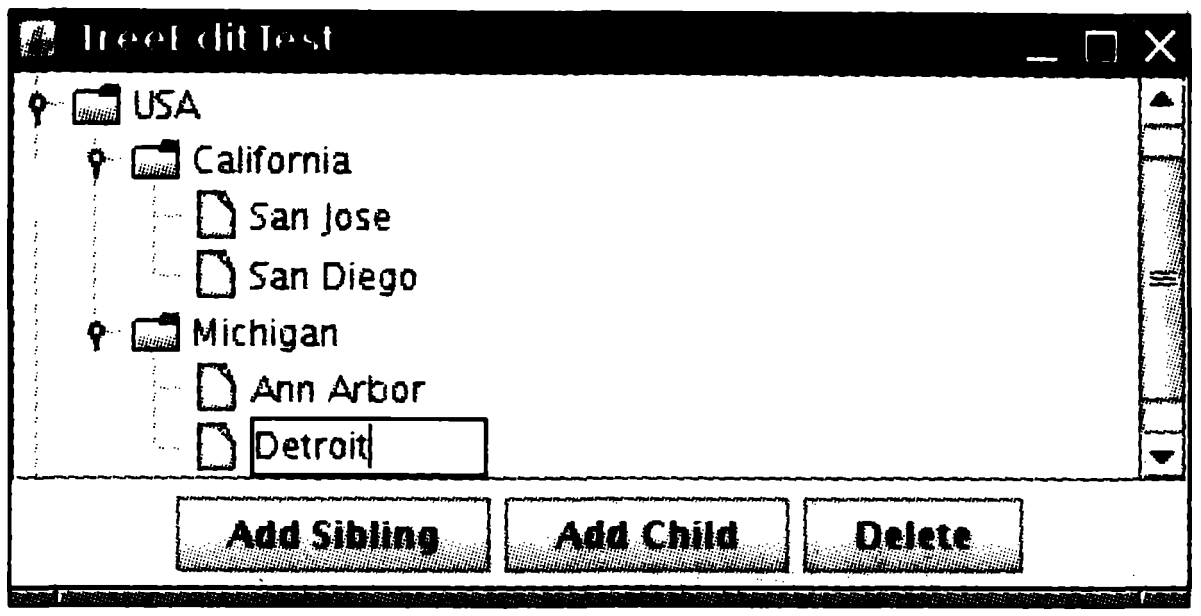


Рис. 6.30. Используемый по умолчанию редактор ячеек дерева

В листинге 6.13 приведен весь исходный код рассматриваемого здесь примера программы редактирования отдельных узлов дерева. Запустите эту программу на выполнение, создайте несколько узлов и отредактируйте их, дважды щелкнув на имени узла. Убедитесь в том, что свернутые деревья разворачиваются и содержимое окна прокручивается для отображения нового дочернего узла.

Листинг 6.13. Исходный код из файла `treeEdit/TreeEditFrame.java`

```
1   package treeEdit;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6   import javax.swing.tree.*;
7
8   /**
9    * Фрейм с деревом и кнопками для его редактирования
10  */
11  public class TreeEditFrame extends JFrame
12  {
13    private static final int DEFAULT_WIDTH = 400;
14    private static final int DEFAULT_HEIGHT = 200;
15
16    private DefaultTreeModel model;
17    private JTree tree;
18
19    public TreeEditFrame()
20    {
21      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
22
23      // построить дерево
24
25      TreeNode root = makeSampleTree();
26      model = new DefaultTreeModel(root);
27      tree = new JTree(model);
```

```
28        tree.setEditable(true);
29
30        // ввести прокручиваемую панель с деревом
31
32        JScrollPane scrollPane = new JScrollPane(tree);
33        add(scrollPane, BorderLayout.CENTER);
34
35        makeButtons();
36     }
37
38     public TreeNode makeSampleTree()
39     {
40        DefaultMutableTreeNode root =
41           new DefaultMutableTreeNode("World");
42        DefaultMutableTreeNode country =
43           new DefaultMutableTreeNode("USA");
44        root.add(country);
45        DefaultMutableTreeNode state =
46           new DefaultMutableTreeNode("California");
47        country.add(state);
48        DefaultMutableTreeNode city =
49           new DefaultMutableTreeNode("San Jose");
50        state.add(city);
51        city = new DefaultMutableTreeNode("San Diego");
52        state.add(city);
53        state = new DefaultMutableTreeNode("Michigan");
54        country.add(state);
55        city = new DefaultMutableTreeNode("Ann Arbor");
56        state.add(city);
57        country = new DefaultMutableTreeNode("Germany");
58        root.add(country);
59        state = new DefaultMutableTreeNode("Schleswig-Holstein");
60        country.add(state);
61        city = new DefaultMutableTreeNode("Kiel");
62        state.add(city);
63        return root;
64     }
65
66     /**
67      * Создает кнопки для ввода родственных, дочерних узлов и их удаления
68     */
69     public void makeButtons()
70     {
71        JPanel panel = new JPanel();
72        JButton addSiblingButton = new JButton("Add Sibling");
73        addSiblingButton.addActionListener(new ActionListener()
74           {
75              public void actionPerformed(ActionEvent event)
76              {
77                 DefaultMutableTreeNode selectedNode =
78                    (DefaultMutableTreeNode)
79                     tree.getLastSelectedPathComponent();
80
81                 if (selectedNode == null) return;
82
83                 DefaultMutableTreeNode parent =
84                    (DefaultMutableTreeNode) selectedNode.getParent();
85
```

```
86              if (parent == null) return;
87
88              DefaultMutableTreeNode newNode =
89                 new DefaultMutableTreeNode("New");
90
91              int selectedIndex = parent.getIndex(selectedNode);
92              model.insertNodeInto(newNode, parent, selectedIndex + 1);
93
94              // отобразить теперь новый узел
95
96              TreeNode[] nodes = model.getPathToRoot(newNode);
97              TreePath path = new TreePath(nodes);
98              tree.scrollPathToVisible(path);
99            }
100         });
101       panel.add(addSiblingButton);
102       JButton addChildButton = new JButton("Add Child");
103       addChildButton.addActionListener(new ActionListener()
104         {
105           public void actionPerformed(ActionEvent event)
106           {
106             DefaultMutableTreeNode selectedNode =
107                (DefaultMutableTreeNode)
108                 tree.getLastSelectedPathComponent();
109                 if (selectedNode == null) return;
110
111             DefaultMutableTreeNode newNode =
112                new DefaultMutableTreeNode("New");
113             model.insertNodeInto(newNode, selectedNode,
114                selectedNode.getChildCount());
115
116             // отобразить теперь новый узел
117
118             TreeNode[] nodes = model.getPathToRoot(newNode);
119             TreePath path = new TreePath(nodes);
120             tree.scrollPathToVisible(path);
121           }
122         });
123       panel.add(addChildButton*);
124
125        JButton deleteButton = new JButton("Delete");
126       deleteButton.addActionListener(new ActionListener()
127         {
128           public void actionPerformed(ActionEvent event)
129           {
130             DefaultMutableTreeNode selectedNode =
131                (DefaultMutableTreeNode)
132                 tree.getLastSelectedPathComponent();
133
134             if (selectedNode != null && selectedNode.getParent() != null)
135                 model.removeNodeFromParent(selectedNode);
136           }
137         });
138       panel.add(deleteButton);
139       add(panel, BorderLayout.SOUTH);
140     }
141 }
```

javax.swing.JTree 1.2

- `TreePath getSelectionPath()`

  Получает путь к текущему выбранному узлу (или путь к первому выбранному узлу, если выбрано сразу несколько узлов). Возвращает пустое значение `null`, если ни один из узлов не выбран.

- `Object getLastSelectedPathComponent()`

  Получает объект, который представляет текущий выбранный узел (или первый выбранный узел, если выбрано сразу несколько узлов). Возвращает пустое значение `null`, если ни один из узлов не выбран.

- `void makeVisible(TreePath` *`path`*`)`

  Развертывает все узлы по заданному пути.

- `void scrollPathToVisible(TreePath` *`path`*`)`

  Развертывает все узлы по заданному пути и, если дерево находится на прокручиваемой панели, прокручивает дерево, чтобы обеспечить отображение последнего узла по заданному пути.

javax.swing.tree.TreePath 1.2

- `Object getLastPathComponent()`

  Получает последний объект по заданному пути.

javax.swing.tree.TreeNode 1.2

- `TreeNode getParent()`

  Возвращает родительский узел данного узла.

- `TreeNode getChildAt(int` *`index`*`)`

  Ищет дочерний узел по указанному индексу. Значение индекса должно находиться в пределах от **0** до **`getChildCount() - 1`**.

- `int getChildCount()`

  Возвращает количество дочерних узлов данного узла.

- `Enumeration children()`

  Возвращает объект типа `Enumeration` для перебора всех дочерних узлов данного узла.

javax.swing.tree.DefaultTreeModel 1.2

- `void insertNodeInto(MutableTreeNode` *`newChild`*`, MutableTreeNode` *`parent`*`, int` *`index`*`)`

  Вводит объект *`newChild`* в качестве дочернего узла в родительский узел *`parent`* по указанному индексу и уведомляет приемники событий от модели дерева.

- `void removeNodeFromParent(MutableTreeNode` *`node`*`)`

  Удаляет узел `node` из модели дерева и уведомляет приемники событий от этой модели.

- `void nodeChanged(TreeNode` *`node`*`)`

  Уведомляет приемники событий от модели дерева об изменениях в узле `node`.

- `void nodesChanged(TreeNode` *`parent`*`, int[]` *`changedChildIndexes`*`)`

  Уведомляет приемники событий от модели дерева об изменениях во всех дочерних узлах родительского узла *`parent`* по указанным индексам.

- `void reload()`

  Перезагружает все узлы в модели дерева. Эту операцию нужно выполнять только в тех случаях, когда узлы полностью изменились под внешним воздействием.

Перечисление узлов

Иногда требуется найти узел дерева, начиная с корневого узла и перебирая все дочерние узлы до тех пор, пока не будет найдет совпадающий. Для перебора узлов в классе `DefaultMutableTreeNode` предоставляется несколько служебных методов.

Методы `breadthFirstEnumeration()` и `depthFirstEnumeration()` возвращают объект типа `Enumeration`. У этого объекта имеется метод `nextElement()`, предназначенный для последовательного обращения ко всем дочерним узлам текущего узла. Если объект типа `Enumeration` получен с помощью метода `breadthFirstEnumeration()`, он представляет результаты обхода в ширину всех дочерних узлов текущего узла. А объект типа `Enumeration`, возвращаемый методом `depthFirstEnumeration()`, содержит результаты обхода всех дочерних узлов текущего узла в глубину. На рис. 6.31 схематически показаны оба способа обхода узлов дерева.

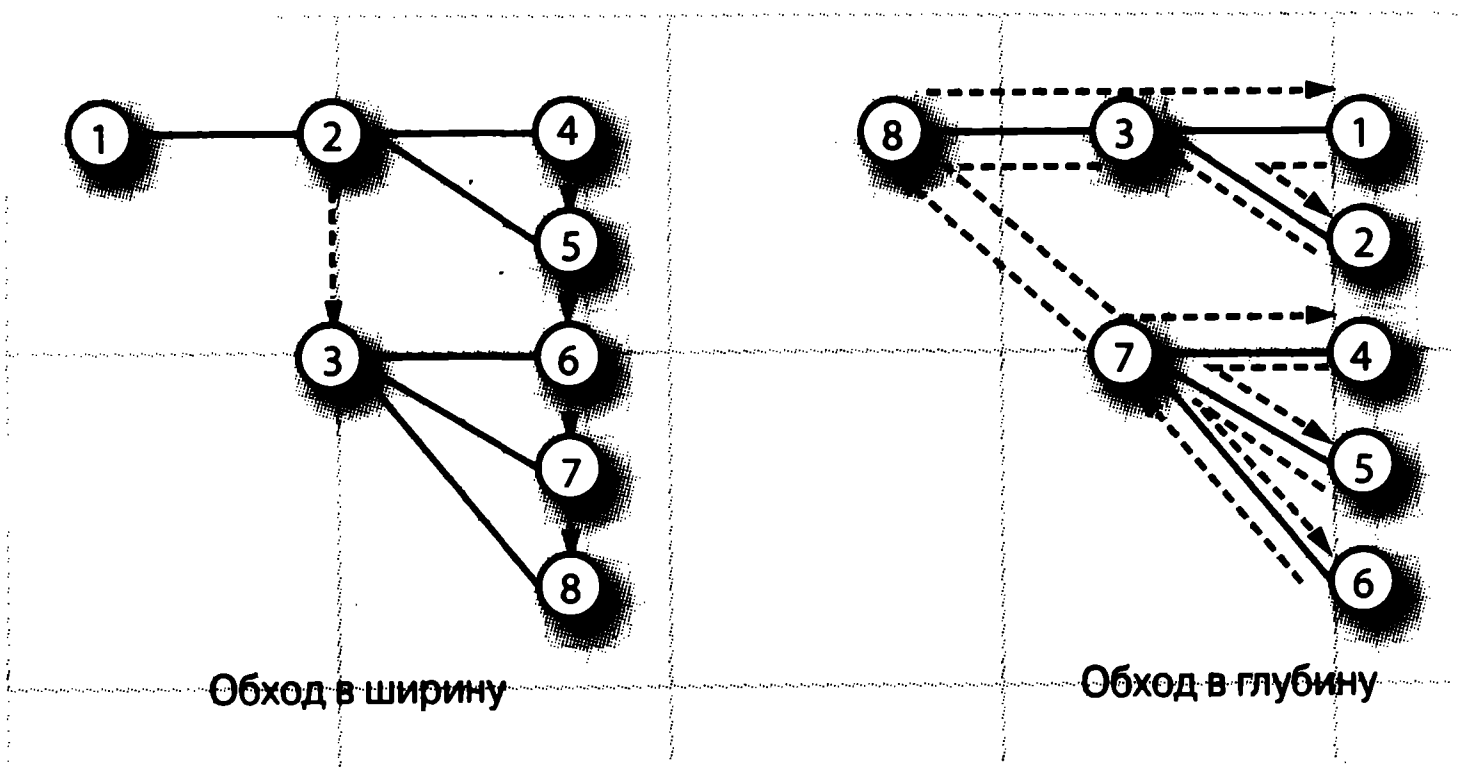


Рис. 6.31. Способы обхода узлов дерева

Обход узлов дерева в ширину выполняется послойно: сначала корневой узел, затем все его дочерние узлы, после чего все дочерние узлы этих дочерних узлов и т.д. Воспроизвести результаты такого обхода совсем не трудно.

А вот обход узлов дерева в глубину похож на поиск выхода из лабиринта. В этом случае обход выполняется по какому-то одному пути до тех пор, пока не будет достигнут листовой узел. Затем происходит возврат назад и выбор ближайшего нового

пути, после чего обход продолжается по этому же пути до тех пор, пока не будет достигнут листовой узел, и т.д.

Такой способ иначе называется *обходом в обратном порядке*, поскольку в ходе поиска сначала посещаются дочерние, а затем родительские узлы. Поэтому метод `postOrderTraversal()` действует подобно методу `depthFirstTraversal()`. Следует отметить и метод `preOrderTraversal()`, который также предназначен для обхода в глубину, но в получаемых результатах родительские узлы предшествуют дочерним. Ниже приведен типичный пример воплощения обхода дерева в ширину непосредственно в коде.

```
Enumeration breadthFirst = node.breadthFirstEnumeration();
while (breadthFirst.hasMoreElements())
    сделать что-нибудь с результатом вызова
    метода breadthFirst.nextElement();
```

И, наконец, метод `pathFromAncestorEnumeration()` предназначается для поиска пути от родительского узла к заданному и для перечисления узлов по этому пути. Принцип его работы основывается на вызове метода `getParent()` для данного узла до тех пор, пока не будет найден заданный родительский узел. После этого метод представляет путь в обратном порядке.

В следующем примере программы демонстрируется применение описанных выше методов для обхода дерева иерархического наследования классов. Как только в текстовом в поле, расположенном в нижней части рабочего окна данной программы, будет введено имя конкретного класса, он будет введен в дерево иерархического наследования со всеми своими суперклассами, как показано на рис. 6.32.

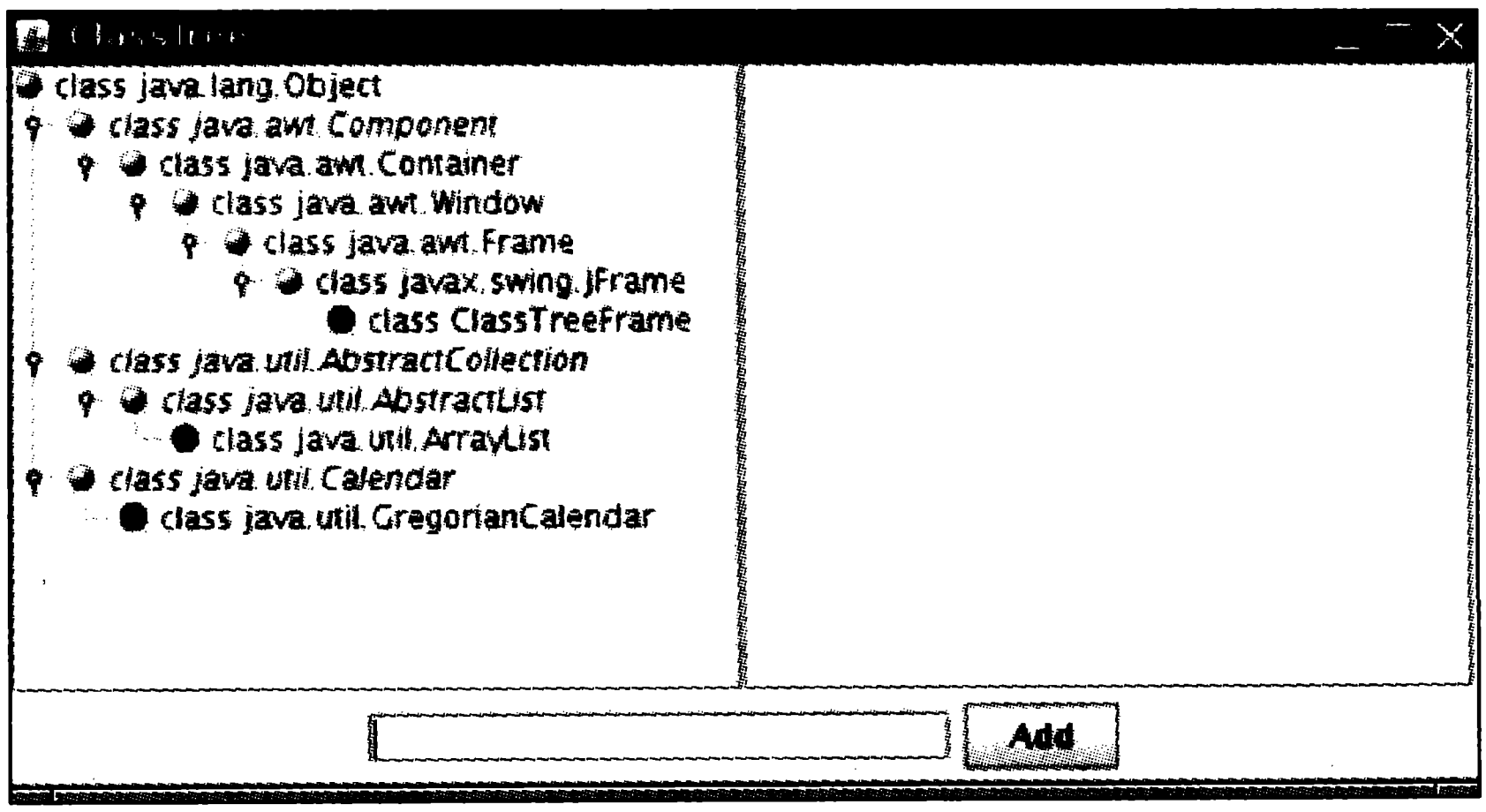


Рис. 6.32. Дерево иерархического наследования классов

В рассматриваемом здесь примере используется следующая важная особенность дерева: пользовательский объект узла может быть объектом произвольного типа. В данном примере узлы дерева представляют иерархическую структуру наследования классов, поэтому они относятся к одному типу `Class`. Во избежание дублирования необходимо организовать проверку наличия добавляемого класса в дереве. Это делается с помощью приведенного ниже метода.

```
public DefaultMutableTreeNode findUserObject(Object obj)
{
```

```
Enumeration e = root.breadthFirstEnumeration();
while (e.hasMoreElements())
{
  DefaultMutableTreeNode node =
     (DefaultMutableTreeNode) e.nextElement();
  if (node.getUserObject().equals(obj))
  return node;
  }
  return null;
}
```

Воспроизведение узлов

При разработке прикладных программ нередко возникает потребность изменить способ отображения узлов дерева, например, обозначить их другими пиктограммами или выделить их названия другим шрифтом. Подобные изменения можно вносить с помощью *средства воспроизведения ячеек дерева*. По умолчанию для воспроизведения узлов дерева в компоненте `JTree` используется класс `DefaultTreeCellRenderer`, который расширяет класс `JLabel`. Этот класс формирует метку, состоящую из пиктограммы и метки узла.

НА ЗАМЕТКУ! Средство воспроизведения ячеек дерева не отображает маркеры свертывания и развертывания подчиненных деревьев. Они являются частью общего визуального стиля, и поэтому изменять их не рекомендуется.

Специальную настройку воспроизведения узлов дерева можно произвести одним из трех способов.

- Заменить с помощью класса `DefaultTreeCellRenderer` пиктограммы, шрифты и цвета фона, применяемые во всех узлах дерева.
- Установить используемое по умолчанию средство воспроизведения, расширяющее класс `DefaultTreeCellRenderer`, а также применить разные пиктограммы, шрифты и цвета фона в разных узлах дерева.
- Установить средство воспроизведения, реализующее интерфейс `TreeCellRenderer`, чтобы воспроизводить в отдельных узлах дерева специально предусмотренные для них изображения.

Рассмотрим каждый из этих способов более подробно. Наиболее простой способ изменения пиктограммы, шрифта и цвета фона узлов состоит в том, чтобы построить объект типа `DefaultTreeCellRenderer` в качестве средства воспроизведения узлов дерева по умолчанию и установить его вместе с требуемыми пиктограммами объекта и дереве, как показано ниже. Результат выполнения этих действий приведен на рис. 6.32, где в качестве пиктограмм узлов используются изображения шариков.

```
DefaultTreeCellRenderer renderer = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif")); // используется
                                                      // для листовых узлов
renderer.setClosedIcon(new ImageIcon("red-ball.gif")); // используется
                                                      // для свернутых узлов
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif")); // используется
                                                     // для развернутых узлов
tree.setCellRenderer(renderer);
```

Изменять шрифт и цвет фона не рекомендуется, потому что в этом случае легко нарушить общий стиль оформления ГПИ. Изменять шрифт допускается только для выделения отдельных узлов. Так, на рис. 6.32 курсивом выделены абстрактные классы.

Для изменения внешнего вида отдельных узлов следует установить специальное средство воспроизведения ячеек дерева. Это средство во многом похоже на средство воспроизведения ячеек из списка, рассмотренное ранее в данной главе. У интерфейса `TreeCellRenderer` имеется следующий единственный метод `getTreeCellRendererComponent()`:

```
Component getTreeCellRendererComponent(JTree tree, Object value,
        boolean selected, boolean expanded, boolean leaf,
        int row, boolean hasFocus)
```

Метод `getTreeCellRendererComponent()` из класса `DefaultTreeCellRenderer` возвращает ссылку `this`, т.е. текущую метку. (Напомним, что класс `DefaultTreeCellRenderer` расширяет класс `JLabel`.) Для создания специального компонента, предназначенного для воспроизведения ячеек дерева, нужно сначала расширить класс `DefaultTreeCellRenderer`. Затем следует переопределить метод `getTreeCellRendererComponent()`, предусмотрев в нем вызов из суперкласса метода, который подготовит все данные, необходимые для создания метки. Кроме того, в данном методе задаются нужные значения свойств, а по завершении своего выполнения он возвращает логическое значение `this`.

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer
{
  public Component getTreeCellRendererComponent(JTree tree,
        Object value, boolean selected, boolean expanded,
        boolean leaf, int row, boolean hasFocus)
  {
     Component comp =
        super.getTreeCellRendererComponent(tree, value, selected,
                                    expanded, leaf, row, hasFocus);
     DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
     look at node.getUserObject();
     Font font = appropriate font;
     comp.setFont(font);
     return comp;
  }
};
```

ВНИМАНИЕ! Параметр `value` метода `getTreeCellRendererComponent()` является *узловым*, а *не* пользовательским объектом. Напомним, что пользовательский объект относится к классу узлов `DefaultMutableTreeNode`, а класс `JTree` может содержать узлы произвольного типа. Если в дереве используются узлы типа `DefaultMutableTreeNode`, то на втором этапе нужно извлечь пользовательский объект, как это было сделано в предыдущем примере программы.

ВНИМАНИЕ! В классе `DefaultTreeCellRenderer` используется один и тот же объект метки для всех узлов, но для каждого узла изменяется ее текст. Если изменить шрифт для выделения названия отдельного узла, то при последующем вызове упомянутого выше метода нужно восстановить используемый по умолчанию шрифт. В противном случае названия всех последующих узлов будут выделены новым шрифтом! Один из способов восстановления исходного шрифта представлен ниже, в листинге 6.14.

Здесь не приводится пример, демонстрирующий применение средства воспроизведения ячеек дерева для отображения произвольной графики. Если у вас возникнет потребность в этом, вы можете без особого труда приспособить под свои нужды средство воспроизведения ячеек из списка, представленное в листинге 6.4.

Средство воспроизведения типа `ClassNameTreeCellRenderer` из листинга 6.14 выделяет имя класса обычным шрифтом или курсивом в зависимости от наличия модификатора `ABSTRACT` у объекта типа `Class`. Для обозначения абстрактных классов не выбирается какой-то другой шрифт, чтобы не изменять визуальный стиль, обычно применяемый для отображения дерева. По этой причине курсив для обозначения абстрактного класса получается путем наклона начертания исходного шрифта, которым выделяются метки. Напомним, что в результате всех вызовов возвращается только один общий объект типа `JLabel`. При последующих вызовах метода `getTreeCellRendererComponent()` необходимо снова вернуться к исходному шрифту. Обратите также внимание на то, каким образом изменяются пиктограммы узлов в конструкторе класса `ClassTreeFrame`.

javax.swing.tree.DefaultMutableTreeNode 1.2

- `Enumeration breadthFirstEnumeration()`
- `Enumeration depthFirstEnumeration()`
- `Enumeration preOrderEnumeration()`
- `Enumeration postOrderEnumeration()`

  Возвращают объект типа `Enumeration`, представляющий результаты обхода всех узлов в модели дерева. При обходе в ширину дочерние узлы, находящиеся ближе к корневому узлу, посещаются раньше. При обходе в глубину перед переходом к равноправному узлу посещаются все дочерние узлы. Метод `postOrderEnumeration()` является аналогом метода `depthFirstEnumeration()`. Обход в ширину (или в прямом порядке) подобен обходу в глубину (или в обратном порядке) за исключением того, что родительские узлы перечисляются раньше дочерних.

javax.swing.tree.TreeCellRenderer 1.2

- `Component getTreeCellRendererComponent(JTree` *`tree`*`, Object` *`value`*`, boolean` *`selected`*`, boolean` *`expanded`*`, boolean` *`leaf`*`, int` *`row`*`, boolean` *`hasFocus`*`)`

  Возвращает компонент, метод `paint()` которого вызывается для воспроизведения ячейки (или узла) дерева.

Параметры:		
	`tree`	Дерево, содержащее воспроизводимый узел
	`value`	Воспроизводимый узел
	`selected`	Принимает логическое значение `true`, если данный узел выбран

expanded	Принимает логическое значение `true`, если видны дочерние узлы данного узла
leaf	Принимает логическое значение `true`, если данный узел должен быть отображен как лист
row	Отображаемая строка, содержащая узел
hasFocus	Принимает логическое значение `true`, если данный узел обладает фокусом ввода

javax.swing.tree.DefaultTreeCellRenderer 1.2

- `void setLeafIcon(Icon icon)`
- `void setOpenIcon(Icon icon)`
- `void setClosedIcon(Icon icon)`

  Задают пиктограмму для обозначения листового, развернутого или свернутого узла.

Обработка событий в деревьях

Дерево чаще всего используется совместно с каким-нибудь другим компонентом. Например, при выборе какого-нибудь узла дерева в соседнем окне может быть показана определенная сопроводительная информация. На рис. 6.33 представлен пример программы, в которой для каждого узла в правом текстовом окне автоматически отображаются статические переменные и переменные экземпляра соответствующего класса.

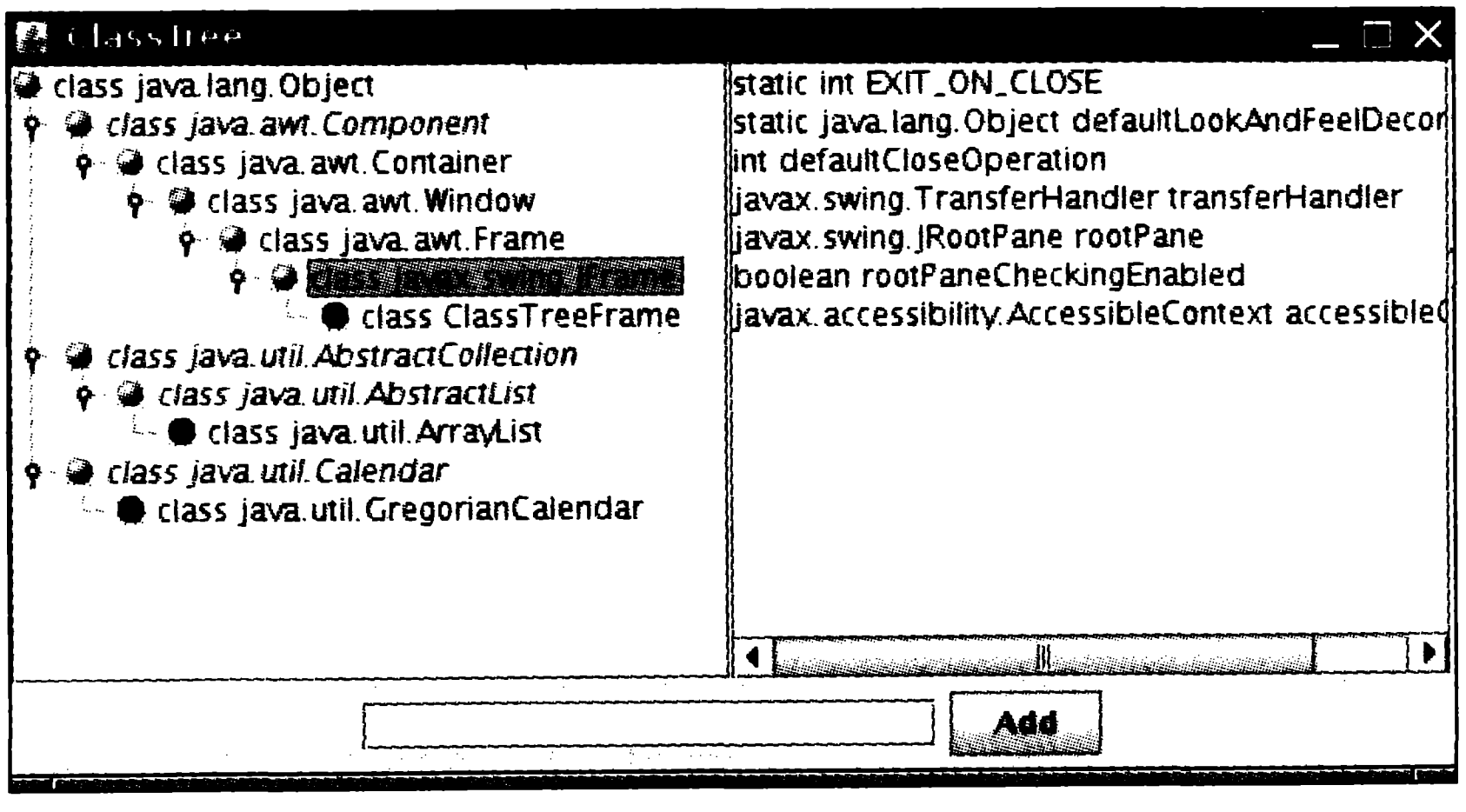


Рис. 6.33. Пример программы для просмотра классов

Для того чтобы добиться такого поведения, нужно установить *приемник событий выбора из дерева*. Класс такого приемника событий должен реализовать интерфейс `TreeSelectionListener` со следующим единственным методом `valueChanged()`:

```
void valueChanged(TreeSelectionEvent event)
```

Метод `valueChanged()` вызывается при каждом событии выбора или отмены выбора узлов дерева. Приемник событий добавляется к дереву обычным образом:

```
tree.addTreeSelectionListener(listener);
```

С помощью свойств модели типа `TreeSelectionModel` можно специально указать конкретный режим выбора узлов дерева: только один узел (свойство `SINGLE_TREE_SELECTION`), группа смежных узлов (свойство `CONTINGUOUS_TREE_SELECTION`) или произвольная группа несмежных узлов (свойство `DISCONTINGUOUS_TREE_SELECTION`). По умолчанию допускается выбор произвольной группы несмежных узлов. В рассматриваемом здесь примере программы для просмотра классов допускается выбор только одного узла дерева, как показано ниже. Никаких других действий, кроме указания конкретного режима выбора, предпринимать не нужно.

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

НА ЗАМЕТКУ! Порядок выбора нескольких элементов зависит от конкретного визуального стиля ГПИ. Так, если применяется визуальный стиль Metal, для выбора несмежных элементов (в данном случае — узлов дерева) следует нажать клавишу <Ctrl> и, не отпуская ее, щелкнуть по очереди на каждом выбираемом элементе, а для выбора нескольких смежных элементов — нажать клавишу <Shift> и, не отпуская ее, щелкнуть сначала на первом, а затем на последнем элементе из выбираемой группы.

Для того чтобы выяснить, что именно выбрано в настоящий момент, нужно запросить дерево, вызвав метод `getSelectionPaths()` следующим образом:

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

Если же требуется ограничить возможности пользователя, разрешив ему выбирать узлы дерева только по одному, то для этой цели удобнее воспользоваться служебным методом `getSelectionPath()`, который в общем случае возвращает первый же выбранный путь к элементу или пустое значение `null`, если такой путь не выбран.

ВНИМАНИЕ! В классе `TreeSelectionEvent` имеется метод `getPaths()`, который возвращает массив объектов типа `TreePath`, но этот массив описывает *изменения* в самом выборе, а *не* текущий результат выбора.

В листинге 6.14 приведен исходный код класса фрейма для программы, отображающей иерархию наследования классов в виде древовидной структуры, где абстрактные классы выделены курсивом. А в листинге 6.15 представлен исходный код, реализующий средство воспроизведения ячеек дерева. Если ввести полное имя класса и нажать клавишу <Enter> или щелкнуть на кнопке Add, в дерево будет введен новый класс со всеми его суперклассами. В полное имя класса следует непременно включить имя пакета, как, например, `java.util.ArrayList`.

Рассматриваемая здесь программа немного сложна для понимания, поскольку в ней для построения дерева классов используется механизм рефлексии. Весь код построения дерева классов находится в методе `addClass()`. (Иерархия наследования классов используется здесь лишь в качестве удобного примера для демонстрации особенностей построения деревьев и обращения с ними, не усложняя программирование. При разработке собственных прикладных программ вы можете использовать любые другие источники иерархических данных.) Для проверки наличия вводимого класса в дереве вызывается метод `findUserObject()`, реализация которого описана в предыдущем разделе. В этом методе реализуется алгоритм обхода дерева в ширину. Если класс отсутствует в дереве, то сначала вводятся его суперклассы, а затем создается и отображается новый узел для данного класса.

Когда выбран узла дерева, текстовая область справа заполняется полями выбранного класса. В конструкторе фрейма накладываются ограничения, разрешающие пользователю данной программы выбирать узлы дерева только по одному, а также вводится приемник событий выбора из дерева. При вызове метода `valueChanged()` его параметр события игнорируется, а у дерева запрашивается только путь к текущему выбранному узлу. Далее нужно, как обычно, получить последний узел по заданному пути и извлечь из него пользовательский объект. После этого вызывается метод `getFieldDescription()`, в котором для формирования символьной строки со всеми полями выбранного класса применяется механизм рефлексии.

Листинг 6.14. Исходный код из файла `treeRender/ClassTreeFrame.java`

```
1   package treeRender;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.lang.reflect.*;
6   import java.util.*;
7   import javax.swing.*;
8   import javax.swing.event.*;
9   import javax.swing.tree.*;
10
11  /**
12   * В этом фрейме отображается дерево иерархии классов, текстовое
13   * поле и кнопка Add для ввода новых классов в дерево
14  */
15  public class ClassTreeFrame extends JFrame
16  {
17    private static final int DEFAULT_WIDTH = 400;
18    private static final int DEFAULT_HEIGHT = 300;
19
20    private DefaultMutableTreeNode root;
21    private DefaultTreeModel model;
22    private JTree tree;
23    private JTextField textField;
24    private JTextArea textArea;
25
26    public ClassTreeFrame()
27    {
28      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
29
30      // в корне дерева иерархии классов находится класс Object
```

```
31      root = new DefaultMutableTreeNode(java.lang.Object.class);
32      model = new DefaultTreeModel(root);
33      tree = new JTree(model);
34
35      // ввести этот класс, чтобы заполнить дерево некоторыми данными
36      addClass(getClass());
37
38      // установить пиктограммы для обозначения узлов
39      ClassNameTreeCellRenderer renderer =
40         new ClassNameTreeCellRenderer();
41      renderer.setClosedIcon(
42         new ImageIcon(getClass().getResource("red-ball.gif")));
43      renderer.setOpenIcon(
44         new ImageIcon(getClass().getResource("yellow-ball.gif")));
45      renderer.setLeafIcon(
46         new ImageIcon(getClass().getResource("blue-ball.gif")));
47      tree.setCellRenderer(renderer);
48
49      // установить режим выбора узлов дерева
50      tree.addTreeSelectionListener(new TreeSelectionListener()
51        {
52          public void valueChanged(TreeSelectionEvent event)
53          {
54            // пользователь выбрал другой узел — обновить его описание
55            TreePath path = tree.getSelectionPath();
56            if (path == null) return;
57            DefaultMutableTreeNode selectedNode =
58               (DefaultMutableTreeNode) path.getLastPathComponent();
59            Class<?> c = (Class<?>) selectedNode.getUserObject();
60            String description = getFieldDescription(c);
61            textArea.setText(description);
62          }
63        });
64      int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
65      tree.getSelectionModel().setSelectionMode(mode);
66
67      // в этой текстовой области находится описание класса
68      textArea = new JTextArea();
69
70      // добавить дерево и текстовую область
71      JPanel panel = new JPanel();
72      panel.setLayout(new GridLayout(1, 2));
73      panel.add(new JScrollPane(tree));
74      panel.add(new JScrollPane(textArea));
75
76      add(panel, BorderLayout.CENTER);
77
78      addTextField();
79    }
80
81    /**
82     * Добавляет в фрейм текстовое поле и кнопку Add
83     * для ввода нового класса
84    */
85    public void addTextField()
86    {
87      JPanel panel = new JPanel();
```

```
88
89      ActionListener addListener = new ActionListener()
90        {
91          public void actionPerformed(ActionEvent event)
92          {
93            // ввести класс, имя которого находится в текстовом поле
94            try
95            {
96              String text = textField.getText();
97              addClass(Class.forName(text)); // очистить текстовое поле,
98                          // чтобы указать на удачный исход ввода класса
99              textField.setText("");
100           }
101           catch (ClassNotFoundException e)
102           {
103             JOptionPane.showMessageDialog(null, "Class not found");
104           }
105         }
106       };
107
108     // в этом текстовом поле вводятся имена новых классов
109     textField = new JTextField(20);
110     textField.addActionListener(addListener);
111     panel.add(textField);
112
113     JButton addButton = new JButton("Add");
114     addButton.addActionListener(addListener);
115     panel.add(addButton);
116
117     add(panel, BorderLayout.SOUTH);
118   }
119
120   /**
121    * Находит искомый объект в дереве
122    * @param obj Искомый объект
123    * @return Возвращает узел, содержащий объект, или пустое
124     *         значение null, если объект отсутствует в дереве
125   */
126   @SuppressWarnings("unchecked")
127   public DefaultMutableTreeNode findUserObject(Object obj)
128   {
129     // найти узел, содержащий пользовательский объект
130     Enumeration<TreeNode> e =
131        (Enumeration<TreeNode>) root.breadthFirstEnumeration();
132     while (e.hasMoreElements())
133     {
134       DefaultMutableTreeNode node =
135          (DefaultMutableTreeNode) e.nextElement();
136       if (node.getUserObject().equals(obj)) return node;
137     }
138     return null;
139   }
140
141   /**
142    * Вводит новый класс и любые его родительские классы,
143    * пока еще отсутствующие в дереве
144    * @param c Вводимый класс
```

```
145    * @return Возвращает узел с вновь введенным классом
146    */
147   public DefaultMutableTreeNode addClass(Class<?> c)
148   {
149     // ввести новый класс в дерево
150
151     // пропустить типы данных, не относящиеся к классам
152     if (c.isInterface() || c.isPrimitive()) return null;
153
154     // если класс уже присутствует в дереве, возвратить его узел
155     DefaultMutableTreeNode node = findUserObject(c);
156     if (node != null) return node;
157
158     // класс отсутствует в дереве — ввести сначала его родительские
159     // классы рекурсивным способом
160     Class<?> s = c.getSuperclass();
161     DefaultMutableTreeNode parent;
162     if (s == null) parent = root;
163     else parent = addClass(s);
164
165     // ввести затем класс в качестве потомка его родительского класса
166     DefaultMutableTreeNode newNode = new DefaultMutableTreeNode(c);
167     model.insertNodeInto(newNode, parent, parent.getChildCount());
168
169     // сделать видимым узел с вновь введенным классом
170     TreePath path = new TreePath(model.getPathToRoot(newNode));
171     tree.makeVisible(path);
172
173     return newNode;
174   }
175
176   /**
177    * Возвращает описание полей класса
178    * @param Описываемый класс
179    * @return Возвращает символьную строку, содержащую все
180    *         типы и имена полей описываемого класса
181   */
182   public static String getFieldDescription(Class<?> c)
183   {
184     // использовать рефлексию для обнаружения типов и имен полей
185     StringBuilder r = new StringBuilder();
186     Field[] fields = c.getDeclaredFields();
187     for (int i = 0; i < fields.length; i++)
188     {
189       Field f = fields[i];
190       if ((f.getModifiers() & Modifier.STATIC) != 0)
191          r.append("static ");
192       r.append(f.getType().getName());
193       r.append(" ");
194       r.append(f.getName());
195       r.append("\n");
196     }
197     return r.toString();
198   }
199 }
```

Листинг 6.15. Исходный код из файла `treeRender/ClassNameTreeCellRenderer.java`

```
1  package treeRender;
2
3  import java.awt.*;
4  import java.lang.reflect.*;
5  import javax.swing.*;
6  import javax.swing.tree.*;
7  /**
8   * Этот класс воспроизводит имя класса, выделяя его простым шрифтом
9   * или курсивом. Абстрактные классы выделяются только курсивом
10 */
11 public class ClassNameTreeCellRenderer extends DefaultTreeCellRenderer
12 {
13   private Font plainFont = null;
14   private Font italicFont = null;
15
16   public Component getTreeCellRendererComponent(
17      JTree tree, Object value, boolean selected, boolean expanded,
18      boolean leaf, int row, boolean hasFocus)
19   {
20     super.getTreeCellRendererComponent(
21        tree, value, selected, expanded, leaf, row, hasFocus);
22     // получить пользовательский объект
23     DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
24     Class<?> c = (Class<?>) node.getUserObject();
25
26     // сначала сделать простой шрифт наклонным
27     if (plainFont == null)
28     {
29       plainFont = getFont();
30       // средство воспроизведения ячеек дерева иногда
31       // вызывается с меткой, имеющей пустой шрифт
32       if (plainFont != null) italicFont =
33           plainFont.deriveFont(Font.ITALIC);
34     }
35
36     // установить наклонный шрифт, если класс является
37     // абстрактным, а иначе — простой шрифт
38     if ((c.getModifiers() & Modifier.ABSTRACT) == 0) setFont(plainFont);
39     else setFont(italicFont);
40     return this;
41   }
42 }
```

javax.swing.JTree 1.2

- `TreePath getSelectionPath()`
- `TreePath[] getSelectionPaths()`

  Возвращают путь к первому выбранному узлу или массив путей ко всем выбранным узлам дерева. Если ни один из узлов дерева не выбран, оба метода возвращают пустое значение `null`.

javax.swing.event.TreeSelectionListener 1.2

- `void valueChanged(TreeSelectionEvent event)`

  Вызывается всякий раз, когда происходит выбор или отмена выбора узлов дерева.

javax.swing.event.TreeSelectionEvent 1.2

- `TreePath getPath()`
- `TreePath[] getPaths()`

  Получают первый путь или все пути, которые *изменились* в результате данного события выбора из дерева. Для получения сведений о текущем выборе, а не изменении выбора следует вызывать метод `JTree.getSelectionPath()`.

Специальные модели деревьев

В качестве последнего примера обращения с деревьями рассмотрим программу, которая, подобно отладчику, проверяет содержимое переменной или объекта. На рис. 6.34 показано рабочее окно данной программы.

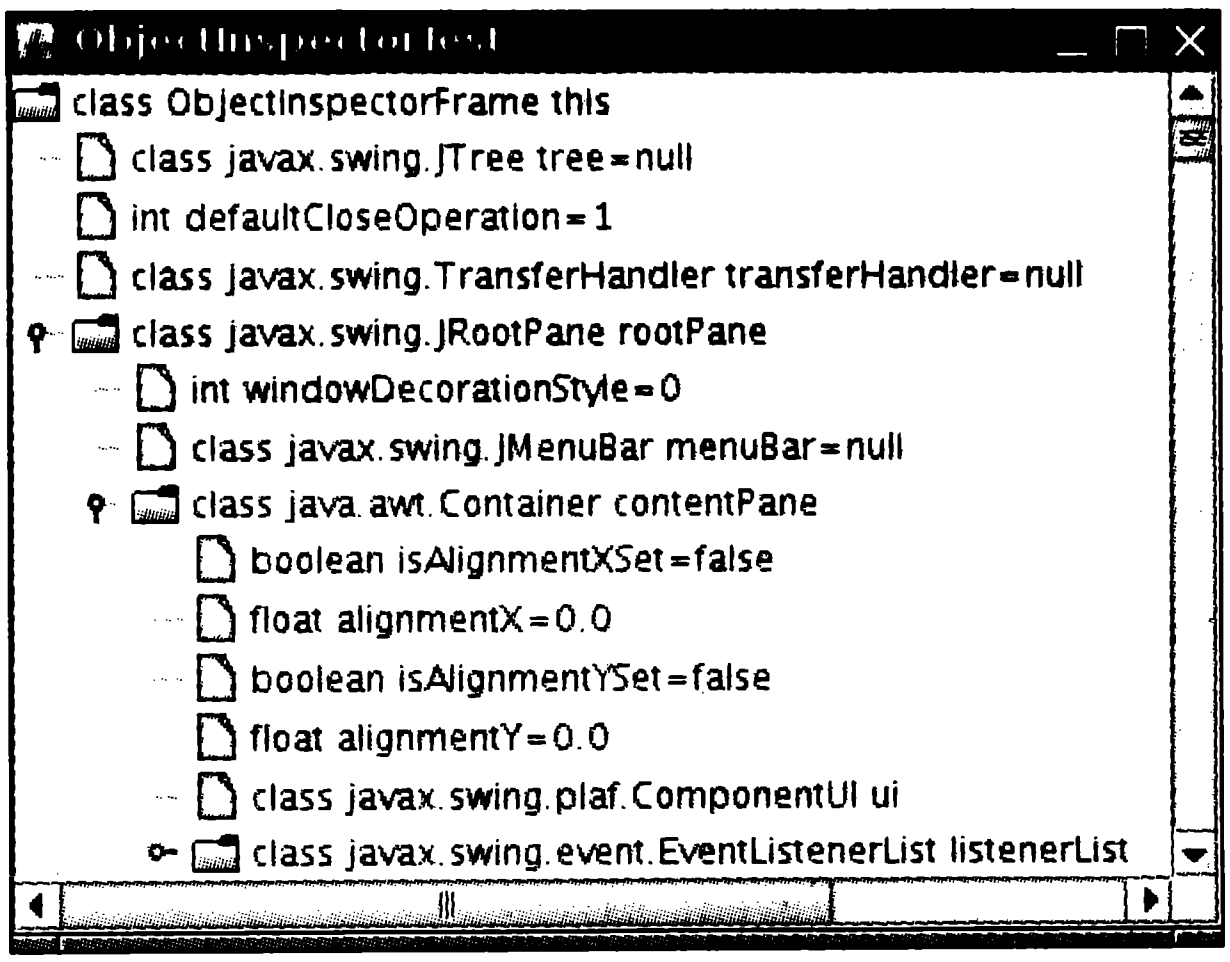


Рис. 6.34. Пример дерева с проверяемыми объектами

Скомпилируйте и запустите эту программу на выполнение, прежде чем читать ее описание. Каждый узел дерева соответствует какому-нибудь полю экземпляра. Если поле содержит объект, разверните структуру этого объекта для просмотра полей экземпляра его класса. Как видите, данная программа позволяет просматривать содержимое своего фрейма. При более внимательном изучении нескольких полей экземпляра вы можете обнаружить некоторые уже знакомые вам классы. Кроме того, эта программа позволяет оценить, насколько сложны компоненты библиотеки Swing, предназначенные для построения ГПИ.

Данная программа примечательна тем, что для построения дерева в ней не используется модель типа `DefaultTreeModel`. При наличии данных, которые уже иерархически организованы в виде древовидной структуры, вряд ли имеет смысл использовать предлагаемое по умолчанию дерево и синхронизировать оба дерева. Именно такая ситуация возникает в рассматриваемом здесь примере: проверяемые объекты уже связаны друг с другом с помощью ссылок на них, и поэтому нет никакой нужды дублировать структуру их связей.

В интерфейсе `TreeModel` объявлено лишь несколько удобных методов. Первая группа методов позволяет найти узлы дерева, начиная с корня и продолжая дочерними узлами. Эти методы вызываются в классе `JTree` только в том случае, если узел был развернут пользователем, как показано ниже.

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

В рассматриваемом здесь примере наглядно демонстрируется, почему для интерфейса `TreeModel`, как и для класса `JTree`, не требуется явное обозначение узлов. Как видите, корневые и дочерние узлы могут быть объектами произвольного типа. Интерфейс `TreeModel` отвечает за уведомление класса `JTree` об установлении связи между ними. Следующий метод выполняет действия, обратные методу `getChild()`:

```
int getIndexOfChild(Object parent, Object child)
```

Как следует из листинга 6.16, этот метод можно реализовать на основе первых трех методов. Модель дерева сообщает компоненту `JTree`, какие именно узлы следует отобразить как листовые, следующим образом:

```
boolean isLeaf(Object node)
```

Если при выполнении кода изменяется модель дерева, то дерево нужно уведомить о необходимости его перерисовки. Поэтому дерево вводится в модель как приемник событий типа `TreeModelListener`. Следовательно, в модели должны использоваться обычные методы управления обработчиками событий, как показано ниже, а их реализация представлена в листинге 6.17.

```
void addTreeModelListener(TreeModelListener l)
void removeTreeModelListener(TreeModelListener l)
```

Для изменения содержимого дерева в его модели вызывается один из четырех перечисленных ниже методов из интерфейса `TreeModelListener`.

```
void treeNodesChanged(TreeModelEvent e)
void treeNodesInserted(TreeModelEvent e)
void treeNodesRemoved(TreeModelEvent e)
void treeStructureChanged(TreeModelEvent e)
```

Объект типа `TreeModelEvent` описывает место, где происходят изменения в дереве. Подробности организации событий, наступающих в модели дерева при вводе и удалении узлов, здесь не рассматриваются в силу того, что они носят слишком технический характер. Вам нужно лишь позаботиться об инициировании подобных событий, когда в дереве вводятся и удаляются отдельные узлы. В листинге 6.16 демонстрируется пример инициирования события, наступающего при замене корневого узла новым объектом.

СОВЕТ. Для упрощения кода инициирования событий рекомендуется служебный класс `javax.swing.EventListenerList`, который составляет список из приемников событий. На примере трех последних методов из листинга 6.17 показано, как пользоваться этим классом в прикладном коде.

И наконец, если пользователь редактирует узел дерева, то его модель вызывается с помощью следующего метода, где указывается вносимое изменение:

```
void valueForPathChanged(TreePath path, Object newValue)
```

Если же редактирование узлов не разрешается, этот метод вообще не вызывается. В таком случае построить модель дерева много проще. Для этого достаточно реализовать три метода:

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

Эти методы описывают структуру дерева. После реализации остальных пяти методов, как показано в листинге 6.16, можно приступать к отображению дерева.

А теперь перейдем непосредственно к реализации рассматриваемого здесь примера программы. В ней строится дерево, состоящее из объектов типа `Variable` в его узлах.

НА ЗАМЕНТКУ! Если бы в данном примере использовалась модель дерева типа `DefaultTreeModel`, то узлы были бы объектами типа `DefaultMutableTreeNode` с *пользовательскими объектами* типа `Variable`.

Допустим, требуется проверить переменную, объявленную следующим образом:

```
Employee joe;
```

Эта переменная имеет *тип* `Employee.class`, *имя* `joe` и *значение* в виде ссылки на объект `joe`. В листинге 6.18 класс `Variable`, описывающий эту переменную в рассматриваемом здесь примере программы, определяется следующим образом:

```
Variable v = new Variable(Employee.class, "joe", joe);
```

Если переменная относится к примитивному типу, то для ее значения придется создать объектную оболочку:

```
new Variable(double.class, "salary", new Double(salary));
```

Если же переменная относится к типу класса, то она имеет *поля*, которые можно перечислить и собрать в объекте типа `ArrayList` с помощью механизма рефлексии. Метод `getFields()` из класса `Class` не возвращает поля суперкласса, поэтому данный метод придется вызвать для всех суперклассов проверяемого класса. Для этой цели служит исходный код, находящийся в конструкторе класса `Variable`. Метод `getFields()` из класса `Variable` возвращает массив полей. И наконец, метод `toString()` из класса `Variable` форматирует метку узла дерева. Метка всегда содержит тип и имя переменной, а если переменная не относится к типу класса, то приводится также ее значение.

НА ЗАМЕТКУ! Если тип переменной представлен массивом, то его элементы в данном примере не отображаются. Впрочем, сделать это совсем не трудно, и поэтому мы предоставляем такую возможность читателю в качестве дополнительного упражнения.

Перейдем непосредственно к модели дерева. Исходный код первых двух ее методов несложен:

```
public Object getRoot()
{
  return root;
}

public int getChildCount(Object parent)
{
  return ((Variable) parent).getFields().size();
}
```

Метод `getChild()` возвращает новый объект типа `Variable`, описывающий поле по указанному индексу. С помощью методов `getType()` и `getName()` из класса `Field` можно получить тип и имя поля, а с помощью механизма рефлексии — прочитать значение поля, вызвав метод `f.get(parentValue)`. Этот метод может сгенерировать исключение типа `IllegalAccessException`. Но в конструкторе класса `Variable` предоставлен свободный доступ ко всем полям, поэтому подобная исключительная ситуация вряд ли вообще возникнет. Ниже приведен весь исходный код метода `getChild()`.

```
public Object getChild(Object parent, int index)
{
  ArrayList fields = ((Variable) parent).getFields();
  Field f = (Field) fields.get(index);
  Object parentValue = ((Variable) parent).getValue();
  try
  {
    return new Variable(f.getType(), f.getName(), f.get(parentValue));
  }
  catch (IllegalAccessException e)
  {
    return null;
  }
}
```

Упомянутые выше три метода выявляют структуру дерева объектов для компонента `JTree`. А остальные методы выполняют другие рутинные операции, как следует из листинга 6.17.

Необходимо отметить следующую особенность рассматриваемой здесь модели дерева: она описывает *бесконечное* дерево. Это можно проверить на примере одного из объектов типа `WeakReference`. Так, если щелкнуть в дереве на имени переменной `referent`, произойдет возврат к исходному объекту, который содержит идентичное подчиненное дерево с объектом типа `WeakReference`. Разумеется, сохранить бесконечное количество узлов невозможно, поэтому данная модель дерева формирует дочерние узлы по требованию в процессе постепенного развертывания родительских узлов. Исходный код класса фрейма для данной программы представлен в листинге 6.16.

Листинг 6.16. Исходный код из файла `treeModel/ObjectInspectorFrame.java`

```
1  package treeModel;
2
3  import java.awt.*;
4  import javax.swing.*;
```

```
5
6  /**
7   * Этот фрейм содержит дерево объектов
8  */
9  public class ObjectInspectorFrame extends JFrame
10 {
11   private JTree tree;
12   private static final int DEFAULT_WIDTH = 400;
13   private static final int DEFAULT_HEIGHT = 300;
14
15   public ObjectInspectorFrame()
16   {
17     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19     // проверить объекты в этом фрейме
20
21     Variable v = new Variable(getClass(), "this", this);
22     ObjectTreeModel model = new ObjectTreeModel();
23     model.setRoot(v);
24
25     // построить и показать дерево
26
27     tree = new JTree(model);
28     add(new JScrollPane(tree), BorderLayout.CENTER);
29   }
30 }
```

Листинг 6.17. Исходный код из файла `treeModel/ObjectTreeModel.java`

```
1  package treeModel;
2
3  import java.lang.reflect.*;
4  import java.util.*;
5  import javax.swing.event.*;
6  import javax.swing.tree.*;
7
8  /**
9   * Эта модель дерева описывает древовидную структуру объектов
10  * в Java. Дочерние узлы дерева являются объектами, хранящимися
11  * в переменных экземпляра
12 */
13 public class ObjectTreeModel implements TreeModel
14 {
15   private Variable root;
16   private EventListenerList listenerList = new EventListenerList();
17
18   /**
19    * Строит пустое дерево
20   */
21   public ObjectTreeModel()
22   {
23     root = null;
24   }
25
26   /**
27    * Устанавливает заданную переменную в корне дерева
28    * @param v Переменная, описываемая в данном дереве
```

```
29    */
30    public void setRoot(Variable v)
31    {
32      Variable oldRoot = v;
33      root = v;
34      fireTreeStructureChanged(oldRoot);
35    }
36
37    public Object getRoot()
38    {
39      return root;
40    }
41
42    public int getChildCount(Object parent)
43    {
44      return ((Variable) parent).getFields().size();
45    }
46
47    public Object getChild(Object parent, int index)
48    {
49      ArrayList<Field> fields = ((Variable) parent).getFields();
50      Field f = (Field) fields.get(index);
51      Object parentValue = ((Variable) parent).getValue();
52      try
53      {
54        return new Variable(f.getType(), f.getName(), f.get(parentValue));
55      }
56      catch (IllegalAccessException e)
57      {
58        return null;
59      }
60    }
61
62    public int getIndexOfChild(Object parent, Object child)
63    {
64      int n = getChildCount(parent);
65      for (int i = 0; i < n; i++)
66        if (getChild(parent, i).equals(child)) return i;
67      return -1;
68    }
69
70    public boolean isLeaf(Object node)
71    {
72      return getChildCount(node) == 0;
73    }
74
75    public void valueForPathChanged(TreePath path, Object newValue)
76    {
77    }
78
79    public void addTreeModelListener(TreeModelListener l)
80    {
81      listenerList.add(TreeModelListener.class, l);
82    }
83
84    public void removeTreeModelListener(TreeModelListener l)
85    {
86      listenerList.remove(TreeModelListener.class, l);
```

```
87    }
88
89    protected void fireTreeStructureChanged(Object oldRoot)
90    {
91       TreeModelEvent event =
92          new TreeModelEvent(this, new Object[] { oldRoot });
93       for (TreeModelListener l :
94             listenerList.getListeners(TreeModelListener.class))
95          l.treeStructureChanged(event);
96    }
97 }
```

Листинг 6.18. Исходный код из файла `treeModel/Variable.java`

```
1  package treeModel;
2
3  import java.lang.reflect.*;
4  import java.util.*;
5
6  /**
7   * Переменная с типом, именем и значением
8  */
9  public class Variable
10 {
11   private Class<?> type;
12   private String name;
13   private Object value;
14   private ArrayList<Field> fields;
15
16   /**
17    * Сконструировать переменную
18    * @param aType Тип
19    * @param aName Имя
20    * @param aValue Значение
21   */
22   public Variable(Class<?> aType, String aName, Object aValue)
23   {
24      type = aType;
25      name = aName;
26      value = aValue;
27      fields = new ArrayList<>();
28
29      // найти все поля, если это тип класса, за исключением символьных
30      // строк и пустых значений null
31
32      if (!type.isPrimitive() && !type.isArray() &&
33            !type.equals(String.class) && value != null)
34      {
35         // получить поля из класса и всех его суперклассов
36         for (Class<?> c = value.getClass(); c != null;
37               c = c.getSuperclass())
38         {
39            Field[] fs = c.getDeclaredFields();
40            AccessibleObject.setAccessible(fs, true);
41
42            // получить все нестатические поля
43            for (Field f : fs)
```

```
44            if ((f.getModifiers() & Modifier.STATIC) == 0) fields.add(f);
45         }
46      }
47   }
48
49   /**
50    * Получает значение данной переменной
51    * @return Возвращает значение
52   */
53   public Object getValue()
54   {
55     return value;
56   }
57
58   /**
59    * Получает все нестатические поля из данной переменной
60    * @return Возвращает списочный массив переменных с описаниями полей
61   */
62   public ArrayList<Field> getFields()
63   {
64     return fields;
65   }
66
67   public String toString()
68   {
69     String r = type + " " + name;
70     if (type.isPrimitive()) r += "=" + value;
71     else if (type.equals(String.class)) r += "=" + value;
72     else if (value == null) r += "=null";
73     return r;
74  }
75 }
```

javax.swing.tree.TreeModel 1.2

- `Object getRoot()`

  Возвращает корневой узел.

- `int getChildCount(Object` *parent*`)`

  Получает количество дочерних узлов заданного родительского узла.

- `Object getChild(Object` *parent*`, int` *index*`)`

  Получает дочерний узел заданного родительского узла по указанному индексу.

- `int getIndexOfChild(Object` *parent*`, Object` *child*`)`

  Получает индекс указанного дочернего узла из заданного родительского узла или значение **-1**, если *child* не является дочерним узлом родительского узла *parent* в данной модели дерева.

- `boolean isLeaf(Object` *node*`)`

  Возвращает логическое значение `true`, если указанный узел является листом дерева.

- `void addTreeModelListener(TreeModelListener` *l*`)`
- `void removeTreeModelListener(TreeModelListener` *l*`)`

  Вводят или удаляют приемник событий, который уведомляется о происходящих в дереве изменениях.

- `void valueForPathChanged(TreePath path, Object newValue)`

  Вызывается в тех случаях, когда значение узла изменяется в редакторе ячеек дерева.

  Параметры:

path	Путь к отредактированному узлу
newValue	Замещающее значение, возвращаемое редактором ячеек дерева

javax.swing.event.TreeModelListener 1.2

- `void treeNodesChanged(TreeModelEvent e)`
- `void treeNodesInserted(TreeModelEvent e)`
- `void treeNodesRemoved(TreeModelEvent e)`
- `void treeStructureChanged(TreeModelEvent e)`

  Вызываются моделью при изменении узлов дерева.

javax.swing.event.TreeModelEvent 1.2

- `TreeModelEvent(Object eventSource, TreePath node)`

  Конструирует событие в модели дерева.

  Параметры:

eventSource	Модель дерева, инициирующая данное событие
node	Путь к измененному узлу

Текстовые компоненты

На рис. 6.35 показаны все текстовые компоненты, входящие в состав библиотеки Swing. Три наиболее употребительных компонента из этой библиотеки были рассмотрены ранее: `JTextField`, `JPasswordField` и `JTextArea` (см. главу 9 первого тома данной книги). А в этом разделе речь пойдет об остальных текстовых компонентах. Вместе с ними будет рассмотрен компонент `JSpinner`, содержащий поле форматированного текста и крошечные кнопки со стрелками вверх и вниз для изменения содержимого этого поля.

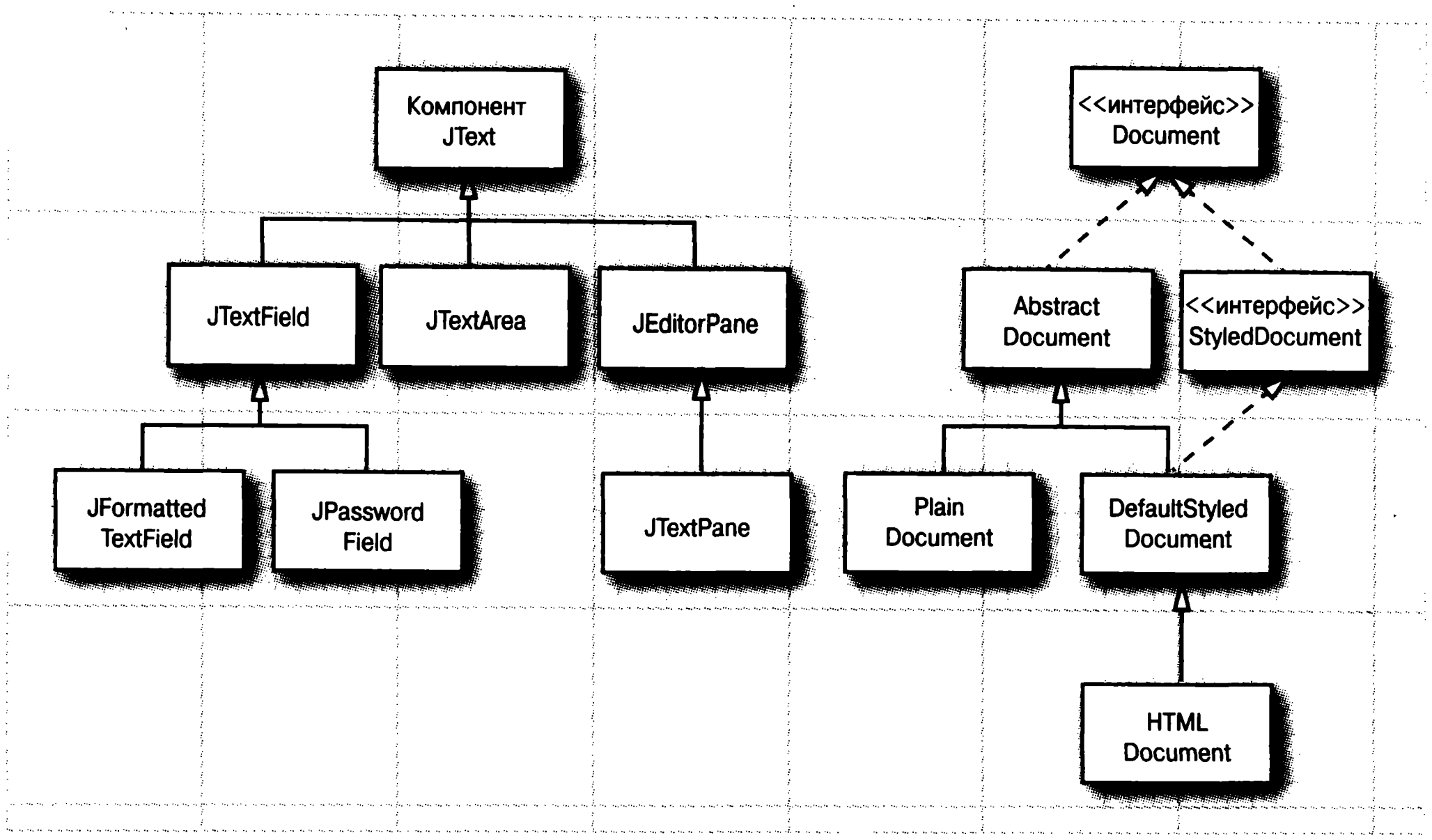


Рис. 6.35. Иерархическая структура текстовых компонентов и документов

Все текстовые компоненты отображают и редактируют данные, хранящиеся в объектной модели класса, реализующего интерфейс `Document`. Компоненты `JTextField` и `JTextArea` используют класс `PlainDocument` для сохранения последовательности строк простого текста без всякого форматирования.

Компонент `JEditorPane` обеспечивает отображение и редактирование стилизованного текста (цветом, шрифтом и проч.) в самых разных форматах, наиболее распространенным из которых является HTML; см. раздел "Отображение HTML-документов средствами `JEditorPane`" далее в этой главе. Интерфейс `StyledDocument` описывает дополнительные требования к стилям, шрифтам и цветам, а класс `HTMLDocument` реализует этот интерфейс.

Компонент `JTextPane`, производный от компонента `JEditorPane`, также позволяет хранить стилизованный текст и встраиваемые компоненты Swing. Но компонент `JTextPane` имеет довольно сложную структуру, и поэтому он здесь не рассматривается. Его подробное описание можно найти в упоминавшейся ранее книге *Swing: Advanced Programming* Кима Топли. А типичный пример применения компонента `JTextPane` представлен в демонстрационной программе `StylePad`, входящей в состав в JDK.

Отслеживание изменений в текстовых компонентах

Сложности в обращении с интерфейсом `Document` возникают только при попытке реализовать свой собственный текстовый редактор. Тем не менее этот интерфейс чаще всего применяется для отслеживания изменений в текстовых компонентах.

Иногда требуется обновить часть ГПИ, не дожидаясь до тех пор, пока пользователь щелкнет на кнопке, завершив редактирование текста. Рассмотрим простой пример программы, в которой отображаются три текстовых поля для красной, синей и зеленой составляющих цвета. Всякий раз, когда изменяется содержимое этих текстовых

полей, возникает потребность в немедленном обновлении цвета. На рис. 6.36 показано рабочее окно данной программы, а ее исходный код представлен в листинге 6.19.

Рис. 6.36. Программа для отслеживания изменений в текстовых полях

Прежде всего следует заметить, что отслеживание нажатий клавиш на клавиатуре — далеко не самая блестящая идея. Дело в том, что некоторые нажатия клавиш (например, клавиш управления курсором) не приводят к изменению текста. Текст может быть обновлен в результате действий мышью (например, средней кнопкой мыши для ввода данных в ОС X11). Вместо этого нужно запросить *документ* (а *не* текстовый компонент), чтобы он уведомлял всякий раз, когда происходит изменение данных. И для этой цели придется установить *приемник событий в документе*, как показано ниже.

```
textField.getDocument().addDocumentListener(listener);
```

После того как текст будет изменен, вызывается один из следующих методов:

```
void insertUpdate(DocumentEvent event)
void removeUpdate(DocumentEvent event)
void changedUpdate(DocumentEvent event)
```

Первые два метода вызываются при вводе или удалении символов. Третий метод вообще не вызывается при редактировании в текстовых полях. Для более сложных видов документов он может быть вызван при ряде других изменений, включая форматирование. К сожалению, уведомить об изменении текста с помощью одиночного обратного вызова нельзя, хотя зачастую и не так важно, каким образом изменяется текст. На этот случай даже не предусмотрен класс адаптера. Таким образом, все три упомянутых выше метода должны быть реализованы в приемнике событий в документе. Ниже показано, как это делается в рассматриваемом здесь примере программы. В методе `setColor()` вызывается метод `getText()` для получения символьных строк, вводимых пользователем в текстовых полях, а также для установки цвета.

```
DocumentListener listener = new DocumentListener()
   {
     public void insertUpdate(DocumentEvent event) { setColor(); }
     public void removeUpdate(DocumentEvent event) { setColor(); }
     public void changedUpdate(DocumentEvent event) {}
   }
```

Рассматриваемая здесь программа имеет следующее ограничение: пользователь может ввести данные в текстовом поле с ошибками или вообще оставить его пустым. Пока что в данной программе перехватывается исключение типа `NumberFormatException`, генерируемое методом `parseInt()`, и просто не обновляется цвет, если в текстовом поле введены данные, которые не являются числовыми. А далее будет показано, как предотвратить ввод пользователем недостоверных данных.

НА ЗАМЕТКУ! Вместо того чтобы принимать события в документе, можно добавить в поле приемник событий действия. Приемнику подобных событий уведомление посылается всякий раз, когда пользователь нажимает клавишу <Enter>. Но такой прием все же не рекомендуется, поскольку пользователи нередко забывают, что нужно нажать клавишу <Enter> по завершении ввода данных. Если вы все-таки решите воспользоваться приемником событий действия, вам придется также установить приемник фокуса ввода, чтобы не пропустить момент, когда пользователь покинет текстовое поле.

Листинг 6.19. Исходный код из файла `textChange/ColorFrame.java`

```
1  package textChange;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.event.*;
6
7  /**
8   * Фрейм с тремя текстовыми полями для установки цвета фона
9  */
10 public class ColorFrame extends JFrame
11 {
12   private JPanel panel;
13   private JTextField redField;
14   private JTextField greenField;
15   private JTextField blueField;
16
17   public ColorFrame()
18   {
19     DocumentListener listener = new DocumentListener()
20       {
21         public void insertUpdate(DocumentEvent event) { setColor(); }
22         public void removeUpdate(DocumentEvent event) { setColor(); }
23         public void changedUpdate(DocumentEvent event) {}
24       };
25
26     panel = new JPanel();
27
28     panel.add(new JLabel("Red:"));
29     redField = new JTextField("255", 3);
30     panel.add(redField);
31     redField.getDocument().addDocumentListener(listener);
32
33     panel.add(new JLabel("Green:"));
34     greenField = new JTextField("255", 3);
35     panel.add(greenField);
36     greenField.getDocument().addDocumentListener(listener);
37     panel.add(new JLabel("Blue:"));
38     blueField = new JTextField("255", 3);
39     panel.add(blueField);
40     blueField.getDocument().addDocumentListener(listener);
41
42     add(panel);
43     pack();
44   }
45
```

```
46   /**
47    * Установить цвет фона в соответствии со значениями,
48    * хранящимися в текстовых полях
49   */
50   public void setColor()
51   {
52     try
53     {
54       int red = Integer.parseInt(redField.getText().trim());
55       int green = Integer.parseInt(greenField.getText().trim());
56       int blue = Integer.parseInt(blueField.getText().trim());
57       panel.setBackground(new Color(red, green, blue));
58     }
59     catch (NumberFormatException e)
60     {
61       // не устанавливать цвет, если введенные данные
62       // нельзя проанализировать синтаксически
63     }
64   }
65 }
```

javax.swing.JComponent 1.2

- `Dimension getPreferredSize()`
- `void setPreferredSize(Dimension d)`

  Получают или устанавливают предпочтительные размеры данного компонента.

javax.swing.text.Document 1.2

- `int getLength()`

  Возвращает количество символов, которые документ содержит в данный момент.

- `String getText(int offset, int length)`

  Возвращает текст, содержащийся в данной части документа.

Параметры:	*offset*	Начало текста
	length	Требуемая длина символьной строки

- `void addDocumentListener(DocumentListener listener)`

  Регистрирует приемник событий для получения уведомлений об изменениях в документе.

javax.swing.event.DocumentEvent 1.2

- `Document getDocument()`

  Получает документ, являющийся источником событий.

javax.swing.event.DocumentListener 1.2

- `void changedUpdate(DocumentEvent event)`
  Вызывается всякий раз, когда изменяется атрибут или ряд атрибутов.
- `void insertUpdate(DocumentEvent event)`
  Вызывается всякий раз, когда текст вводится в документ.
- `void removeUpdate(DocumentEvent event)`
  Вызывается всякий раз, когда удаляется часть документа.

Поля ввода форматируемого текста

В предыдущем примере программы требовалось, чтобы пользователь вводил числа, а не произвольные символьные строки, т.е. пользователю разрешалось вводить только цифры от **0** до **9** и дефис (**-**). Но если дефис вводится, он должен быть *первым* символом во вводимой строке.

На первый взгляд, проверка такого ввода не является сложной задачей. Так, в текстовом поле можно установить приемник событий от клавиш, а затем удалять все события от клавиш, которые не связаны с вводом цифр или дефиса. К сожалению, этот простой способ, хоть он обычно и рекомендуется для проверки вводимых данных, на практике оказывается ненадежным. Во-первых, не всякая комбинация допустимых для ввода символов является достоверным числом. Например, комбинации символов **--3** и **3-3** не являются достоверными комбинациями, несмотря на то, что они состоят из допустимых символов. Но самое главное, что имеются и другие способы изменения текста, не подразумевающие ввод символов с клавиатуры. В зависимости от предпочтений пользователя для вырезания, копирования и вставки текста могут использоваться определенные комбинации клавиш. Например, в визуальном стиле Metal с помощью комбинации клавиш <Ctrl+V> производится вставка содержимого из буфера обмена в текстовое поле. Это означает, что нужно следить и за тем, чтобы пользователь не вставил недействительный символ. Ясно, что попытка отфильтровывать нажатия клавиш с целью гарантировать достоверность содержимого текстового поля может превратиться в настоящий кошмар. Поэтому разработчики прикладных программ должны быть избавлены от подобных хлопот.

Как ни странно, но до выхода версии Java SE 1.4 не существовало компонентов, предназначенных для ввода числовых значений. Начиная с первого издания данной книги мы предоставляли реализацию текстового поля типа `IntTextField` для ввода целого числа с подходящим форматированием. В каждом последующем издании мы изменяли эту реализацию, чтобы воспользоваться любым преимуществом от еще незрелых схем проверки, внедрявшихся в каждую версию Java. И наконец, в версии Java SE 1.4 разработчики библиотеки Swing внедрили класс `JFormattedTextField`, который можно использовать не только для ввода чисел, но и для ввода дат и даже более экзотических форматируемых значений вроде IP-адресов.

Ввод целочисленных значений

Начнем с простого случая, когда текстовое поле используется для ввода целочисленных значений следующим образом:

```
JFormattedTextField intField =
    new JFormattedTextField(NumberFormat.getIntegerInstance());
```

Метод `NumberFormat.getIntegerInstance()` возвращает объект, который форматирует целые числа с учетом текущих региональных настроек. В США, например, запятые используются в качестве десятичных разделителей, позволяя вводить значения вроде **1,729**. В главе 5 уже пояснялось, каким образом можно сменить текущие региональные настройки.

Как и для любого другого текстового поля, в данном случае можно установить несколько столбцов. Ниже показано, как это делается.

```
intField.setColumns(6);
```

Значение по умолчанию можно задать с помощью метода `setValue()`. Этот метод принимает параметр типа `Object`, поэтому значение типа `int` необходимо представить как объект типа `Integer` следующим образом:

```
intField.setValue(new Integer(100));
```

Как правило, пользователи вводят данные в нескольких текстовых полях, а затем щелкают на кнопке, чтобы все введенные значения были прочитаны. После активизации кнопки щелчком на ней введенное пользователем значение можно получить, вызвав метод `getValue()`. Этот метод возвращает результат типа `Object`, поэтому его придется привести к соответствующему типу. Из текстового поля типа `JFormattedTextField` возвращается объект типа `Long`, если пользователь отредактировал значение. А если пользователь не вносил никаких изменений, то возвращается исходный объект типа `Integer`. Таким образом, возвращаемое значение необходимо привести к обычному суперклассу `Number`, как показано ниже.

```
Number value = (Number) intField.getValue();
int v = value.intValue();
```

Поле ввода форматируемого текста не представляет особого интереса до тех пор, пока не возникает потребность учесть то, что может произойти, если пользователь введет недостоверные данные. Этот вопрос обсуждается в следующем подразделе.

Поведение при потере фокуса ввода

Попробуем выяснить, что же происходит, когда пользователь вводит данные в текстовом поле. Если пользователь введет какие-нибудь данные и в какой-то момент решит покинуть текстовое поле, возможно, щелкнув кнопкой мыши на другом компоненте, то текстовое поле *потеряет фокус ввода*. Мигающий курсор больше не появляется в текстовом поле, а нажатия клавиш на клавиатуре направляются другому компоненту.

Когда поле ввода форматируемого текста теряет фокус ввода, средство форматирования анализирует текстовую строку, введенную пользователем. Если этому средству известно, как преобразовать текстовую строку в объект, то введенный текст считается достоверным, а иначе — недостоверным. Для того чтобы проверить, является ли текущее содержимое текстового поля достоверным, можно вызвать метод `isEditValid()`.

Поведение по умолчанию при потере фокуса называется "принять или возвратить". Если текстовая строка является достоверной, она *принимается*. Средство форматирования преобразует ее в объект, который становится текущим значением поля (т.е. значением, возвращаемым из метода `getValue()`, как было показано в предыдущем подразделе). Затем это значение преобразуется обратно в символьную

строку, которая становится текстовой строкой, отображаемой в поле ввода. Например, средство форматирования целочисленных значений распознает введенную комбинацию **1729** как достоверную, устанавливает текущее значение в виде объекта `new Long(1729)`, а затем преобразует его обратно в символьную строку с запятой в качестве десятичного разделителя: **1,729**.

А если текстовая строка оказывается недостоверной, то текущее значение не меняется и текстовое поле *возвращается* к символьной строке, представляющей прежнее значение. Так, если пользователь введет неверное значение вроде **x1**, то при потере фокуса ввода будет восстановлено прежнее значение.

НА ЗАМЕТКУ! Средство форматирования целых чисел считает текстовую строку достоверной, если она начинается с целого числа. Например, строка **1729x** является действительной. Она преобразуется в число **1729**, которое затем преобразуется в символьную строку **1,729**.

С помощью метода `setFocusLostBehavior()` можно установить другие виды поведения. Поведение "принять" немногим отличается от поведения по умолчанию. Если текстовая строка оказывается недостоверной, то сама текстовая строка и соответствующее ей значение в текстовом поле остаются без изменения, но с этого момента они перестают быть синхронизированными. Поведение "продолжить" является еще более консервативным. Даже если текстовая строка оказывается недостоверной, ни текстовое поле, ни текущее значение не изменяется. Для того чтобы синхронизировать их снова, можно даже вызвать метод `commitEdit()` или `setText()`. И наконец, имеется еще и поведение "возвратить", хотя пользы от него немного. Всякий раз, когда теряется фокус ввода, данные, введенные пользователем, игнорируются, а текстовая строка возвращается к старому значению.

НА ЗАМЕТКУ! Как правило, по умолчанию используется поведение "принять или возвратить", но оно чревато серьезными осложнениями. Допустим, диалоговое окно содержит текстовое поле для ввода целочисленных значений. Пользователь вводит текстовую строку `" 1729"` с пробелом в качестве первого символа и щелкает на кнопке OK. Этот пробел делает введенное числовое значение недостоверным, и поэтому текстовое поле возвращается к старому значению. Приемник событий от кнопки OK получает значение из текстового поля и просто закрывает диалоговое окно, а пользователь так и не узнает, что новое значение было отклонено. В подобных случаях следует выбрать поведение "принять" и принудить приемник событий от кнопки OK проверить все правки в текстовом поле на достоверность, прежде чем закрывать диалоговое окно.

Фильтры

Основные функции полей ввода форматируемого текста просты и достаточны для большинства случаев их применения. Тем не менее их можно дополнить некоторыми усовершенствованиями. Допустим, требуется сделать так, чтобы пользователь вообще не смог ввести нецифровые значения. Этого можно добиться с помощью *фильтра документа*. Напомним, что контроллер в шаблоне проектирования "модель–представление–контроллер" преобразует события ввода в команды и изменяет базовый документ в текстовом поле, т.е. текстовую строку, хранящуюся в объекте типа `PlainDocument`. Например, всякий раз, когда контроллер обрабатывает команду вставки текста в документ, он вызывает команду "вставить строку". Строка, которую требуется вставить, может быть представлена одним символом или содержимым буфера вставки. Фильтр документа может перехватить эту команду и изменить

символьную строку или вообще отменить ее вставку. Ниже приведен исходный код метода `insertString()` для фильтра, анализирующего вставляемую символьную строку и вставляющего только те символы, которые являются цифрами или знаком -. (В этом методе обрабатываются дополнительные символы в уникоде, как пояснялось в главе 3 первого тома данной книги. А описание класса `StringBuilder` приведено в главе 1 настоящего тома.)

```
public void insertString(FilterBypass fb, int offset, String string,
      AttributeSet attr) throws BadLocationException
{
  StringBuilder builder = new StringBuilder(string);
  for (int i = builder.length() - 1; i >= 0; i--)
  {
    int cp = builder.codePointAt(i);
    if (!Character.isDigit(cp) && cp != '-')
    {
      builder.deleteCharAt(i);
      if (Character.isSupplementaryCodePoint(cp))
      {
        i--;
        builder.deleteCharAt(i);
      }
    }
  }
  super.insertString(fb, offset, builder.toString(), attr);
}
```

Необходимо также переопределить метод `replace()` из класса `DocumentFilter`. Он вызывается при выборе текста и последующей его вставке. Реализация метода `replace()` не представляет особых трудностей, как будет показано в листинге 6.21.

Затем следует установить фильтр документа. К сожалению, простого способа сделать это не существует. Для этого придется сначала переопределить метод `getDocumentFilter()` из класса, реализующего средство форматирования, а затем передать форматирующий объект этого класса в текстовое поле типа `JFormattedTextField`. Так, в текстовом поле ввода целочисленных значений используется средство форматирования типа `InternationalFormatter`, которое инициализируется с помощью метода `NumberFormat.getIntegerInstance()`. В приведенном ниже фрагменте кода показано, каким образом осуществляется установка средства форматирования для получения требуемого фильтра.

```
JFormattedTextField intField = new JFormattedTextField(
      new InternationalFormatter(NumberFormat.getIntegerInstance())
{
  private DocumentFilter filter = new IntFilter();
  protected DocumentFilter getDocumentFilter()
  {
    return filter;
  }
});
```

Попробуйте выполнить пример программы `FormatTest`, исходный код которой представлен в листинге 6.20. В третьем текстовом поле этой программы установлен фильтр. В этом поле допускается вставлять только цифры или знак "минус" (-), но по-прежнему можно вводить недостоверные символьные строки наподобие `"1-2-3"`. Вообще говоря, фильтрация не позволяет застраховаться от ввода всех недостоверных символьных строк. Например, строка `"-"` является недостоверной, хотя фильтр

не может отклонить ее, поскольку это приставка к достоверной строке "-1". И хотя фильтры не в состоянии обеспечить идеальную защиту от ввода недостоверных данных, пользоваться ими все же следует для отклонения тех вводимых данных, которые явно недействительны.

НА ЗАМЕТКУ! В документации на Java SE сказано, что класс `DocumentFilter` разработан для того, чтобы избавиться от лишней подклассификации. До версии Java SE 1.3 фильтрация в текстовом поле достигалась расширением класса `PlainDocument` и переопределением методов `insertString()` и `replace()`. Теперь же в классе `PlainDocument` имеется подключаемый фильтр. И это, конечно, замечательное усовершенствование, но оно могло бы быть еще лучше, если бы фильтр был сделан подключаемым и в классе, реализующем средство форматирования. Увы, этого не было сделано, и поэтому приходится выполнять подклассификацию класса, реализующего средство форматирования.

СОВЕТ. Фильтрация применяется и для того, чтобы перевести все символы в верхний регистр. Написать код для такого фильтра нетрудно. В методах `insertString()` и `replace()` из класса этого фильтра необходимо перевести символы вставляемой строки в верхний регистр, а затем вызвать соответствующий метод из суперкласса.

Средства проверки достоверности

Существует еще один потенциально полезный механизм уведомления пользователей о неверно введенных данных. К любому компоненту типа `JComponent` можно присоединить *средство проверки достоверности*. Если компонент теряет фокус ввода, происходит обращение к средству проверки достоверности. И если это средство уведомляет о недостоверности содержимого компонента, то фокус ввода сразу же возвращается к данному компоненту. Таким образом, пользователю придется устранить ошибку во введенных данных, прежде чем вводить другие данные.

Средство проверки достоверности должно расширять абстрактный класс `InputVerifier` и определять метод `verify()`. Определить средство проверки достоверности для полей ввода форматируемого текста совсем не трудно. Метод `isEditValid()` из класса `JFormattedTextField` вызывает средство форматирования и возвращает логическое значение `true`, если это средство преобразует текстовую строку в объект. В приведенном ниже фрагменте кода показано, каким образом средство проверки достоверности присоединяется к текстовому полю типа `JFormattedTextField`.

```
intField.setInputVerifier(new InputVerifier()
  {
    public boolean verify(JComponent component)
    {
      JFormattedTextField field = (JFormattedTextField) component;
      return field.isEditValid();
    }
  })
```

Четвертое текстовое поле в рассматриваемом здесь примере программы имеет средство проверки достоверности. Попробуйте ввести недействительное число (например, **x1729**) и нажмите клавишу <Tab> или щелкните кнопкой мыши на другом текстовом поле. Обратите внимание на то, что поле тут же получит фокус ввода. Но

если вы щелкнете на кнопке OK, приемник действий вызовет метод `getValue()`, который сообщит о последнем действительном значении.

Но и средство проверки достоверности нельзя считать совершенно надежным. Если щелкнуть на кнопке, она уведомит приемники действий до того, как компонент с недействительными данными снова получит фокус ввода. И тогда приемники действий могут получить недействительный результат из компонента, который не прошел проверку на достоверность. Подобное поведение объясняется тем, что пользователь может при желании щелкнуть на кнопке Cancel (Отмена), так и не устранив ошибку во введенных данных.

Другие стандартные средства форматирования

Помимо средств форматирования целочисленных значений, в классе `JFormattedTextField` поддерживается также ряд других средств форматирования числовых значений с плавающей точкой, в процентах и денежных единицах. Для этой цели в классе `NumberFormat` имеются следующие статические методы:

```
getNumberInstance()
getCurrencyInstance()
getPercentInstance()
```

Например, чтобы получить текстовое поле для ввода денежных сумм, нужно вызвать следующий метод:

```
JFormattedTextField currencyField =
    new JFormattedTextField(NumberFormat.getCurrencyInstance());
```

Для редактирования даты и времени достаточно вызвать один из следующих статических методов из класса `DateFormat`:

```
getDateInstance()
getTimeInstance()
getDateTimeInstance()
```

Например, в приведенном ниже поле дата редактируется в принятом по умолчанию так называемом "среднем" формате **Aug 5, 2007**.

```
JFormattedTextField dateField =
    new JFormattedTextField(DateFormat.getDateInstance());
```

Вместо него можно выбрать другой, так называемый "короткий" формат **8/5/07**, вызвав следующий метод:

```
DateFormat.getDateInstance(DateFormat.SHORT)
```

НА ЗАМЕТКУ! По умолчанию для дат выбирается так называемый "снисходительный" формат. Иными словами, несуществующая дата вроде **February 31, 2002** будет приведена к дате **March 3, 2002**. Подобное поведение может показаться необычным для пользователей. В таком случае следует вызвать метод `setLenient(false)` для объекта типа `DateFormat`.

Средство форматирования по умолчанию типа `DefaultFormatter` позволяет форматировать объекты любого класса, имеющего конструктор со строковым параметром и соответствующий метод `toString()`. Например, в классе `URL` имеется конструктор `URL(String)`, который можно использовать для составления URL из символьной строки следующим образом:

```
URL url = new URL("http://horstmann.com");
```

Таким образом, средство типа `DefaultFormatter` можно использовать для форматирования объектов типа `URL`. Это средство вызывает метод `toString()` со значением поля, чтобы инициализировать текст в данном поле. Когда поле теряет фокус ввода, средство форматирования создает новый объект того же класса, что и класс текущего значения, используя конструктор с параметром типа `String`. Если этот конструктор генерирует исключение, то результат редактирования в данном поле считается недостоверным. Вы можете сами убедиться в этом, введя URL без префикса **`"http:"`** в рассматриваемом здесь примере программы.

НА ЗАМЕТКУ! По умолчанию средство форматирования типа `DefaultFormatter` работает в *режиме перезаписи*. Этим оно отличается от остальных средств форматирования. Но пользы от такого режима немного, поэтому его лучше отключить, вызвав метод `setOverwriteMode(false)`.

Наконец, средство форматирования типа `MaskFormatter` оказывается удобным для форматирования по шаблонам фиксированного размера, состоящим из констант и переменных символов. Например, номера карточек социального страхования в США (вроде **`078-05-1120`**) могут быть отформатированы по приведенному ниже шаблону, где знак **`#`** обозначает одну цифру. В табл. 6.3 перечислены символы, которые можно использовать в средстве форматирования по шаблону.

```
new MaskFormatter("###-##-####")
```

Таблица 6.3. Символы, употребляемые в средстве форматирования типа `MaskFormatter`

Символ	Описание
`#`	Цифра
`?`	Буква
`U`	Буква, преобразуемая в прописную
`L`	Буква, преобразуемая в строчную
`A`	Буква или цифра
`H`	Шестнадцатеричная цифра **`[0-9A-Fa-f]`**
`*`	Любой символ
`'`	Символ перехода, который позволяет включить специальный символ в шаблон, экранируя его

На символы, которые допускается вводить в поле, можно наложить определенные ограничения, вызвав один из следующих методов из класса `MaskFormatter`:

```
setValidCharacters()
setInvalidCharacters()
```

Для того чтобы, например, прочитать буквенную оценку академической успеваемости (в частности, A+ или F), можно было вы воспользоваться приведенным ниже фрагментом кода. Но указать, что второй символ не может быть буквой, никак нельзя.

```
MaskFormatter formatter = new MaskFormatter("U*");
formatter.setValidCharacters("ABCDF+- ");
```

Следует иметь в виду, что символьная строка, форматируемая средством форматирования по шаблону, имеет такую же длину, как и у шаблона. Если пользователь удаляет символы во время редактирования, они будут заменяться *символом-заполнителем*.

По умолчанию в качестве такого символа используется пробел, но его можно заменить, вызвав метод `setPlaceholderCharacter()`, как показано ниже.

```
formatter.setPlaceholderCharacter('0');
```

По умолчанию средство форматирования по шаблону работает в режиме замены вводимых данных, понять который можно интуитивно, если опробовать его в рассматриваемом здесь примере программы. Следует также иметь в виду, что позиция вставки переходит через фиксированные символы в шаблоне.

Средства форматирования по шаблону очень эффективны при использовании строгих шаблонов вроде номеров карточек социального страхования и номеров телефонов в США. Но, поскольку в шаблонах не допускается никаких отклонений, средство форматирования по шаблону не годится, например, для международных номеров телефонов, имеющих разное количество цифр.

Специальные средства форматирования

Если ни одно из стандартных средств форматирования не подходит, можно без труда определить собственное. Рассмотрим, например, 4-байтовые IP-адреса наподобие следующего:

```
130.65.86.66
```

Для форматирования подобных адресов по шаблону нельзя воспользоваться классом `MaskFormatter`, поскольку каждый байт в IP-адресе может быть представлен одной, двумя или тремя цифрами. Кроме того, в средстве форматирования необходимо проверить, не превышает ли значение в каждом байте величину **255**. Чтобы определить собственное средство форматирования, нужно расширить класс `DefaultFormatter` и переопределить следующие методы:

```
String valueToString(Object value)
Object stringToValue(String text)
```

Первый метод преобразует значение из текстового поля в символьную строку, которая затем отображается в этом поле. А второй метод анализирует введенный пользователем текст и преобразует его обратно в объект. Если в одном из этих методов обнаружится ошибка, должно быть сгенерировано исключение типа `ParseException`.

В рассматриваемом здесь примере программы IP-адреса сохраняются в массиве `byte[]` длиной, равной **4**. Метод `valueToString()` формирует символьную строку, разделяя в ней байты точками. Следует иметь в виду, что значения типа `byte` могут находиться в пределах от **-128** до **127**. (Например, в IP-адресе **130.65.86.66** первый октет на самом деле является байтом со значением **-126**.) Чтобы преобразовать отрицательные значения байтов в целочисленные значения без знака, к ним следует прибавить значение **256**, как показано ниже.

```
public String valueToString(Object value) throws ParseException
{
  if (!(value instanceof byte[]))
    throw new ParseException("Not a byte[]", 0);
  byte[] a = (byte[]) value;
  if (a.length != 4)
    throw new ParseException("Length != 4", 0);
  StringBuilder builder = new StringBuilder();
  for (int i = 0; i < 4; i++)
  {
    int b = a[i];
```

```
    if (b < 0) b += 256;
    builder.append(String.valueOf(b));
    if (i < 3) builder.append('.');
  }
  return builder.toString();
}
```

Напротив, метод `stringToValue()` осуществляет синтаксический анализ символьной строки и выдает объект типа `byte[]`, если эта строка оказывается достоверной. В противном случае генерируется исключение типа `ParseException`, как показано в приведенном ниже фрагменте кода.

```
public Object stringToValue(String text) throws ParseException
{
  StringTokenizer tokenizer = new StringTokenizer(text, ".");
  byte[] a = new byte[4];
  for (int i = 0; i < 4; i++)
  {
    int b = 0;
    try
    {
      b = Integer.parseInt(tokenizer.nextToken());
    }
    catch (NumberFormatException e)
    {
      throw new ParseException("Not an integer", 0);
    }
    if (b < 0 || b >= 256)
      throw new ParseException("Byte out of range", 0);
    a[i] = (byte) b;
  }
  return a;
}
```

Попробуйте ввести данные в поле IP-адреса из рассматриваемого здесь примера программы. Если вы введете недостоверный адрес, в этом поле отобразится последний достоверный адрес. Весь исходный код специального средства форматирования представлен в листинге 6.22, а в примере программы из листинга 6.20 демонстрируется применение различных полей ввода форматируемого текста (рис. 6.37). Для того чтобы извлечь текущие значения из этих полей, достаточно щелкнуть на кнопке OK.

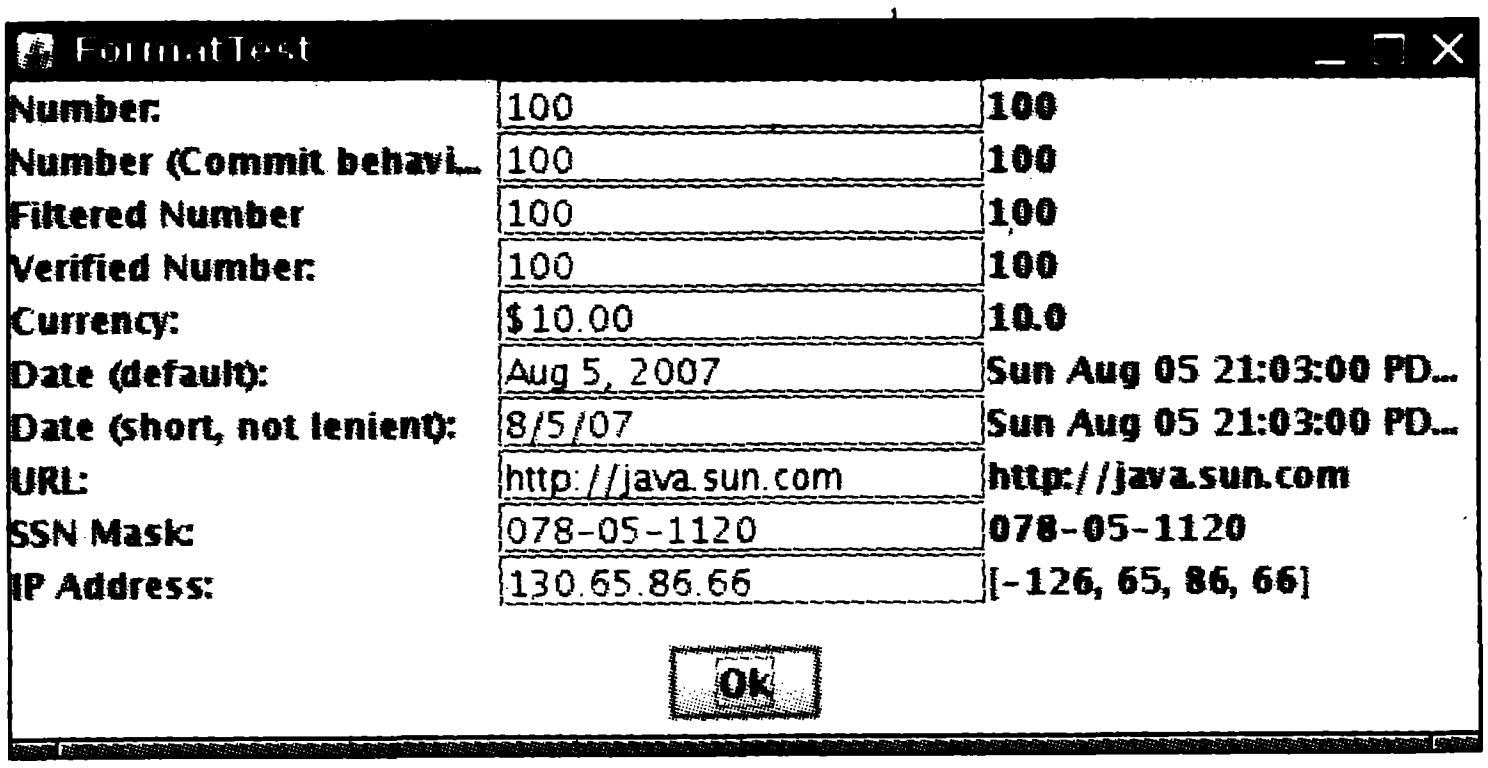


Рис. 6.37. Программа `FormatTest`

НА ЗАМЕТКУ! По адресу `www.oracle.com/technetwork/java/reftf-138955.html` доступна краткая статья, в которой описывается средство форматирования, для которого подходит любое регулярное выражение.

Листинг 6.20. Исходный код из файла `textFormat/FormatTestFrame.java`

```
1   package textFormat;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.net.*;
6   import java.text.*;
7   import java.util.*;
8   import javax.swing.*;
9   import javax.swing.text.*;
10  /**
11   * Фрейм с целым рядом полей ввода форматируемого текста и
12   * кнопкой для отображения значений в этих полях
13  */
14  public class FormatTestFrame extends JFrame
15  {
16    private DocumentFilter filter = new IntFilter();
17    private JButton okButton;
18    private JPanel mainPanel;
19
20    public FormatTestFrame()
21    {
22      JPanel buttonPanel = new JPanel();
23      okButton = new JButton("Ok");
24      buttonPanel.add(okButton);
25      add(buttonPanel, BorderLayout.SOUTH);
26
27      mainPanel = new JPanel();
28      mainPanel.setLayout(new GridLayout(0, 3));
29      add(mainPanel, BorderLayout.CENTER);
30
31      JFormattedTextField intField =
32         new JFormattedTextField(NumberFormat.getIntegerInstance());
33      intField.setValue(new Integer(100));
34      addRow("Number:", intField);
35
36      JFormattedTextField intField2 =
37         new JFormattedTextField(NumberFormat.getIntegerInstance());
38      intField2.setValue(new Integer(100));
39      intField2.setFocusLostBehavior(JFormattedTextField.COMMIT);
40      addRow("Number (Commit behavior):", intField2);
41
42      JFormattedTextField intField3 =
43           new JFormattedTextField(new InternationalFormatter(
44              NumberFormat.getIntegerInstance())
45        {
46          protected DocumentFilter getDocumentFilter()
47          {
48            return filter;
49          }
50        });
```

```
51        intField3.setValue(new Integer(100));
52        addRow("Filtered Number", intField3);
53
54        JFormattedTextField intField4 =
55           new JFormattedTextField(NumberFormat.getIntegerInstance());
56        intField4.setValue(new Integer(100));
57        intField4.setInputVerifier(new InputVerifier()
58          {
59            public boolean verify(JComponent component)
60            {
61              JFormattedTextField field = (JFormattedTextField) component;
62              return field.isEditValid();
63            }
64          });
65        addRow("Verified Number:", intField4);
66
67        JFormattedTextField currencyField =
68           new JFormattedTextField(
69              NumberFormat.getCurrencyInstance());
70        currencyField.setValue(new Double(10));
71        addRow("Currency:", currencyField);
72
73        JFormattedTextField dateField =
74           new JFormattedTextField(DateFormat.getDateInstance());
75        dateField.setValue(new Date());
76        addRow("Date (default):", dateField);
77
78        DateFormat format = DateFormat.getDateInstance(DateFormat.SHORT);
79        format.setLenient(false);
80        JFormattedTextField dateField2 = new JFormattedTextField(format);
81        dateField2.setValue(new Date());
82        addRow("Date (short, not lenient):", dateField2);
83
84        try
85        {
86          DefaultFormatter formatter = new DefaultFormatter();
87          formatter.setOverwriteMode(false);
88          JFormattedTextField urlField =
89             new JFormattedTextField(formatter);
90          urlField.setValue(new URL("http://java.sun.com"));
91          addRow("URL:", urlField);
92        }
93        catch (MalformedURLException ex)
94        {
95          ex.printStackTrace();
96        }
97
98        try
99        {
100         MaskFormatter formatter = new MaskFormatter("###-##-####");
101         formatter.setPlaceholderCharacter('0');
102         JFormattedTextField ssnField =
103            new JFormattedTextField(formatter);
104         ssnField.setValue("078-05-1120");
105         addRow("SSN Mask:", ssnField);
106       }
107       catch (ParseException ex)
108       {
```

```
109         ex.printStackTrace();
110       }
111
112       JFormattedTextField ipField =
113          new JFormattedTextField(new IPAddressFormatter());
114       ipField.setValue(new byte[] { (byte) 130, 65, 86, 66 });
115       addRow("IP Address:", ipField);
116       pack();
117    }
118
119    /**
120     * вводит ряд полей на основной панели
121     * @param labelText Метка поля
122     * @param field Образцовое поле
123    */
124    public void addRow(String labelText, final JFormattedTextField field)
125    {
126       mainPanel.add(new JLabel(labelText));
127       mainPanel.add(field);
128       final JLabel valueLabel = new JLabel();
129       mainPanel.add(valueLabel);
130       okButton.addActionListener(new ActionListener()
131          {
132             public void actionPerformed(ActionEvent event)
133             {
134                Object value = field.getValue();
135                Class<?> cl = value.getClass();
136                String text = null;
137                if (cl.isArray())
138                {
139                   if (cl.getComponentType().isPrimitive())
140                   {
141                      try
142                      {
143                         text = Arrays.class.getMethod(
144                            "toString", cl).invoke(null, value).toString();
145                      }
146                      catch (ReflectiveOperationException ex)
147                      {
148                         // игнорировать ожидаемые результаты рефлексии
149                      }
150                   }
151                   else text = Arrays.toString((Object[]) value);
152                }
153                else text = value.toString();
154                valueLabel.setText(text);
155             }
156          });
157    }
158 }
```

Листинг 6.21. Исходный код из файла `textFormat/IntFilter.java`

```
1  package textFormat;
2
3  import javax.swing.text.*;
4
```

```
5  /**
6   * Фильтр, ограничивающий ввод данных цифрами и знаком '-'
7  */
8  public class IntFilter extends DocumentFilter
9  {
10   public void insertString(FilterBypass fb, int offset,
11      String string, AttributeSet attr) throws BadLocationException
12   {
13     StringBuilder builder = new StringBuilder(string);
14     for (int i = builder.length() - 1; i >= 0; i--)
15     {
16       int cp = builder.codePointAt(i);
17       if (!Character.isDigit(cp) && cp != '-')
18       {
19         builder.deleteCharAt(i);
20         if (Character.isSupplementaryCodePoint(cp))
21         {
22           i--;
23           builder.deleteCharAt(i);
24         }
25       }
26     }
27     super.insertString(fb, offset, builder.toString(), attr);
28   }
29
30   public void replace(FilterBypass fb, int offset, int length,
31      String string, AttributeSet attr) throws BadLocationException
32   {
33     if (string != null)
34     {
35       StringBuilder builder = new StringBuilder(string);
36       for (int i = builder.length() - 1; i >= 0; i--)
37       {
38         int cp = builder.codePointAt(i);
39         if (!Character.isDigit(cp) && cp != '-')
40         {
41           builder.deleteCharAt(i);
42           if (Character.isSupplementaryCodePoint(cp))
43           {
44             i--;
45             builder.deleteCharAt(i);
46           }
47         }
48       }
49       string = builder.toString();
50     }
51     super.replace(fb, offset, length, string, attr);
52   }
53 }
```

Листинг 6.22. Исходный код из файла `textFormat/IPAddressFormatter.java`

```
1  package textFormat;
2
3  import java.text.*;
4  import java.util.*;
5  import javax.swing.text.*;
```

```
6
7  /**
8   * Средство форматирования 4-байтовых IP-адресов в форме a.b.c.d
9  */
10 public class IPAddressFormatter extends DefaultFormatter
11 {
12   public String valueToString(Object value) throws ParseException
13   {
14     if (!(value instanceof byte[]))
15        throw new ParseException("Not a byte[]", 0);
16     byte[] a = (byte[]) value;
17     if (a.length != 4) throw new ParseException("Length != 4", 0);
18     StringBuilder builder = new StringBuilder();
19     for (int i = 0; i < 4; i++)
20     {
21       int b = a[i];
22       if (b < 0) b += 256;
23       builder.append(String.valueOf(b));
24       if (i < 3) builder.append('.');
25     }
26     return builder.toString();
27   }
28
29   public Object stringToValue(String text) throws ParseException
30   {
31     StringTokenizer tokenizer = new StringTokenizer(text, ".");
32     byte[] a = new byte[4];
33     for (int i = 0; i < 4; i++)
34     {
35       int b = 0;
36       if (!tokenizer.hasMoreTokens())
37          throw new ParseException("Too few bytes", 0);
38       try
39       {
40         b = Integer.parseInt(tokenizer.nextToken());
41       }
42       catch (NumberFormatException e)
43       {
44         throw new ParseException("Not an integer", 0);
45       }
46       if (b < 0 || b >= 256) throw new ParseException(
47          "Byte out of range", 0);
48       a[i] = (byte) b;
49     }
50     if (tokenizer.hasMoreTokens()) throw new ParseException(
51         "Too many bytes", 0);
52     return a;
53   }
54 }
```

javax.swing.JFormattedTextField 1.4

- `JFormattedTextField(Format fmt)`

  Создает текстовое поле, в котором используется указанный формат.

- `JFormattedTextField(JFormattedTextField.AbstractFormatter` *`formatter`*`)`
  Создает текстовое поле, в котором используется указанное средство форматирования. Следует иметь в виду, что классы `DefaultFormatter` и `InternationalFormatter` являются производными от класса `JFormattedTextField.AbstractFormatter`.
- `Object getValue()`
  Возвращает текущее достоверное значение из поля. Следует иметь в виду, что оно может не соответствовать символьной строке, редактируемой в данном поле.
- `void setValue(Object` *`value`*`)`
  Пытается установить значение для указанного объекта. Попытка окажется неудачной, если средству форматирования не удастся преобразовать объект в символьную строку.
- `void commitEdit()`
  Пытается установить достоверное значение символьной строки, редактируемой в поле. Попытка может оказаться неудачной, если средству форматирования не удастся преобразовать символьную строку.
- `boolean isEditValid()`
  Проверяет, представляет ли редактируемая символьная строка достоверное значение.
- `int getFocusLostBehavior()`
- `void setFocusLostBehavior(int` *`behavior`*`)`
  Получают или устанавливают поведение "потеря фокуса ввода". Параметр *`behavior`* может принимать значения следующих констант из класса `JFormattedTextField`: `COMMIT_OR_REVERT`, `REVERT`, `COMMIT` и `PERSIST`.

javax.swing.JFormattedTextField.AbstractFormatter 1.4

- `abstract String valueToString(Object` *`value`*`)`
  Преобразует значение в редактируемую символьную строку. Генерирует исключение типа `ParseException`, если значение не подходит для данного средства форматирования.
- `abstract Object stringToValue(String` *`s`*`)`
  Преобразует символьную строку в значение. Генерирует исключение типа `ParseException`, если формат строки *`s`* неподходящий.
- `DocumentFilter getDocumentFilter()`
  Этот метод следует переопределить, чтобы создать фильтр документа, ограничивающий ввод данных в текстовом поле. Возвращаемое пустое значение `null` указывает на то, что фильтрация не нужна.

javax.swing.text.DefaultFormatter 1.3

- `boolean getOverwriteMode()`
- `void setOverwriteMode(boolean` *`mode`*`)`
  Получают или устанавливают режим перезаписи. Если параметр *`mode`* принимает логическое значение `true`, новые символы перезапишут прежние символы при редактировании текста.

javax.swing.text.DocumentFilter 1.4

- `void insertString(DocumentFilter.FilterBypass` *bypass*, `int` *offset*, `String` *text*, `AttributeSet` *attrib*)

  Вызывается перед вставкой символьной строки в документ. Этот метод можно переопределить и видоизменить символьную строку. Вставку можно отменить, не вызывая метод `super.insertString()` или же вызывая методы объекта *bypass*, чтобы видоизменить документ без фильтрации.

Параметры:		
	bypass	Объект, позволяющий выполнять команды редактирования в обход фильтра
	offset	Смещение, с которого начинается вставка текста
	text	Вставляемые символы
	attrib	Атрибуты форматирования вставляемого текста

- `void replace(DocumentFilter.FilterBypass` *bypass*, `int` *offset*, `int` *length*, `String` *text*, `AttributeSet` *attrib*)

  Вызывается перед заменой части документа новой символьной строкой. Этот метод можно переопределить и видоизменить символьную строку. Замену можно отменить, не вызывая метод `super.replace()` или же вызывая объект *bypass*, чтобы видоизменить документ без фильтрации.

Параметры:		
	bypass	Объект, позволяющий выполнять команды редактирования в обход фильтра
	offset	Смещение, с которого начинается замена части документа
	length	Длина заменяемой части документа
	text	Заменяющие символы
	attrib	Атрибуты форматирования заменяющего текста

- `void remove(DocumentFilter.FilterBypass` *bypass*, `int` *offset*, `int` *length*)

  Вызывается перед удалением части документа. Если требуется проанализировать результат удаления, документ можно получить, вызвав метод `bypass.getDocument()`.

Параметры:	`bypass`	Объект, позволяющий выполнять команды редактирования в обход фильтра
	`offset`	Смещение, с которого начинается удаление части документа
	`length`	Длина удаляемой части документа

javax.swing.text.MaskFormatter 1.4

- `MaskFormatter(String mask)`

  Создает средство форматирования по заданному шаблону. Символы, которые допускается употреблять в шаблоне, перечислены в табл. 6.3.

- `String getValidCharacters()`
- `void setValidCharacters(String characters)`

  Получают или устанавливают символы, допустимые для редактирования. Для переменных частей шаблона допускается использовать только символы из указанной строки.

- `String getInvalidCharacters()`
- `void setInvalidCharacters(String characters)`

  Получают или устанавливают символ-заполнитель, используемый для тех переменных символов в шаблоне, которые пользователь еще не ввел. По умолчанию символом-заполнителем служит пробел.

- `char getPlaceholderCharacter()`
- `void setPlaceholderCharacter(char ch)`

  Получают или устанавливают строку-заполнитель. Конечная часть этой строки используется в том случае, если пользователь не ввел все переменные символы в шаблоне. Если эта строка имеет пустое значение `null` или оказывается короче шаблона, остальная часть вводимых данных заполняется символом-заполнителем.

- `boolean getValueContainsLiteralCharacters()`
- `void setValueContainsLiteralCharacters(boolean b)`

  Получают или устанавливают признак "значение содержит буквенные символы". Если этот признак принимает логическое значение `true`, то значение из поля содержит буквенные (неизменяемые) части шаблона. А если этот признак принимает логическое значение `false`, то буквенные символы удаляются. По умолчанию признак принимает логическое значение `true`.

Компонент JSpinner

Компонент `JSpinner` содержит счетчик в виде текстового поля и двух небольших кнопок со стрелками справа от него. Если щелкнуть на этих кнопках, значение текстового поля увеличится или уменьшится на заданную величину (рис. 6.38).

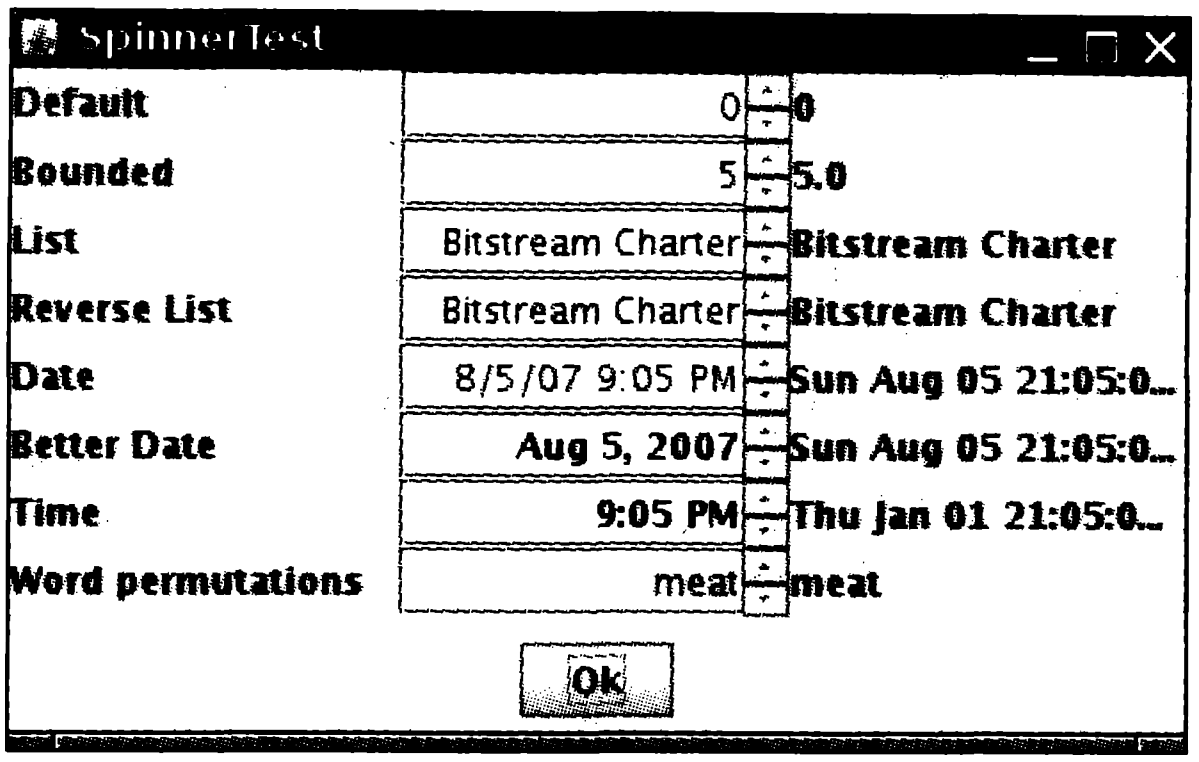


Рис. 6.38. Несколько вариантов исполнения компонента `JSpinner`

В качестве значений в счетчике могут использоваться числа, даты, значения, выбираемые из списка, или, как это чаще всего бывает, любая последовательность значений, для которых можно определить предшествующие и последующие значения. Для первых трех случаев в классе `JSpinner` определяются стандартные модели данных. Имеется также возможность определить собственную модель данных для описания произвольных последовательностей значений.

По умолчанию счетчик оперирует целыми числами, а его кнопки увеличивают или уменьшают целое число на **1**. Текущее значение в счетчике можно получить, вызвав метод `getValue()`. Этот метод возвращает объект типа `Object`. Его нужно сначала привести к типу `Integer`, а затем извлечь упакованное значение, как показано ниже.

```
JSpinner defaultSpinner = new JSpinner();
. . .
int value = (Integer) defaultSpinner.getValue();
```

Исходную величину приращения в счетчике на **1** можно изменить, задав нижние и верхние пределы приращения. Ниже приведен пример реализации в коде счетчика с начальным значением **5**, которое может изменяться в пределах от **0** и **10** с приращением **0,5**.

```
JSpinner boundedSpinner =
    new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
```

Для класса `SpinnerNumberModel` предусмотрены два конструктора, причем один из них имеет только параметры типа `int`, а другой — только параметры типа `double`. Если любой из этих параметров принимает числовое значение с плавающей запятой, то используется второй конструктор. Он выбирает объект типа `Double` для представления значений в счетчике.

Применение счетчиков не ограничивается только числовыми значениями. Имеется также возможность создать счетчик для перебора любого набора значений. Для этого достаточно передать модель типа `SpinnerListModel` конструктору класса `JSpinner`. Модель типа `SpinnerListModel` можно создать на основе любого массива

или класса, реализующего интерфейс List (например, ArrayList). Так, в рассматриваемом здесь примере программы демонстрируется счетчик со всеми доступными именами шрифтов:

```
String[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().
        getAvailableFontFamilyNames();
JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
```

Но направление приращения в счетчике оказалось нетипичным, поскольку оно отличалось от привычного для пользователей направления выбора значений в комбинированных списках. В комбинированном списке большие значения обычно располагаются *ниже* меньших значений, поэтому можно было вполне обоснованно ожидать, что после щелчка в счетчике на кнопке со стрелкой вниз будет выбрано большее значение. Но приращение счетчика происходит при увеличении индекса массива, а следовательно, большее значение получается после щелчка на кнопке со стрелкой вверх. Изменить направление приращения в модели типа SpinnerListModel нельзя, но желаемого результата можно все же добиться с помощью следующего анонимного подкласса:

```
JSpinner reverseListSpinner = new JSpinner(
    new SpinnerListModel(fonts)
  {
    public Object getNextValue()
    {
      return super.getPreviousValue();
    }

    public Object getPreviousValue()
    {
      return super.getNextValue();
    }
  });
```

Опробуйте оба варианта счетчика, чтобы выбрать наиболее интуитивный из них. Еще одно полезное применение счетчик находит для выбора дат. Создать такой счетчик с текущей датой в качестве начальной можно следующим образом:

```
JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
```

Но если посмотреть внимательно на рис. 6.38, то можно заметить, что текст в счетчике представляет как дату, так и время: **8/05/07 9:05 PM**. Для выбора даты время не так уж и важно. А для того чтобы счетчик показывал только дату, потребуется следующий код:

```
JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
String pattern =
    ((SimpleDateFormat) DateFormat.getDateInstance()).toPattern();
betterDateSpinner.setEditor(
    new JSpinner.DateEditor(betterDateSpinner, pattern));
```

Аналогичным образом можно выбирать из счетчика только время:

```
JSpinner timeSpinner = new JSpinner(new SpinnerDateModel());
pattern = ((SimpleDateFormat)
    DateFormat.getTimeInstance(DateFormat.SHORT)).toPattern();
timeSpinner.setEditor(new JSpinner.DateEditor(timeSpinner, pattern));
```

В счетчике можно отображать произвольные последовательности, определив собственную модель счетчика. В рассматриваемом здесь примере программы

демонстрируется счетчик, предоставляющий все возможные варианты перестановки букв в слове "meat". Так, из него можно выбрать слова "mate", "meta", "team" и еще двадцать вариантов перестановки, щелкая на кнопках счетчика.

Для определения собственной модели счетчика придется расширить класс `AbstractSpinnerModel` и определить следующие четыре метода:

```
Object getValue()
void setValue(Object value)
Object getNextValue()
Object getPreviousValue()
```

Метод `getValue()` должен вызывать метод `fireStateChanged()` после установки нового значения. А метод `setValue()` устанавливает новое значение. Он должен генерировать исключение типа `IllegalArgumentException`, если новое значение оказывается неподходящим. Методы `getNextValue()` и `getPreviousValue()` возвращают значения, которые должны следовать после или перед текущим значением, или же пустое значение `null`, если достигнут конец последовательности.

ВНИМАНИЕ! После установки нового значения метод `setValue()` должен вызвать метод `fireStateChanged()`. В противном случае поле счетчика не будет обновлено.

ВНИМАНИЕ! Методы `getNextValue()` и `getPreviousValue()` не должны изменять текущее значение. Когда пользователь щелкает на кнопке счетчика со стрелкой вверх, вызывается метод `getNextValue()`. Если возвращаемое значение будет отличаться от пустого значения `null`, оно будет далее задано при вызове метода `setValue()`.

В рассматриваемом здесь примере программы для определения следующего и предыдущего вариантов перестановки букв в слове применяется стандартный алгоритм, как показано в листинге 6.24. Внутренний механизм работы этого алгоритма в данном случае не имеет особого значения. А в листинге 6.23 демонстрируется, каким образом создаются различные виды счетчиков. Для того чтобы увидеть значения в счетчиках, следует щелкнуть на кнопке OK.

Листинг 6.23. Исходный код из файла `spinner/SpinnerFrame.java`

```
1  package spinner;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.text.*;
6  import javax.swing.*;
7
8  /**
9   * Фрейм с панелью, содержащей несколько счетчиков и
10  * кнопку для отображения значений в счетчиках
11 */
12 public class SpinnerFrame extends JFrame
13 {
14   private JPanel mainPanel;
15   private JButton okButton;
16
17   public SpinnerFrame()
18   {
```

```
19      JPanel buttonPanel = new JPanel();
20      okButton = new JButton("Ok");
21      buttonPanel.add(okButton);
22      add(buttonPanel, BorderLayout.SOUTH);
23
24      mainPanel = new JPanel();
25      mainPanel.setLayout(new GridLayout(0, 3));
26      add(mainPanel, BorderLayout.CENTER);
27      JSpinner defaultSpinner = new JSpinner();
28      addRow("Default", defaultSpinner);
29
30      JSpinner boundedSpinner =
31         new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
32      addRow("Bounded", boundedSpinner);
33
34      String[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment()
35         .getAvailableFontFamilyNames();
36
37      JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
38      addRow("List", listSpinner);
39
40      JSpinner reverseListSpinner = new JSpinner(new SpinnerListModel(fonts)
41        {
42          public Object getNextValue() { return super.getPreviousValue(); }
43          public Object getPreviousValue() { return super.getNextValue(); }
44        });
45      addRow("Reverse List", reverseListSpinner);
46
47      JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
48      addRow("Date", dateSpinner);
49
50      JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
51      String pattern =
52         ((SimpleDateFormat) DateFormat.getDateInstance()).toPattern();
53      betterDateSpinner.setEditor(new JSpinner.DateEditor(
54         betterDateSpinner, pattern));
55      addRow("Better Date", betterDateSpinner);
56
57      JSpinner timeSpinner = new JSpinner(new SpinnerDateModel());
58      pattern = ((SimpleDateFormat)
59         DateFormat.getTimeInstance(DateFormat.SHORT)).toPattern();
60      timeSpinner.setEditor(new JSpinner.DateEditor(timeSpinner, pattern));
61      addRow("Time", timeSpinner);
62
63      JSpinner permSpinner =
64         new JSpinner(new PermutationSpinnerModel("meat"));
65      addRow("Word permutations", permSpinner);
66      pack();
67    }
68
69    /**
70     * Ввести ряд счетчиков на основной панели
71     * @param labelText Метка счетчика
72     * @param spinner Образцовый счетчик
73    */
74    public void addRow(String labelText, final JSpinner spinner)
75    {
76      mainPanel.add(new JLabel(labelText));
```

```
77      mainPanel.add(spinner);
78      final JLabel valueLabel = new JLabel();
79      mainPanel.add(valueLabel);
80      okButton.addActionListener(new ActionListener()
81        {
82          public void actionPerformed(ActionEvent event)
83          {
84            Object value = spinner.getValue();
85            valueLabel.setText(value.toString());
86          }
87        });
88    }
89 }
```

Листинг 6.24. Исходный код из файла `spinner/PermutationSpinnerModel.java`

```
1  package spinner;
2
3  import javax.swing.*;
4
5  /**
6   * Модель, динамически формирующая перестановки слов
7  */
8  public class PermutationSpinnerModel extends AbstractSpinnerModel
9  {
10   private String word;
11
12   /**
13    * Конструирует модель
14    * @param w Переставляемое слово
15   */
16   public PermutationSpinnerModel(String w)
17   {
18     word = w;
19   }
20
21   public Object getValue()
22   {
23     return word;
24   }
25
26   public void setValue(Object value)
27   {
28     if (!(value instanceof String)) throw new IllegalArgumentException();
29     word = (String) value;
30     fireStateChanged();
31   }
32
33   public Object getNextValue()
34   {
35     int[] codePoints = toCodePointArray(word);
36
37     for (int i = codePoints.length - 1; i > 0; i--)
38     {
39       if (codePoints[i - 1] < codePoints[i])
40       {
41         int j = codePoints.length - 1;
```

```
42          while (codePoints[i - 1] > codePoints[j])
43            j--;
44          swap(codePoints, i - 1, j);
45          reverse(codePoints, i, codePoints.length - 1);
46          return new String(codePoints, 0, codePoints.length);
47        }
48      }
49      reverse(codePoints, 0, codePoints.length - 1);
50      return new String(codePoints, 0, codePoints.length);
51    }
52
53    public Object getPreviousValue()
54    {
55      int[] codePoints = toCodePointArray(word);
56      for (int i = codePoints.length - 1; i > 0; i--)
57      {
58        if (codePoints[i - 1] > codePoints[i])
59        {
60          int j = codePoints.length - 1;
61          while (codePoints[i - 1] < codePoints[j])
62            j--;
63          swap(codePoints, i - 1, j);
64          reverse(codePoints, i, codePoints.length - 1);
65          return new String(codePoints, 0, codePoints.length);
66        }
67      }
68      reverse(codePoints, 0, codePoints.length - 1);
69      return new String(codePoints, 0, codePoints.length);
70    }
71
72    private static int[] toCodePointArray(String str)
73    {
74      int[] codePoints = new int[str.codePointCount(0, str.length())];
75      for (int i = 0, j = 0; i < str.length(); i++, j++)
76      {
77        int cp = str.codePointAt(i);
78        if (Character.isSupplementaryCodePoint(cp)) i++;
79        codePoints[j] = cp;
80      }
81      return codePoints;
82    }
83    private static void swap(int[] a, int i, int j)
84    {
85      int temp = a[i];
86       a[i] = a[j];
87       a[j] = temp;
88    }
89
90    private static void reverse(int[] a, int i, int j)
91    {
92      while (i < j)
93      {
94        swap(a, i, j);
95        i++;
96        j--;
97      }
98    }
99  }
```

javax.swing.JSpinner 1.4

- `JSpinner()`

  Создает счетчик, редактирующий целочисленное значение, начиная с **0**, с приращением **1** и без ограничений.

- `JSpinner(SpinnerModel` *`model`*`)`

  Создает счетчик, использующий указанную мóдель данных.

- `Object getValue()`

  Получает текущее значение счетчика.

- `void setValue(Object` *`value`*`)`

  Пытается установить значение счетчика. Генерирует исключение типа `IllegalArgumentException`, если модель не принимает это значение.

- `void setEditor(JComponent` *`editor`*`)`

  Устанавливает компонент, используемый для редактирования значения счетчика.

javax.swing.SpinnerNumberModel 1.4

- `SpinnerNumberModel(int` *`initval`*`, int` *`minimum`*`, int` *`maximum`*`, int` *`stepSize`*`)`
- `SpinnerNumberModel(double` *`initval`*`, double` *`minimum`*`, double` *`maximum`*`, double` *`stepSize`*`)`

  Эти конструкторы позволяют создавать модели, управляющие значениями типа `Integer` или `Double`. Для значений, не имеющих ограничений, следует использовать константы `MIN_VALUE` и `MAX_VALUE` из классов `Integer` и `Double`.

Параметры:		
	`initval`	Исходное значение
	`minimum`	Минимально допустимое значение
	`maximum`	Максимально допустимое значение
	`stepSize`	Положительное или отрицательное приращение каждого счетчика

javax.swing.SpinnerListModel 1.4

- `SpinnerListModel(Object[]` *`values`*`)`
- `SpinnerListModel(List` *`values`*`)`
- Эти конструкторы позволяют создавать модели для выбора среди заданных значений.

javax.swing.SpinnerDateModel 1.4

- `SpinnerDateModel()`
  Создает модель даты, в которой в качестве исходного значения служит текущая дата, отсутствуют верхние или нижние границы, а в качестве приращения принято значение константы `Calendar.DAY_OF_MONTH`.
- `SpinnerDateModel(Date` *`initval`*`, Comparable` *`minimum`*`, Comparable` *`maximum`*`, int` *`step`*`)`

Параметры:		
	`initval`	Исходное значение
	`minimum`	Минимально допустимое значение или пустое значение `null`, если нижняя граница не требуется
	`maximum`	Максимально допустимое значение или пустое значение `null`, если верхняя граница не требуется
	`step`	Дата, увеличиваемая или уменьшаемая при соответствующем приращении счетчика. Принимает значение одной из следующих констант из класса `Calendar`: `ERA`, `YEAR`, `MONTH`, `WEEK_OF_YEAR`, `WEEK_OF_MONTH`, `DAY_OF_MONTH`, `DAY_OF_YEAR`, `DAY_OF_WEEK`, `DAY_OF_WEEK_IN_MONTH`, `AM_PM`, `HOUR`, `HOUR_OF_DAY`, `MINUTE`, `SECOND` или `MILLISECOND`

java.text.SimpleDateFormat 1.1

- `String toPattern()` **1.2**
  Получает шаблон редактирования для данного средства форматирования даты. Обычно используется следующий шаблон: `"yyyy-MM-dd"`. Подробнее об этом шаблоне см. в документации на Java SE.

javax.swing.JSpinner.DateEditor 1.4

- `DateEditor(JSpinner` *spinner*`, String` *pattern*`)`

  Создает редактор даты для счетчика.

  Параметры:

spinner	Счетчик, которому принадлежит данный редактор
pattern	Шаблон формата для связанного с ним объекта типа `SimpleDateFormat`

javax.swing.AbstractSpinnerModel 1.4

- `Object getValue()`
- Получает текущее значение из модели счетчика.
- `void setValue(Object` *value*`)`
- Пытается установить новое значение для модели счетчика. Генерирует исключение типа `IllegalArgumentException`, если значение неприемлемо. Переопределяя этот метод, следует вызвать метод `fireStateChanged()` после установки нового значения.
- `Object getNextValue()`
- `Object getPreviousValue()`
- Вычисляют (но не устанавливают) последующее или предыдущее значение в последовательности, определяемой в модели счетчика.

Отображение HTML-документов средствами JEditorPane

В отличие от рассмотренных до сих пор текстовых компонентов, компонент `JEditorPane` позволяет отображать и редактировать текст в формате HTML и RTF. (Формат RTF (Rich Text Format — расширенный текстовый формат) используется в приложениях корпорации Microsoft для обмена документами. Этот формат не очень хорошо документирован и не вполне удовлетворительно работает даже в приложениях Microsoft. Поэтому в данной книге он не рассматривается.)

Функциональные возможности компонента `JEditorPane` ограничены. Средство воспроизведения данных в формате HTML способно отображать содержимое простых файлов, но не справляется с отображением сложных страниц, которые нередко встречаются в веб. К тому же этот компонент работает как HTML-редактор неустойчиво.

Наиболее приемлемое применение компоненту `JEditorPane` можно найти для отображения справочной информации в формате HTML. Содержимое справочных файлов приходится формировать самостоятельно, поэтому при отображении данных в формате HTML можно без особого труда обойти все ограничения, присущие компоненту `JEditorPane`.

НА ЗАМЕТКУ! Чтобы получить более подробные сведения об инструментальных средствах для создания справочных систем промышленного образца, обращайтесь по адресу `http://javahelp.java.net`.

В листинге 6.25 представлен исходный код примера программы, в рабочем окне которой содержится панель редактирования. Эта панель способна отображать HTML-страницы. Для этого в поле редактирования следует ввести URL, начинающийся с префикса **`http:`** или **`file:`**, а затем щелкнуть на кнопке Load (Загрузить). В итоге на панели редактирования появится указанная страница (рис. 6.39).

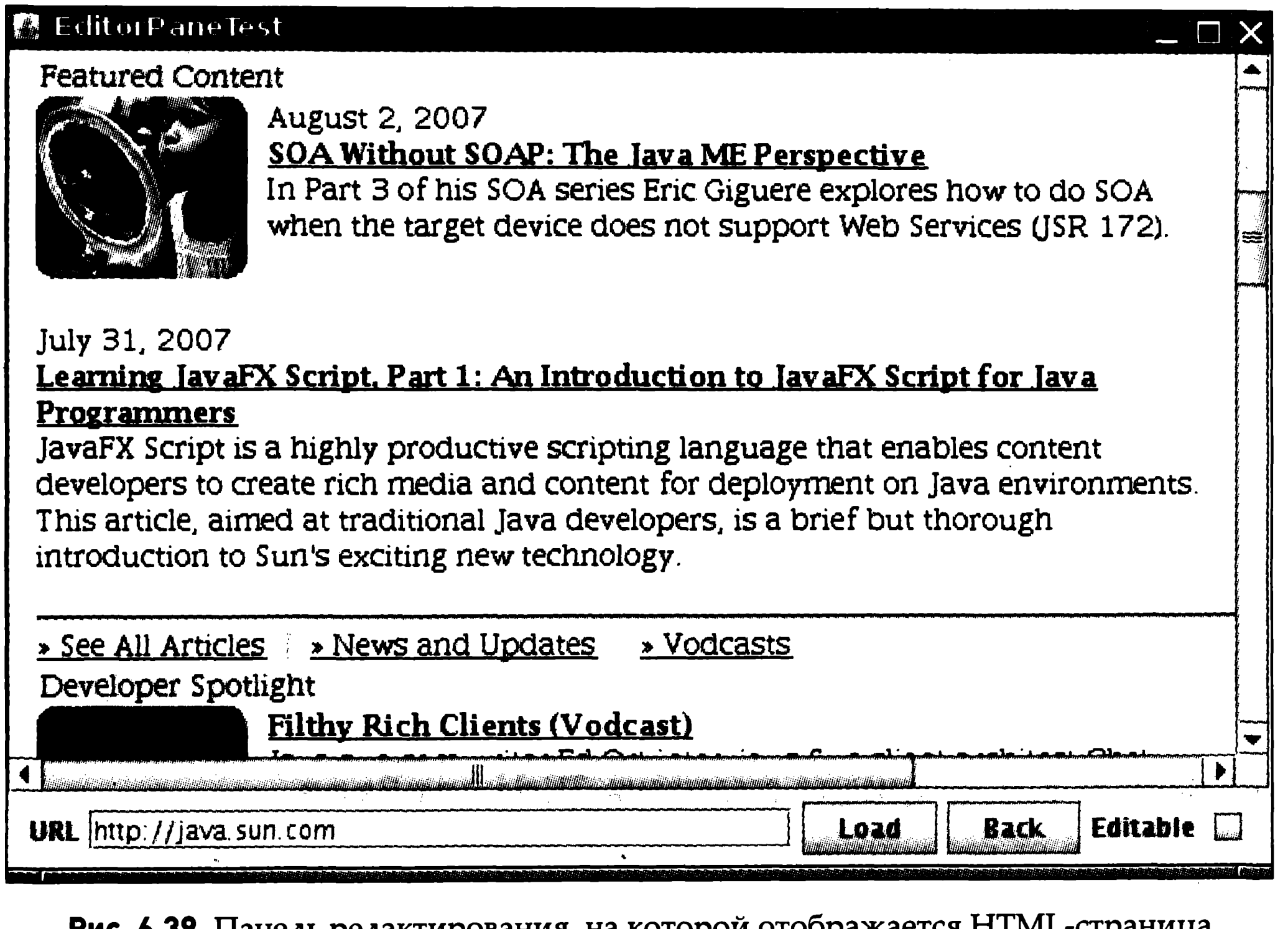


Рис. 6.39. Панель редактирования, на которой отображается HTML-страница

Все гипертекстовые ссылки на этой панели активны. Это означает, что если щелкнуть на любой ссылке, то данная программа загрузит указанную веб-страницу. А если щелкнуть на кнопке Back (Назад), то на панели появится предыдущая веб-страница.

Данная программа, по существу, представляет собой простейший веб-браузер. Безусловно, в нем не поддерживаются специальные возможности, например кеширование страниц или список использованных ссылок, характерные для коммерческих веб-браузеров. Кроме того, на панели редактирования этой программы нельзя отображать аплеты.

Если установить флажок Editable (Редактируемый), содержимое этой панели разрешается редактировать. Теперь на ней можно вводить текст, используя клавишу <Backspace> для удаления символов. Этот компонент способен также обрабатывать стандартные комбинации клавиш <Ctrl+X>, <Ctrl+C> и <Ctrl+V>, которые выполняют

операции вырезания, копирования и вставки текста. Но для того чтобы реализовать поддержку шрифтов и форматирование текста, придется затратить немало труда на программирование соответствующих функций.

При редактировании гипертекстовые ссылки в компоненте не активны. Кроме того, в режиме редактирования на некоторых веб-страницах будут видны команды сценария JavaScript, комментарии и другие дескрипторы (рис. 6.40). В рассматриваемом здесь примере программы режим редактирования разрешается использовать в учебных целях, но для реальных программ он не подходит.

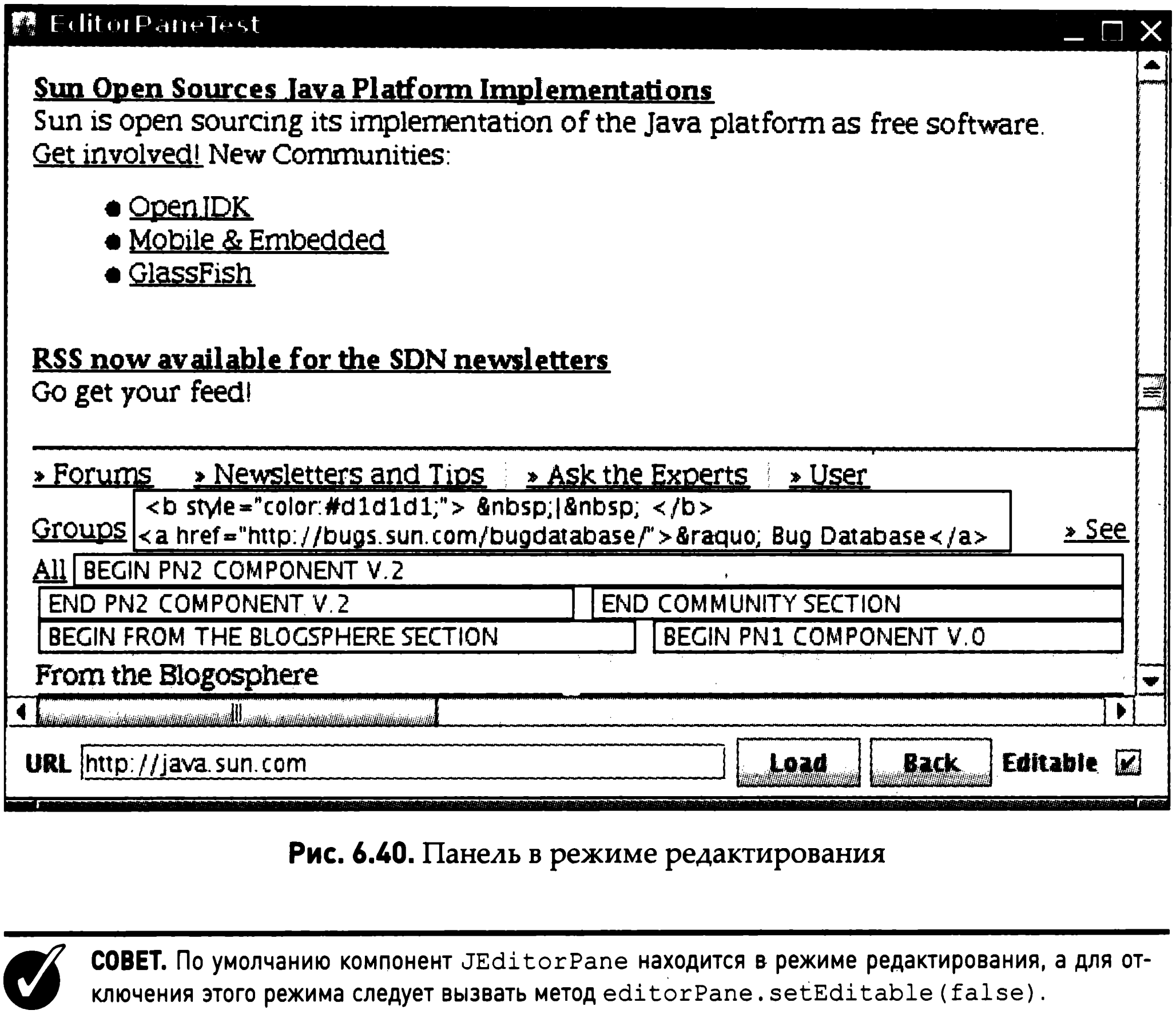


Рис. 6.40. Панель в режиме редактирования

СОВЕТ. По умолчанию компонент `JEditorPane` находится в режиме редактирования, а для отключения этого режима следует вызвать метод `editorPane.setEditable(false)`.

Основные функции панели редактирования в рассматриваемом здесь примере программы достаточно просты. Так, для загрузки документов вызывается метод `setPage()` с параметром в виде символьной строки, содержащей URL, или объекта типа `URL`, как показано ниже. Класс `JEditorPane` расширяет класс `JTextComponent`, поэтому для отображения обычного текста можно вызывать метод `setText()` из его суперкласса.

```
JEditorPane editorPane = new JEditorPane();
editorPane.setPage(url);
```

СОВЕТ. Из документации на прикладной интерфейс API трудно понять, загружает ли метод `setPage()` новый документ в отдельном потоке исполнения, что было бы очень кстати, поскольку компонент `JEditorPane` нельзя назвать сверхбыстродействующим. Для того чтобы явно указать на необходимость загрузки документа в отдельном потоке исполнения, можно воспользоваться следующим фрагментом кода:

```
AbstractDocument doc = (AbstractDocument) editorPane.getDocument();
doc.setAsynchronousLoadPriority(0);
```

Для обработки событий от щелчков кнопкой мыши на гипертекстовых ссылках следует ввести соответствующий обработчик типа `HyperlinkListener`. У интерфейса `HyperlinkListener` имеется единственный метод `hyperlinkUpdate()` с параметром типа `HyperlinkEvent`. Этот метод вызывается при наведении курсора и щелчке кнопкой мыши на гипертекстовой ссылке. А для определения вида наступившего события следует вызвать метод `getEventType()`. Он возвращает одно из перечисленных ниже значений.

```
HyperlinkEvent.EventType.ACTIVATED
HyperlinkEvent.EventType.ENTERED
HyperlinkEvent.EventType.EXITED
```

Значение `HyperlinkEvent.EventType.ACTIVATED` обозначает щелчок кнопкой мыши на гипертекстовой ссылке. В этом случае обычно требуется открыть указанную веб-страницу. А значения `HyperlinkEvent.EventType.ENTERED` и `HyperlinkEvent.EventType.EXITED` обычно используются для организации визуальной обратной связи, например, для изменения внешнего вида ссылки в тот момент, когда пользователь наводит на нее курсор мыши.

НА ЗАМЕТКУ! Не меньшей загадкой остается и то, почему разработчики библиотеки Swing не предусмотрели в интерфейсе `HyperlinkListener` три отдельных метода для обработки событий, связанных с активизацией гипертекстовой ссылки, наведением и отведением курсора от ссылки.

Метод `getURL()` из класса `HyperlinkEvent` возвращает URL по выбранной гипертекстовой ссылке. Ниже приведен пример установки обработчика событий, наступающих при активизации пользователем гипертекстовых ссылок. Этот обработчик событий извлекает URL и обновляет панель редактирования. Метод `setPage()` может сгенерировать исключение типа `IOException`. В этом случае обычным текстом выводится сообщение об ошибке.

```
editorPane.addHyperlinkListener(new
  HyperlinkListener()
  {
    public void hyperlinkUpdate(HyperlinkEvent event)
    {
      if (event.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
      {
        try
        {
          editorPane.setPage(event.getURL());
        }
        catch (IOException e)
        {
```

```
            editorPane.setText("Exception: " + e);
          }
        }
      }
    });
```

В примере программы из листинга 6.25 демонстрируются все основные функции панели редактирования, необходимые для создания справочной системы с выводом текста в формате HTML. Компонент `JEditorPane` имеет более сложную внутреннюю структуру, чем компоненты построения таблиц или деревьев. Но если вам не требуется создавать на его основе текстовые редакторы и средства воспроизведения текста в специальных форматах, то весь сложный внутренний механизм работы этого компонента будет скрыт от вас.

Листинг 6.25. Исходный текст из файла `editorPane/EditorPaneFrame.java`

```
1  package editorPane;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.io.*;
6  import java.util.*;
7  import javax.swing.*;
8  import javax.swing.event.*;
9
10 /**
11  * Этот фрейм содержит панель редактирования, текстовое поле и
12  * кнопку для ввода URL и загрузки документа, а также кнопку
13  * для возврата к предыдущему загруженному документу
14 */
15 public class EditorPaneFrame extends JFrame
16 {
17   private static final int DEFAULT_WIDTH = 600;
18   private static final int DEFAULT_HEIGHT = 400;
19
20   public EditorPaneFrame()
21   {
22     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24     final Stack<String> urlStack = new Stack<>();
25     final JEditorPane editorPane = new JEditorPane();
26     final JTextField url = new JTextField(30);
27
28     // установить приемник событий от гиперссылок
29
30     editorPane.setEditable(false);
31     editorPane.addHyperlinkListener(new HyperlinkListener()
32       {
33         public void hyperlinkUpdate(HyperlinkEvent event)
34         {
35           if (event.getEventType() ==
36              HyperlinkEvent.EventType.ACTIVATED)
37           {
38             try
39             {
40               // запомнить URL для кнопки возврата
41               urlStack.push(event.getURL().toString());
```

```
42                // показать URL в текстовом поле
43                url.setText(event.getURL().toString());
44                editorPane.setPage(event.getURL());
45              }
46              catch (IOException e)
47              {
48                editorPane.setText("Exception: " + e);
49              }
50            }
51          }
52        });
53
54      // настроить флажок для переключения режима редактирования
55
56      final JCheckBox editable = new JCheckBox();
57      editable.addActionListener(new ActionListener()
58        {
59          public void actionPerformed(ActionEvent event)
60          {
61            editorPane.setEditable(editable.isSelected());
62          }
63        });
64
65       // настроить кнопку для загрузки документа по заданному URL
66
67      ActionListener listener = new ActionListener()
68        {
69          public void actionPerformed(ActionEvent event)
70          {
71            try
72            {
73              // запомнить URL для кнопки возврата
74              urlStack.push(url.getText());
75              editorPane.setPage(url.getText());
76            }
77            catch (IOException e)
78            {
79              editorPane.setText("Exception: " + e);
80            }
81          }
82        };
83
84      JButton loadButton = new JButton("Load");
85      loadButton.addActionListener(listener);
86      url.addActionListener(listener);
87
88      // настроить кнопку возврата и ее действие
89      JButton backButton = new JButton("Back");
90      backButton.addActionListener(new ActionListener()
91        {
92          public void actionPerformed(ActionEvent event)
93          {
94            if (urlStack.size() <= 1) return;
95            try
96            {
97              // получить URL из кнопки возврата
98              urlStack.pop();
99              // показать URL в текстовом поле
100             String urlString = urlStack.peek();
```

```
101             url.setText(urlString);
102             editorPane.setPage(urlString);
103          }
104          catch (IOException e)
105          {
106             editorPane.setText("Exception: " + e);
107          }
108       }
109    });
110
111    add(new JScrollPane(editorPane), BorderLayout.CENTER);
112
113    // разместить все элементы управления на панели
114
115    JPanel panel = new JPanel();
116    panel.add(new JLabel("URL"));
117    panel.add(url);
118    panel.add(loadButton);
119    panel.add(backButton);
120    panel.add(new JLabel("Editable"));
121    panel.add(editable);
122
123    add(panel, BorderLayout.SOUTH);
124  }
125 }
```

javax.swing.JEditorPane 1.2

- `void setPage(URL url)`

  Загружает на панели редактирования веб-страницу по указанному URL.

- `void addHyperlinkListener(HyperLinkListener listener)`
- Добавляет к панели редактирования приемник событий от гипертекстовых ссылок.

`javax.swing.event.HyperlinkListener 1.2`

- `void hyperlinkUpdate(HyperlinkEvent event)`

  Вызывается всякий раз, когда выбирается гипертекстовая ссылка.

javax.swing.event.HyperlinkEvent 1.2

- `URL getURL()`

  Возвращает URL по выбранной гипертекстовой ссылке.

Индикаторы состояния

В этом разделе рассматриваются три класса, предназначенные для отображения хода выполнения длительных заданий и процессов. В частности, класс `JProgressBar` — это компонент Swing, который наглядно представляет ход

выполнения процесса. Класс `ProgressMonitor` поддерживает диалоговое окно, в котором находится индикатор выполнения, отображаемый с помощью объекта типа `JProgressBar`. А класс `ProgressMonitorInputStream` отображает диалоговое окно контроля текущего состояния при чтении данных из потока ввода.

Индикаторы выполнения

Индикатор выполнения — это широкая прямоугольная полоса, частично заполненная другим цветом. Такой индикатор дает наглядное представление о ходе выполнения операции. По умолчанию в прямоугольной полосе отображается также символьная строка в формате ***n*%**, где ***n*** — число. Пример рабочего окна, в нижней части которого находится индикатор выполнения, приведен на рис. 6.41.

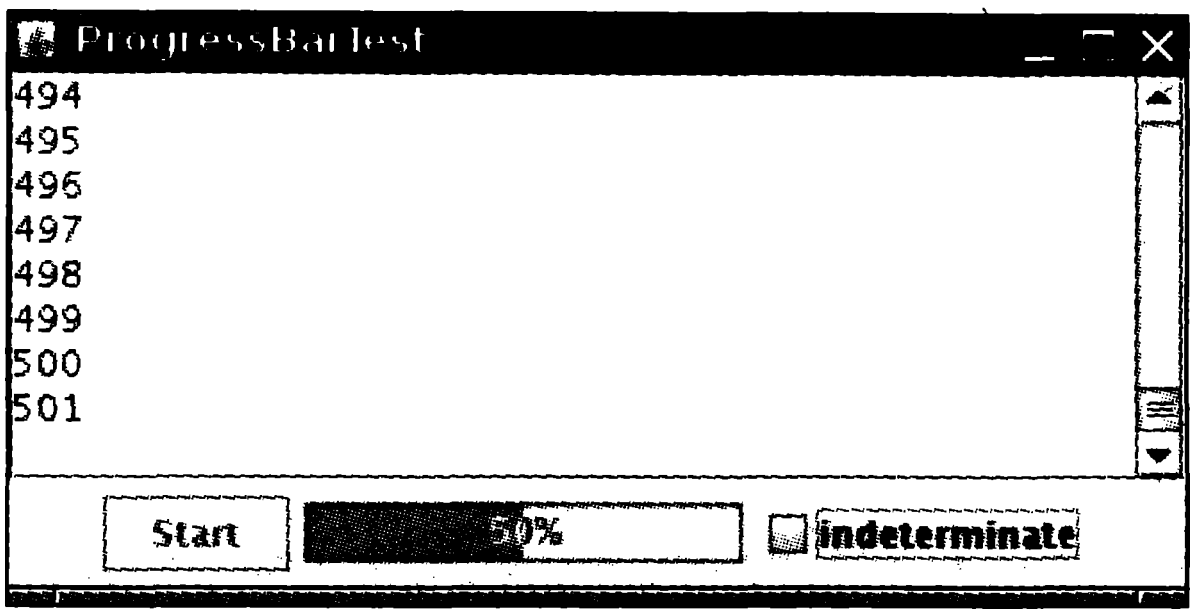


Рис. 6.41. Окно с индикатором выполнения

Индикатор выполнения создается подобно ползунковому регулятору. Для этого конструктору класса `JProgressBar` передается минимальное и максимальное значения, а дополнительно, хотя и необязательно, — ориентация данного компонента:

```
progressBar = new JProgressBar(0, 1000);
progressBar = new JProgressBar(SwingConstants.VERTICAL, 0, 1000);
```

Минимальное и максимальное значения, отображаемые в индикаторе выполнения, можно также задать с помощью методов `setMinimum()` и `setMaximum()`. В отличие от ползункового регулятора, индикатор выполнения не реагирует на действия пользователя. Для того чтобы изменить его состояние, необходимо вызвать метод `setValue()`.

Если вызвать метод `progressBar.setStringPainted(true)`, то в индикаторе будет рассчитана доля выполнения операции в процентах и отображена символьная строка в формате ***n*%**. Если же требуется вывести другую последовательность символов, то соответствующую строку следует передать методу `setString()`, как показано ниже.

```
if (progressBar.getValue() > 900)
  progressBar.setString("Almost Done");
```

В примере программы, исходный код которой приведен в листинге 6.26, демонстрируется индикатор выполнения, предназначенный для визуального контроля за ходом имитируемого длительного процесса. В классе `SimulatedActivity` значение переменной `current` увеличивается 10 раз в секунду. Когда оно достигает конечной величины, задаваемой в переменной `target`, выполнение имитируемого процесса завершается. В данной программе класс `SwingWorker` используется для того, чтобы

сымитировать процесс и обновить индикатор его выполнения в методе `process()`. Этот метод вызывается в классе `SwingWorker` из потока диспетчеризации событий, что дает возможность безопасно обновлять индикатор выполнения. (Более подробно вопросы обеспечения безопасности потоков исполнения в Swing рассматриваются в главе 14 первого тома данной книги.)

В версии JDK 1.4 был реализован индикатор, уведомляющий о самом факте выполнения процесса. Он обеспечивает вывод анимационного изображения, но не отображает никаких сведений, позволяющих судить о том, какая именно часть процесса выполнена. Подобного рода индикаторы часто применяются в браузерах. Они сообщают, что браузер ожидает ответа от сервера, но не дают ясного представления, сколько это ожидание может продлиться. Для отображения такого индикатора "неопределенного ожидания" следует вызвать метод `setIndeterminate()`. Весь исходный код примера программы, демонстрирующей применение индикатора выполнения, приведен в листинге 6.26.

Листинг 6.26. Исходный код из файла `progressBar/ProgressBarFrame.java`

```
1  package progressBar;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.util.List;
6  import javax.swing.*;
7
8  /**
9   * Фрейм, содержащий кнопку для запуска имитируемого процесса,
10  * индикатор выполнения и текстовую область для вывода результатов
11  */
12 public class ProgressBarFrame extends JFrame
13 {
14   public static final int TEXT_ROWS = 10;
15   public static final int TEXT_COLUMNS = 40;
16
17   private JButton startButton;
18   private JProgressBar progressBar;
19   private JCheckBox checkBox;
20   private JTextArea textArea;
21   private SimulatedActivity activity;
22
23   public ProgressBarFrame()
24   {
25     // в этой текстовой области выводятся результаты
26     // выполнения имитируемого процесса
27     textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
28
29     // установить панель с кнопкой и индикатором выполнения
30
31     final int MAX = 1000;
32     JPanel panel = new JPanel();
33     startButton = new JButton("Start");
34     progressBar = new JProgressBar(0, MAX);
35     progressBar.setStringPainted(true);
36     panel.add(startButton);
37     panel.add(progressBar);
38
```

```
39        checkBox = new JCheckBox("indeterminate");
40        checkBox.addActionListener(new ActionListener()
41          {
42            public void actionPerformed(ActionEvent event)
43            {
44              progressBar.setIndeterminate(checkBox.isSelected());
45              progressBar.setStringPainted(!progressBar.isIndeterminate());
46            }
47          });
48        panel.add(checkBox);
49        add(new JScrollPane(textArea), BorderLayout.CENTER);
50        add(panel, BorderLayout.SOUTH);
51
52        // установить приемник действий кнопки
53
54        startButton.addActionListener(new ActionListener()
55          {
56            public void actionPerformed(ActionEvent event)
57            {
58              startButton.setEnabled(false);
59              activity = new SimulatedActivity(MAX);
60              activity.execute();
61            }
62          });
63        pack();
64      }
65
66      class SimulatedActivity extends SwingWorker<Void, Integer>
67      {
68        private int current;
69        private int target;
70
71        /**
72         * Имитирует процесс инкрементирования счетчика от нуля
73         * до заданного конечного значения
74         * @param t Конечное значение счетчика
75        */
76        public SimulatedActivity(int t)
77        {
78          current = 0;
79          target = t;
80        }
81
82        protected Void doInBackground() throws Exception
83        {
84          try
85          {
86            while (current < target)
87            {
88              Thread.sleep(100);
89              current++;
90              publish(current);
91            }
92          }
93          catch (InterruptedException e)
94          {
95          }
96          return null;
```

```
97        }
98
99        protected void process(List<Integer> chunks)
100       {
101         for (Integer chunk : chunks)
102         {
103           textArea.append(chunk + "\n");
104           progressBar.setValue(chunk);
105         }
106       }
107       protected void done()
108     {
109       startButton.setEnabled(true);
110     }
111   }
112 }
```

Мониторы текущего состояния

Индикатор выполнения является простым компонентом, размещаемым в окне. А класс `ProgressMonitor` реализует полностью диалоговое окно монитора текущего состояния, содержащее индикатор выполнения (рис. 6.42), а также кнопку Cancel. Если пользователь щелкнет на этой кнопке, диалоговое окно монитора закроется. Кроме того, программа, выполнение которой отслеживается монитором, может проверить, закрыто ли диалоговое окно, и завершить свою работу, если пользователь выбрал кнопку Cancel. (Обратите внимание на то, что имя класса, реализующего данный компонент, не начинается с буквы J.)

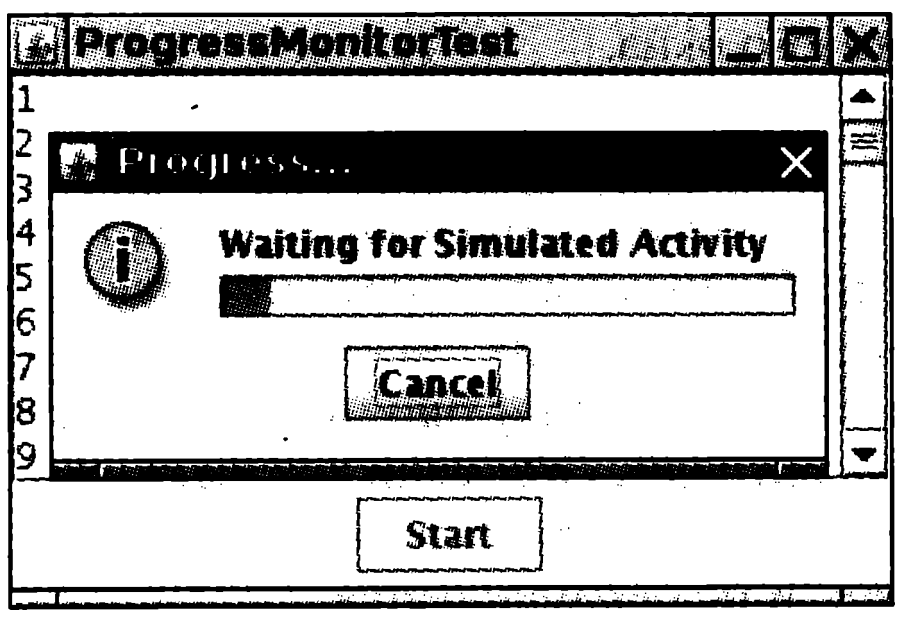


Рис. 6.42. *Диалоговое окно монитора текущего состояния*

Для создания монитора текущего состояния конструктору класса `ProgressMonitor` передаются следующие данные.

- Родительский компонент, поверх которого должно отображаться диалоговое окно монитора.
- Объект (символьная строка, пиктограмма или компонент), отображаемый в диалоговом окне монитора.
- Необязательное примечание, которое выводится ниже данного объекта.
- Минимальное и максимальное значения.

Но монитор текущего состояния не может самостоятельно определить, какая именно часть процесса выполнена, или завершить его. Поэтому состояние диалогового окна монитора необходимо периодически обновлять, вызывая метод `setProgress()`. (Этот метод выполняет те же самые функции, что и метод `setValue()` из класса `JProgressBar`.) Когда контролируемый монитором процесс благополучно завершится, следует вызвать метод `close()`, удалив диалоговое окно монитора с экрана. Впоследствии то же самое окно можно использовать повторно, вызвав метод `start()`.

Самые большие трудности, возникающие при использовании диалогового окна монитора текущего состояния, связаны с обработкой запросов на отмену контролируемого процесса. Дело в том, что к кнопке Cancel нельзя присоединить соответствующий обработчик событий. Вместо этого приходится периодически вызывать метод `isCanceled()`, чтобы выяснить, щелкнул ли пользователь программы на кнопке Cancel.

Если рабочий поток может входить в состояние блокировки на неопределенный срок (например, при чтении введенных данных через сетевое соединение), он не в состоянии следить за кнопкой Cancel. В рассматриваемом здесь примере программы демонстрируется, каким образом для этой цели используется таймер. Кроме того, на таймер возлагается ответственность за обновление замеров текущего состояния и хода выполнения процесса.

Если запустить рассматриваемую здесь программу, исходный код которой приведен в листинге 16.27, то в ходе ее выполнения обнаружится интересная особенность. Диалоговое окно монитора не исчезает, как только завершится контролируемый процесс. Программе нужно сначала убедиться, что имитируемый процесс действительно завершился или, скорее всего, завершится раньше того времени, которое требуется для отображения диалогового окна.

Управление выдержкой времени происходит следующим образом. С помощью метода `setMillisToDecideToPopup()` задается число миллисекунд, которые должны пройти между созданием объекта диалогового окна и принятием решения относительно его отображения. По умолчанию принимается значение **500** миллисекунд. Метод `setMillisToPopup()` служит для оценки времени, требующегося для отображения диалогового окна. Разработчики библиотек Swing установили это значение равным по умолчанию **2** секундам. Они учли тот факт, что диалоговые окна компонентов Swing не появляются так быстро, как хотелось бы. Поэтому изменять это значение без особой надобности не рекомендуется.

Листинг 6.27. Исходный код из файла `progressMonitor/ProgressMonitorFrame.java`

```
1   package progressMonitor;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6
7   /**
8    * Фрейм, содержащий кнопку для запуска имитируемого процесса и
9    * текстовое поле для вывода результатов выполнения этого процесса
10  */
11  class ProgressMonitorFrame extends JFrame
12  {
13    public static final int TEXT_ROWS = 10;
```

```
14    public static final int TEXT_COLUMNS = 40;
15
16    private Timer cancelMonitor;
17    private JButton startButton;
18    private ProgressMonitor progressDialog;
19    private JTextArea textArea;
20    private SimulatedActivity activity;
21    public ProgressMonitorFrame()
22    {
23      // в этой текстовой области выводятся результаты
24      // выполнения имитируемого процесса
25      textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
26
27      // установить панель с кнопками
28      JPanel panel = new JPanel();
29      startButton = new JButton("Start");
30      panel.add(startButton);
31
32      add(new JScrollPane(textArea), BorderLayout.CENTER);
33      add(panel, BorderLayout.SOUTH);
34
35      // установить приемник действий кнопки
36      startButton.addActionListener(new ActionListener()
37        {
38          public void actionPerformed(ActionEvent event)
39          {
40            startButton.setEnabled(false);
41            final int MAX = 1000;
42
43            // начать имитируемый процесс
44            activity = new SimulatedActivity(MAX);
45            activity.execute();
46
47            // открыть диалоговое окно монитора текущего состояния
48            progressDialog = new ProgressMonitor(
49                ProgressMonitorFrame.this,
50                "Waiting for Simulated Activity", null, 0, MAX);
51            cancelMonitor.start();
52          }
53        });
54
55      // установить приемник действий таймера
56      cancelMonitor = new Timer(500, new ActionListener()
57        {
58          public void actionPerformed(ActionEvent event)
59          {
60            if (progressDialog.isCanceled())
61            {
62              activity.cancel(true);
63              startButton.setEnabled(true);
64            }
65            else if (activity.isDone())
66            {
67              progressDialog.close();
68              startButton.setEnabled(true);
69            }
70            else
71            {
```

```
72                progressDialog.setProgress(activity.getProgress());
73              }
74            }
75          });
76        pack();
77    }
78
79    class SimulatedActivity extends SwingWorker<Void, Integer>
80    {
81      private int current;
82      private int target;
83
84      /**
85        * Имитирует процесс инкрементирования счетчика от нуля
86        * до заданного конечного значения
87        * @param t Конечное значение счетчика
88      */
89      public SimulatedActivity(int t)
90      {
91        current = 0;
92        target = t;
93      }
94
95      protected Void doInBackground() throws Exception
96      {
97        try
98        {
99          while (current < target)
100         {
101           Thread.sleep(100);
102           current++;
103           textArea.append(current + "\n");
104           setProgress(current);
105         }
106       }
107       catch (InterruptedException e)
108       {
109       }
110       return null;
111     }
112   }
113 }
```

Контроль чтения данных из потока ввода

В состав библиотеки Swing входит фильтр типа `ProgressMonitorInputStream`, который автоматически отображает диалоговое окно, в котором контролируется процесс чтения данных из потока ввода. Пользоваться фильтром типа `ProgressMonitorInputStream` очень просто. Достаточно ввести его в обычную последовательность фильтруемых потоков ввода-вывода (подробнее о потоках ввода-вывода см. в главе 1). Допустим, требуется прочитать текст из файла. Для этого сначала открывается поток ввода `FileInputStream`, как показано ниже.

```
FileInputStream in = new FileInputStream(f);
```

Обычно этот поток преобразуется в поток ввода `InputStreamReader` следующим образом:

```
InputStreamReader reader = new InputStreamReader(in);
```

Но для того чтобы контролировать чтение данных из потока ввода, нужно сначала преобразовать этот поток в поток ввода с контролем текущего состояния, как показано ниже.

```
ProgressMonitorInputStream progressIn =
    new ProgressMonitorInputStream(parent, caption, in);
```

Конструктору класса `ProgressMonitorInputStream` передаются родительский компонент, заголовок и сам контролируемый поток ввода. В методе `read()` из этого класса анализируется количество прочитанных байтов и обновляется содержимое диалогового окна с монитором текущего состояния потока ввода.

Далее можно продолжить формирование последовательности фильтруемых потоков, как показано в приведенной ниже строке кода. Вот, собственно, и все, что требуется для организации контроля над чтением данных из потока ввода. При чтении из файла автоматически открывается диалоговое окно, показанное на рис. 6.43. Это очень изящное применение фильтрации потоков ввода-вывода.

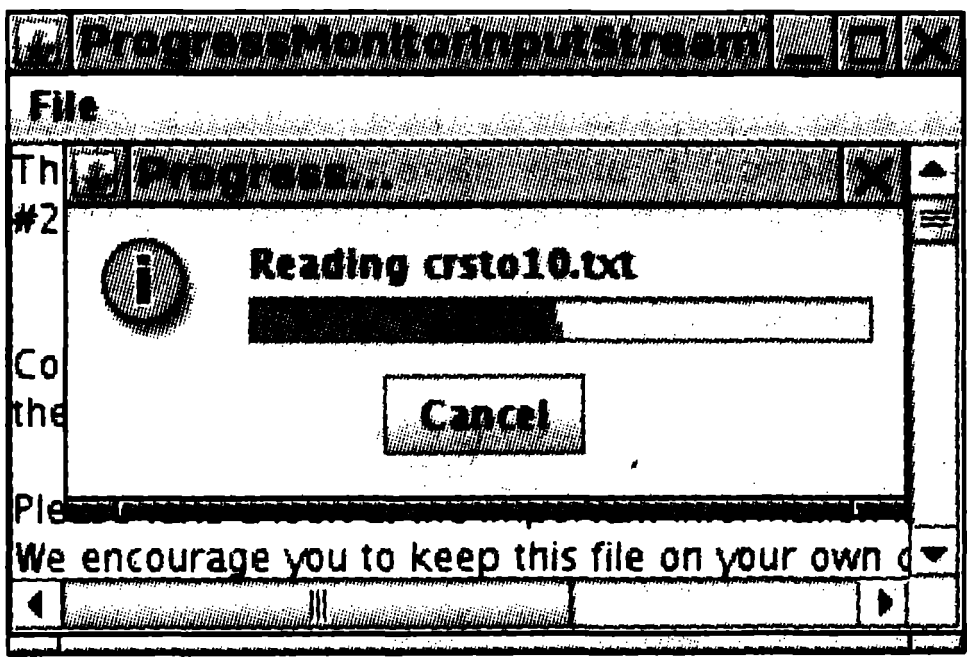


Рис. 6.43. Окно, в котором контролируется чтение данных из потока ввода

НА ЗАМЕТКУ! Для определения общего количества байтов в потоке ввода с контролем текущего состояния служит метод `available()` из класса `InputStream`. Но этот метод возвращает лишь количество байтов, которые могут быть прочитаны из потока ввода *без блокировки*. Таким образом, рассматриваемый здесь монитор текущего состояния потоков ввода вполне пригоден для контроля над чтением данных из файлов и источников в Интернете по протоколу HTTP, поскольку объем этих данных заранее известен. Но, к сожалению, это правило распространяется далеко не на все потоки ввода-вывода.

В примере программы, исходный код которой приведен в листинге 6.28, подсчитывается количество строк в файле. Если файл крупный, то открывается диалоговое окно с монитором текущего состояния процесса чтения данных из файла. Если же пользователь щелкнет на кнопке Cancel, поток ввода закрывается. А поскольку коду, обрабатывающему вводимые данные, заранее известно, что нужно делать по окончании их ввода, то для правильной обработки состояния отмены данного процесса дополнительная логика не требуется.

Следует иметь в виду, что в данной программе применяется далеко не самый эффективный способ заполнения текстовой области. Эта программа работала бы

намного быстрее, если бы в ней можно было прочитать сначала файл в объект типа `StringBuilder`, а затем связать текстовую область с содержимым этого объекта. Но в данном случае медленное выполнение программы вполне уместно, поскольку оно предоставляет достаточно времени, чтобы не спеша оценить выполнение процесса в диалоговом окне. Во избежание мерцания текстовая область не отображается до ее заполнения.

Листинг 6.28. Исходный код из файла `progressMonitorInputStream/TextFrame.java`

```
1   package progressMonitorInputStream;
2
3   import java.awt.event.*;
4   import java.io.*;
5   import java.nio.file.*;
6   import java.util.*;
7   import javax.swing.*;
8
9   /**
10   * Фрейм с меню для загрузки текстового файла и текстовой областью
11   * для отображения его содержимого. Эта область создается при загрузке
12   * файла и устанавливается в виде панели содержимого фрейма по
13   * завершении загрузки во избежание мерцания в ходе данного процесса
14  */
15  public class TextFrame extends JFrame
16  {
17    public static final int TEXT_ROWS = 10;
18    public static final int TEXT_COLUMNS = 40;
19
20    private JMenuItem openItem;
21    private JMenuItem exitItem;
22    private JTextArea textArea;
23    private JFileChooser chooser;
24
25    public TextFrame()
26    {
27      textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
28      add(new JScrollPane(textArea));
29
30      chooser = new JFileChooser();
31      chooser.setCurrentDirectory(new File("."));
32
33      JMenuBar menuBar = new JMenuBar();
34      setJMenuBar(menuBar);
35      JMenu fileMenu = new JMenu("File");
36      menuBar.add(fileMenu);
37      openItem = new JMenuItem("Open");
38      openItem.addActionListener(new ActionListener()
39        {
40          public void actionPerformed(ActionEvent event)
41          {
42            try
43            {
44              openFile();
45            }
46            catch (IOException exception)
47            {
```

```
48                exception.printStackTrace();
49              }
50            }
51          });
52
53        fileMenu.add(openItem);
54        exitItem = new JMenuItem("Exit");
55        exitItem.addActionListener(new ActionListener()
56          {
57            public void actionPerformed(ActionEvent event)
58            {
59              System.exit(0);
60            }
61          });
62         fileMenu.add(exitItem);
63         pack();
64     }
65
66     /**
67      * Приглашает пользователя выбрать файл, загружает его в текстовую
68      * область и устанавливает ее в виде панели содержимого фрейма
69     */
70     public void openFile() throws IOException
71     {
72       int r = chooser.showOpenDialog(this);
73       if (r != JFileChooser.APPROVE_OPTION) return;
74       final File f = chooser.getSelectedFile();
75
76       // установить последовательность фильтруемых потоков ввода данных
77
78       InputStream fileIn = Files.newInputStream(f.toPath());
79       final ProgressMonitorInputStream progressIn =
80          new ProgressMonitorInputStream(
81             this,"Reading " + f.getName(), fileIn);
82
83          textArea.setText("");
84
85       SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>()
86         {
87           protected Void doInBackground() throws Exception
88           {
89             try (Scanner in = new Scanner(progressIn))
90             {
91               while (in.hasNextLine())
92               {
93                 String line = in.nextLine();
94                 textArea.append(line);
95                 textArea.append("\n");
96               }
97             }
98             return null;
99           }
100        };
101      worker.execute();
102    }
103 }
```

javax.swing.JProgressBar 1.2

- `JProgressBar()`
- `JProgressBar(int direction)`
- `JProgressBar(int min, int max)`
- `JProgressBar(int direction, int min, int max)`

Конструируют индикатор выполнения с указанной ориентацией, а также минимальным и максимальным значениями.

Параметры:

direction	Принимает значение одной из следующих констант: `SwingConstants.HORIZONTAL` или `SwingConstants.VERTICAL`. По умолчанию выбирается горизонтальная ориентация
min, *max*	Минимальное и максимальное значения для индикатора выполнения. По умолчанию принимаются значения **0** и **100** соответственно

- `int getMinimum()`
- `int getMaximum()`
- `void setMinimum(int value)`
- `void setMaximum(int value)`

Получают или устанавливают минимальное или максимальное значение.

- `int getValue()`
- `void setValue(int value)`

Получают или устанавливают текущее значение.

- `String getString()`
- `void setString(String s)`

Получают или устанавливают символьную строку, отображаемую на полосе индикатора выполнения. Если параметр *s* принимает пустое значение `null`, отображается символьная строка по умолчанию в формате **n%**.

- `boolean isStringPainted()`
- `void setStringPainted(boolean b)`

Получают или устанавливают значение свойства, определяющего порядок отображения символьной строки. Если это свойство принимает логическое значение `true`, символьная строка отображается на полосе индикатора выполнения. По умолчанию устанавливается логическое значение `false`, запрещающее отображение символьной строки.

- `boolean isIndeterminate()` **1.4**
- `void setIndeterminate(boolean b)` **1.4**

  Получают или устанавливают значение свойства определяющего неопределенное состояние индикатора выполнения. Если это свойство принимает логическое значение `true`, прямоугольник движется вперед и назад по полосе, указывая на неопределенное время ожидания. По умолчанию устанавливается логическое значение `false`.

javax.swing.ProgressMonitor 1.2

- `ProgressMonitor(Component` *parent*`, Object` *message*`, String` *note*`, int` *min*`, int` *max*`)`

  Создает диалоговое окно монитора текущего состояния.

Параметры:	*parent*	Родительский компонент, над которым отображается данное диалоговое окно
	message	Объект сообщения, отображаемый в диалоговом окне
	note	Необязательная строка примечания, выводимая ниже сообщения. Если данный параметр принимает пустое значение `null`, место для строки примечания не выделяется и последующие вызовы метода `setNote()` ничего не дают
	min, *max*	Минимальное и максимальное значения для индикатора выполнения

- `void setNote(String` *note*`)`

  Изменяет текст примечания.

- `void setProgress(int` *value*`)`

  Устанавливает указанное значение для индикатора выполнения.

- `void close()`

  Закрывает диалоговое окно.

- `boolean isCanceled()`

  Возвращает логическое значение `true`, если пользователь закрыл диалоговое окно, щелкнув на кнопке Cancel.

javax.swing.ProgressMonitorInputStream 1.2

- `ProgressMonitorInputStream(Component` *parent*`, Object` *message*`, InputStream` *in*`)`

  Создает фильтр для потока ввода и связывает с ним диалоговое окно монитора текущего состояния.

  Параметры:

parent	Родительский компонент, над которым отображается данное диалоговое окно
message	Объект сообщения, отображаемый в диалоговом окне
in	Контролируемый поток ввода

Организаторы и декораторы компонентов

Обсуждение расширенных средств библиотеки Swing завершается рассмотрением компонентов, помогающих организовать другие компоненты ГПИ. К их числу относятся *разделяемые панели*, разбивающие область окна на несколько частей с регулируемыми границами, *панели с вкладками*, на которых пользователь может переходить от одной вкладки к другой, а также *настольные панели*, на которых можно размещать несколько *внутренних фреймов*. И в заключение будут рассмотрены *слои* — декораторы, накладываемые на другие компоненты ГПИ для их специального оформления.

Разделяемые панели

Разделяемая панель позволяет разбивать компонент на части с регулируемыми границами. На рис. 6.44 приведен фрейм с двумя вложенными разделяемыми панелями. Компоненты на внешней разделяемой панели располагаются по вертикали: текстовая область — снизу, а другая, внутренняя разделяемая панель — сверху. В свою очередь, компоненты на внутренней разделяемой панели располагаются по горизонтали: список — слева, а метка с изображением — справа.

При создании разделяемой панели следует указать ориентацию разделительной полосы по горизонтали (константа `JSplitPane.HORIZONTAL_SPLIT`) или по вертикали (константа `JSplitPane.VERTICAL_SPLIT`), а также два разделяемых компонента, как показано ниже.

```
JSplitPane innerPane =
    new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, planetList, planetImage);
```

Вот, собственно, и все, что требуется сделать. По желанию разделительную полосу можно дополнить пиктограммами оперативного развертывания и свертывания панелей, показанными на рис. 6.44. Для визуального стиля Metal они имеют вид треугольников. Достаточно щелкнуть на такой пиктограмме, чтобы разделительная полоса переместилась в указанном направлении до полного развертывания или свертывания разделяемой панели. Для реализации такой дополнительной возможности нужно вызвать следующий метод:

```
innerPane.setOneTouchExpandable(true);
```

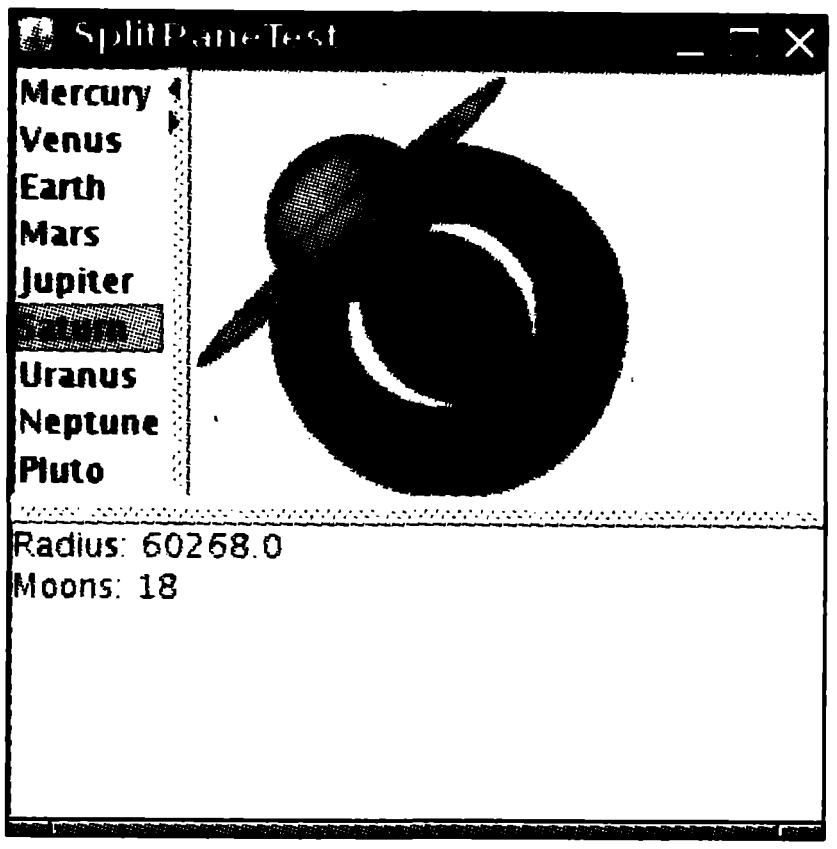


Рис. 6.44. Фрейм с двумя вложенными разделяемыми панелями

Кроме того, можно воспользоваться функцией непрерывной компоновки для перерисовки содержимого компонентов разделяемой панели по мере перемещения разделительной полосы. Результаты выглядят очень эффектно, но выполнение кода замедляется. Поэтому данную функцию лучше отключить, как показано ниже.

```
innerPane.setContinuousLayout(true);
```

В рассматриваемом здесь примере программы нижняя разделительная полоса отображается в исходном режиме без непрерывной компоновки. При ее перетаскивании перемещается только черный контур границы раздела. А компоненты разделяемой панели перерисовываются только после отпускания кнопки мыши. В этой простой программе, исходный код которой приведен в листинге 6.29, создается список, заполняемый названиями планет. Если выбрать какую-нибудь планету из списка, то справа от него показывается изображение выбранной планеты, а снизу — ее краткое описание. Запустив эту программу на выполнение, попробуйте переместить разделительные полосы, чтобы проверить действие функций оперативного развертывания и постоянной перерисовки разделяемых панелей.

Листинг 6.29. Исходный код из файла `splitPane/SplitPaneFrame.java`

```
1  package splitPane;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.event.*;
6
7  /**
8   * Этот фрейм состоит из двух вложенных разделяемых панелей для
9   * показа изображения и краткого описания выбранной планеты
10 */
11 class SplitPaneFrame extends JFrame
12 {
13   private static final int DEFAULT_WIDTH = 300;
14   private static final int DEFAULT_HEIGHT = 300;
15
16   private Planet[] planets = { new Planet("Mercury", 2440, 0),
```

```
17       new Planet("Venus", 6052, 0),
18       new Planet("Earth", 6378, 1), new Planet("Mars", 3397, 2),
19       new Planet("Jupiter", 71492, 16), new Planet("Saturn", 60268, 18),
20       new Planet("Uranus", 25559, 17), new Planet("Neptune", 24766, 8),
21       new Planet("Pluto", 1137, 1), };
22
23    public SplitPaneFrame()
24    {
25       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
26
27       // установить компоненты для названий, изображений
28       // и кратких описаний планет
29
30       final JList<Planet> planetList = new JList<>(planets);
31       final JLabel planetImage = new JLabel();
32       final JTextArea planetDescription = new JTextArea();
33
34       planetList.addListSelectionListener(new ListSelectionListener()
35          {
36             public void valueChanged(ListSelectionEvent event)
37             {
38                Planet value = (Planet) planetList.getSelectedValue();
39
40                // обновить изображение и краткое описание планеты
41
42                planetImage.setIcon(value.getImage());
43                planetDescription.setText(value.getDescription());
44             }
45          });
46
47       // установить разделяемые панели
48       JSplitPane innerPane = new JSplitPane(
49          JSplitPane.HORIZONTAL_SPLIT, planetList, planetImage);
50
51       innerPane.setContinuousLayout(true);
52       innerPane.setOneTouchExpandable(true);
53
54       JSplitPane outerPane = new JSplitPane(
55          JSplitPane.VERTICAL_SPLIT, innerPane, planetDescription);
57
58       add(outerPane, BorderLayout.CENTER);
59    }
60 }
```

javax.swing.JSplitPane 1.2

- JSplitPane()
- JSplitPane(int *direction*)
- JSplitPane(int *direction*, boolean *continuousLayout*)
- JSplitPane(int direction, Component first, Component *second*)
- JSplitPane(int *direction*, boolean *continuousLayout*, Component *first*, Component *second*)

  Конструируют новую разделяемую панель.

Параметры:	*direction*	Принимает значение одной из следующих констант: HORIZONTAL_SPLIT или VERTICAL_SPLIT
	continuousLayout	Принимает логическое значение true, если содержимое компонентов непрерывно обновляется при перемещении разделительной полосы
	first, second	Добавляемые компоненты

- `boolean isOneTouchExpandable()`
- `void setOneTouchExpandable(boolean b)`

  Получают или устанавливают значение свойства, определяющего порядок оперативного развертывания разделяемой панели. Если это свойство принимает логическое значение `true`, то на разделительной полосе появляются пиктограммы для развертывания или свертывания одного или другого компонента панели одним щелчком.

- `boolean isContinuousLayout()`
- `void setContinuousLayout(boolean b)`

  Получают или устанавливают значение свойства, определяющего режим непрерывной компоновки содержимого разделяемой панели. Если это свойство принимает логическое значение `true`, то при перемещении разделительной полосы компоненты панели непрерывно обновляются.

- `void setLeftComponent(Component c)`
- `void setTopComponent(Component c)`

  Эти методы дают один и тот же результат, устанавливая в качестве параметра *c* первый компонент на разделяемой панели.

- `void setRightComponent(Component c)`
- `void setBottomComponent(Component c)`

  Эти методы дают один и тот же результат, устанавливая в качестве параметра *c* второй компонент на разделяемой панели.

Панели с вкладками

Панели с вкладками можно использовать для создания очень удобного пользовательского интерфейса. В этом случае сложное диалоговое окно разбивается на несколько частей, каждая из которых содержит группу связанных по смыслу вариантов выбора. Вкладки можно также применять для просмотра документов или рисунков, как показано на рис. 6.45 и демонстрируется в рассматриваемом здесь примере программы.

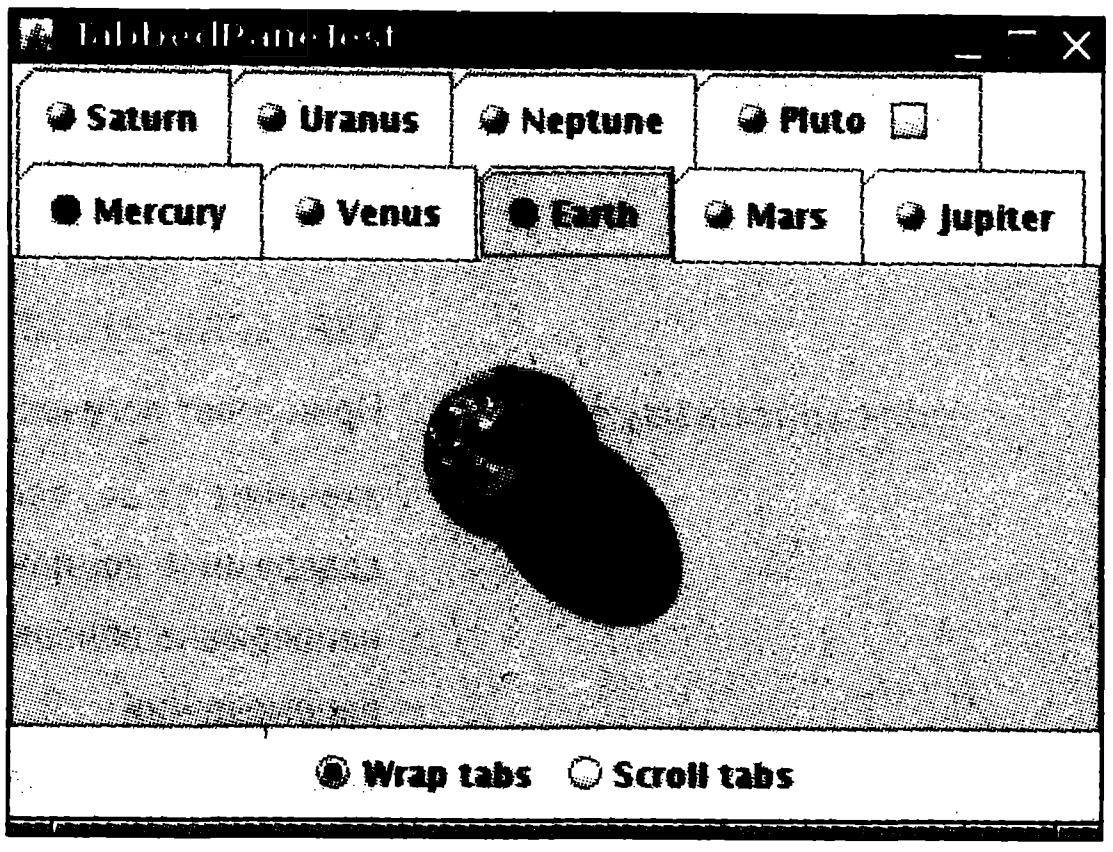


Рис. 6.45. Панель с вкладками

Для того чтобы создать панель с вкладками, прежде всего необходимо сконструировать объект типа `JTabbedPane`, а затем ввести в него вкладки следующим образом:

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.addTab(title, icon, component);
```

Последний параметр метода `addTab()` относится к типу `Component`. Для ввода нескольких компонентов на одной и той же вкладке нужно сначала упаковать их в контейнер (например, `JPanel`). Пиктограмму можно и не указывать в качестве параметра `icon`. Для этого предусмотрен другой вариант вызова метода `addTab()`:

```
tabbedPane.addTab(title, component);
```

Очередную вкладку можно ввести в нужном месте уже существующего ряда вкладок. Для этого необходимо указать ее индекс следующим образом:

```
tabbedPane.insertTab(title, icon, component, tooltip, index);
```

Для удаления ненужной вкладки из ряда вкладок следует вызвать приведенный ниже метод `removeTabAt()`.

```
tabPane.removeTabAt(index);
```

Вновь введенная вкладка не отображается автоматически. Для этого ее придется выбрать с помощью метода `setSelectedIndex()`. Например, для отображения вкладки, введенной в конце ряда вкладок, нужно сделать следующий вызов:

```
tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);
```

Для отображения большого количества вкладок может потребоваться немало места. Ради экономии места в рабочем окне с версии JDK 1.4 предусмотрена возможность для автоматической прокрутки вкладок, отображаемых в одном ряду. На правом краю от вкладок в этом же ряду присутствуют кнопки со стрелками, обозначающими направление прокрутки (рис. 6.46).

Для указания режима отображения вкладок полностью или частично с возможностью прокрутки вызываются методы

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
```

или

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
```

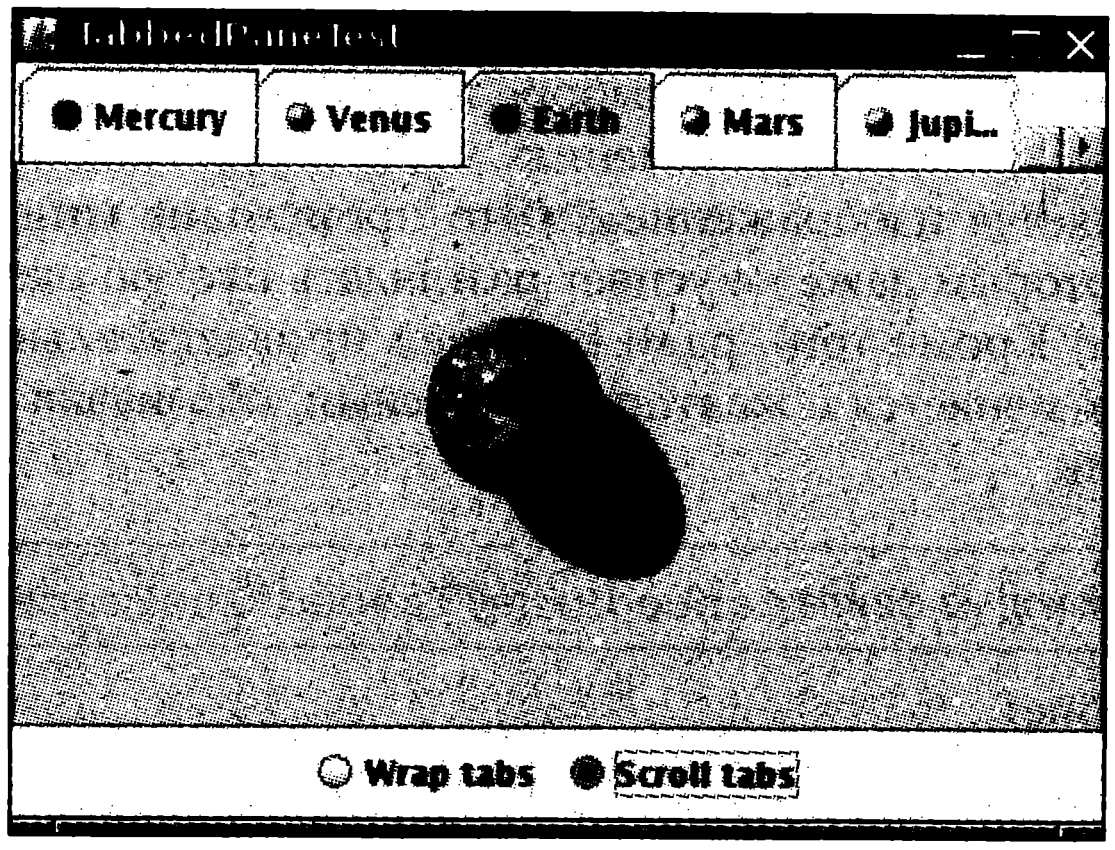


Рис. 6.46. Панель с прокручиваемыми вкладками

Заголовки вкладок могут иметь мнемонику подобно пунктам меню. Например, в приведенном ниже фрагменте кода прописная буква **M** подчеркивается, и пользователи могут выбрать вкладку, нажав комбинацию клавиш <Alt+M>.

```
int marsIndex = tabbedPane.indexOfTab("Mars");
tabbedPane.setMnemonicAt(marsIndex, KeyEvent.VK_M);
```

В заголовки вкладок можно включать произвольные компоненты. Для этого нужно сначала ввести вкладку на панели, а затем вызвать следующий метод:

```
tabbedPane.setTabComponentAt(index, component);
```

В рассматриваемом здесь примере программы на вкладке Pluto вводится элемент закрытия, поскольку некоторые астрономы не считают Плутон настоящей планетой. Для этого в качестве компонента данной вкладки устанавливается панель со следующими двумя компонентами: меткой со значком и текстом вкладки, а также флажок с приемником действия, удаляющего вкладку.

В данной программе на примере панели с вкладками демонстрируется удобный прием. Иногда содержимое компонента требуется обновить перед его отображением. Поэтому в данной программе изображение планеты загружается только после того, как пользователь щелкнет кнопкой мыши на соответствующей вкладке. Для уведомления о таком событии нужно создать обработчик типа `ChangeListener` и связать его с панелью, как показано ниже. Следует, однако, иметь в виду, что такой обработчик создается для панели в целом, а не для отдельного ее компонента.

```
tabbedPane.addChangeListener(listener);
```

Как только пользователь выберет вкладку, вызывается метод `stateChanged()` из класса `ChangeListener`. В качестве источника событий можно выбрать панель с вкладками. Для этого следует вызвать метод `getSelectedIndex()` и определить панель, которую требуется отобразить:

```
public void stateChanged(ChangeEvent event)
{
  int n = tabbedPane.getSelectedIndex();
  loadTab(n);
}
```

В примере программы из листинга 6.30 для всех компонентов вкладки сначала задается пустое значение null. При выборе новой вкладки выполняется проверка компонента на равенство пустому значению null, и при положительном результате компонент заменяется изображением. (Это происходит мгновенно и незаметно для пользователя, который даже не успеет разглядеть пустую панель после того, как щелкнет на вкладке.) Кроме того, ради большей привлекательности цвет шариков в заголовках вкладок меняется с желтого на красный, обозначая те панели, которые уже просматривались.

Листинг 6.30. Исходный код из файла tabbedPane/TabbedPaneFrame.java

```
1   package tabbedPane;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6   import javax.swing.event.*;
7
8   /**
9    * Этот фрейм содержит панель с вкладками и кнопки-переключатели
10   * режимов отображения вкладок полностью или частично с прокруткой
11  */
12  public class TabbedPaneFrame extends JFrame
13  {
14    private static final int DEFAULT_WIDTH = 400;
15    private static final int DEFAULT_HEIGHT = 300;
16
17    private JTabbedPane tabbedPane;
18    public TabbedPaneFrame()
19    {
20      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
21
22      tabbedPane = new JTabbedPane();
23      // установить пустое значение null во всех компонентах и
24      // отложить их загрузку вплоть до показа вкладки в первый раз
25      ImageIcon icon =
26         new ImageIcon(getClass().getResource("yellow-ball.gif"));
27
28      tabbedPane.addTab("Mercury", icon, null);
29      tabbedPane.addTab("Venus", icon, null);
30      tabbedPane.addTab("Earth", icon, null);
31      tabbedPane.addTab("Mars", icon, null);
32      tabbedPane.addTab("Jupiter", icon, null);
33      tabbedPane.addTab("Saturn", icon, null);
34      tabbedPane.addTab("Uranus", icon, null);
35      tabbedPane.addTab("Neptune", icon, null);
36      tabbedPane.addTab("Pluto", null, null);
37
38      final int plutoIndex = tabbedPane.indexOfTab("Pluto");
39      JPanel plutoPanel = new JPanel();
40      plutoPanel.add(new JLabel("Pluto", icon, SwingConstants.LEADING));
41      JToggleButton plutoCheckBox = new JCheckBox();
42      plutoCheckBox.addActionListener(new ActionListener()
43        {
44          public void actionPerformed(ActionEvent e)
45          {
```

```
46            tabbedPane.remove(plutoIndex);
47          }
48        });
49      plutoPanel.add(plutoCheckBox);
50      tabbedPane.setTabComponentAt(plutoIndex, plutoPanel);
51
52      add(tabbedPane, "Center");
53
54      tabbedPane.addChangeListener(new ChangeListener()
55        {
56          public void stateChanged(ChangeEvent event)
57          {
58            // проверить, имеется ли все еще пустой компонент на вкладке
59            if (tabbedPane.getSelectedComponent() == null)
60            {
61              // установить пиктограмму для компонента
62              int n = tabbedPane.getSelectedIndex();
63              loadTab(n);
64            }
65          }
66        });
67      loadTab(0);
68
69      JPanel buttonPanel = new JPanel();
70      ButtonGroup buttonGroup = new ButtonGroup();
71      JRadioButton wrapButton = new JRadioButton("Wrap tabs");
72      wrapButton.addActionListener(new ActionListener()
73        {
74          public void actionPerformed(ActionEvent event)
75          {
76            tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
77          }
78        });
79      buttonPanel.add(wrapButton);
80      buttonGroup.add(wrapButton);
81      wrapButton.setSelected(true);
82      JRadioButton scrollButton = new JRadioButton("Scroll tabs");
83      scrollButton.addActionListener(new ActionListener()
84        {
85          public void actionPerformed(ActionEvent event)
86          {
87            tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
88          }
89        });
90      buttonPanel.add(scrollButton);
91      buttonGroup.add(scrollButton);
92      add(buttonPanel, BorderLayout.SOUTH);
93    }
94
95    /**
96     * Загружает вкладку по указанному индексу
97     * @param n Индекс загружаемой вкладки
98    */
99    private void loadTab(int n)
100   {
101     String title = tabbedPane.getTitleAt(n);
102     ImageIcon planetIcon =
103       new ImageIcon(getClass().getResource(title + ".gif"));
```

```
104      tabbedPane.setComponentAt(n, new JLabel(planetIcon));
105
106      // обозначить эту вкладку как уже просматривавшуюся
107      // просто ради большей привлекательности
108      tabbedPane.setIconAt(
109         n, new ImageIcon(getClass().getResource("red-ball.gif")));
110   }
111 }
```

javax.swing.JTabbedPane 1.2

- `JTabbedPane()`
- `JTabbedPane(int placement)`

  Конструируют панель с вкладками.

  Параметры: *placement* — Принимает значение одной из следующих констант: `SwingConstants.TOP`, `SwingConstants.LEFT`, `SwingConstants.RIGHT` или `SwingConstants.BOTTOM`

- `void addTab(String title, Component c)`
- `void addTab(String title, Icon icon, Component c)`
- `void addTab(String title, Icon icon, Component c, String tooltip)`

  Добавляют вкладку в конце ряда на панели с вкладками.

- `void insertTab(String title, Icon icon, Component c, String tooltip, int index)`

  Вставляет вкладку на место по указанному индексу на панели с вкладками.

- `void removeTabAt(int index)`

  Удаляет вкладку по указанному индексу из панели с вкладками.

- `void setSelectedIndex(int index)`

  Выбирает вкладку по указанному индексу.

- `int getSelectedIndex()`

  Возвращает индекс выбранной вкладки.

- `Component getSelectedComponent()`

  Возвращает компонент из выбранной вкладки.

- `String getTitleAt(int index)`
- `void setTitleAt(int index, String title)`
- `Icon getIconAt(int index)`
- `void setIconAt(int index, Icon icon)`

- `Component getComponentAt(int index)`
- `void setComponentAt(int index, Component c)`

  Получают или устанавливают заголовок, пиктограмму или компонент вкладки по указанному индексу.
- `int indexOfTab(String title)`
- `int indexOfTab(Icon icon)`
- `int indexOfComponent(Component c)`

  Возвращают индекс вкладки с указанным заголовком, пиктограммой или компонентом.
- `int getTabCount()`

  Возвращает общее количество вкладок на данной панели с вкладками.
- `int getTabLayoutPolicy()`
- `void setTabLayoutPolicy(int policy)` **1.4**

  Получают или устанавливают порядок отображения вкладок. Они могут отображаться полностью (константа `JTabbedPane.WRAP_TAB_LAYOUT`) или частично с прокруткой (константа `JTabbedPane.SCROLL_TAB_LAYOUT`).
- `int getMnemonicAt(int index)` **1.4**
- `void setMnemonicAt(int index, int mnemonic)`

  Получают или устанавливают мнемонический знак на вкладке по указанному индексу. Знак определяется значением константы `VK_X` из класса `KeyEvent`. Значение **-1** обозначает, что мнемонический знак отсутствует.
- `Component getTabComponentAt(int index)` **6**
- `void setTabComponentAt(int index, Component c)` **6**

  Получают или устанавливают компонент, отображающий заголовок вкладки по указанному индексу. Если этот компонент пустой, то воспроизводятся значок и заголовок вкладки, а иначе — только указанный компонент на вкладке.
- `int indexOfTabComponent(Component c)` **6**

  Возвращает индекс вкладки для ее указанного компонента.
- `void addChangeListener(ChangeListener listener)`

  Вводит приемник событий, который уведомляется о выборе пользователем другой вкладки.

Настольные панели и внутренние фреймы

Во многих приложениях предоставляемые данные отображаются в нескольких окнах, находящихся в одном крупном фрейме. При свертывании фрейма все эти окна скрываются. В Windows такой пользовательский интерфейс называется *многодокументным* (MDI). На рис. 6.47 показано типичное приложение, в котором используется такой интерфейс.

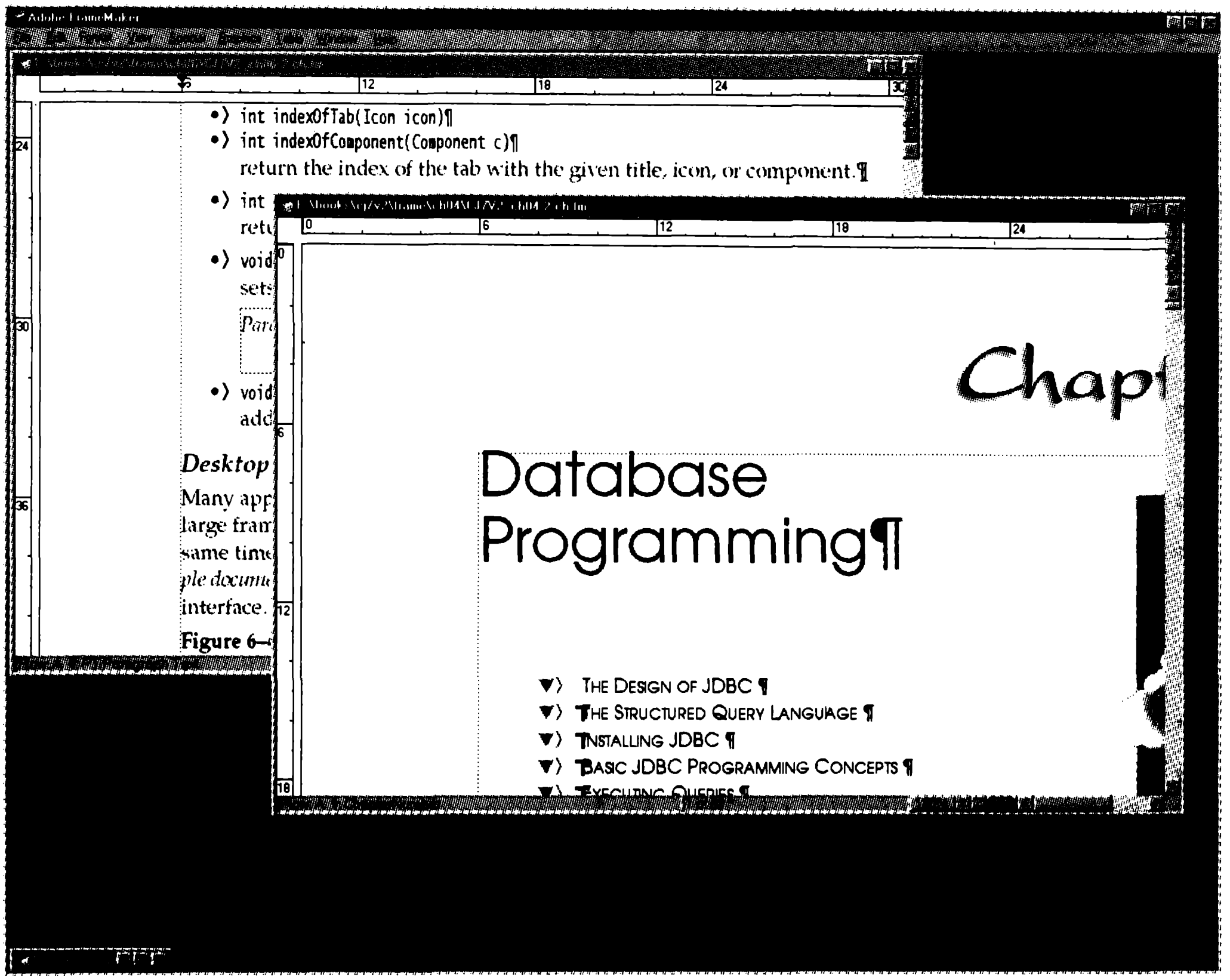


Рис. 6.47. Приложение с многодокументным интерфейсом

Ранее такой стиль представления ГПИ был широко распространен, но в последнее время он стал менее популярным. Так, во многих браузерах для отображения веб-страниц используется несколько фреймов. Какой же способ лучше? Оба способа имеют свои преимущества и недостатки. Многодокументный интерфейс позволяет справиться с беспорядком, который возникает при наличии большого количества открытых окон. Но для перехода между многими окнами верхнего уровня приходится использовать кнопки и оперативные клавиши оконной системы базовой платформы.

В межплатформенной среде Java не предусмотрено полагаться на богатые возможности оконной системы базовой платформы. Поэтому целесообразнее строить ГПИ на основе фреймов. На рис. 6.48 показано приложение Java с тремя внутренними фреймами. Два из них содержат пиктограммы для развертывания, свертывания и закрытия фрейма. А третий фрейм показан в свернутом виде.

При использовании визуального стиля Metal внутренние фреймы имеют отдельные области "захвата", которые можно использовать для перемещения фреймов. А для изменения размеров окон достаточно перетащить в нужном направлении уголки фрейма.

Для того чтобы добиться подобных функциональных возможностей фреймов, выполните следующие действия.

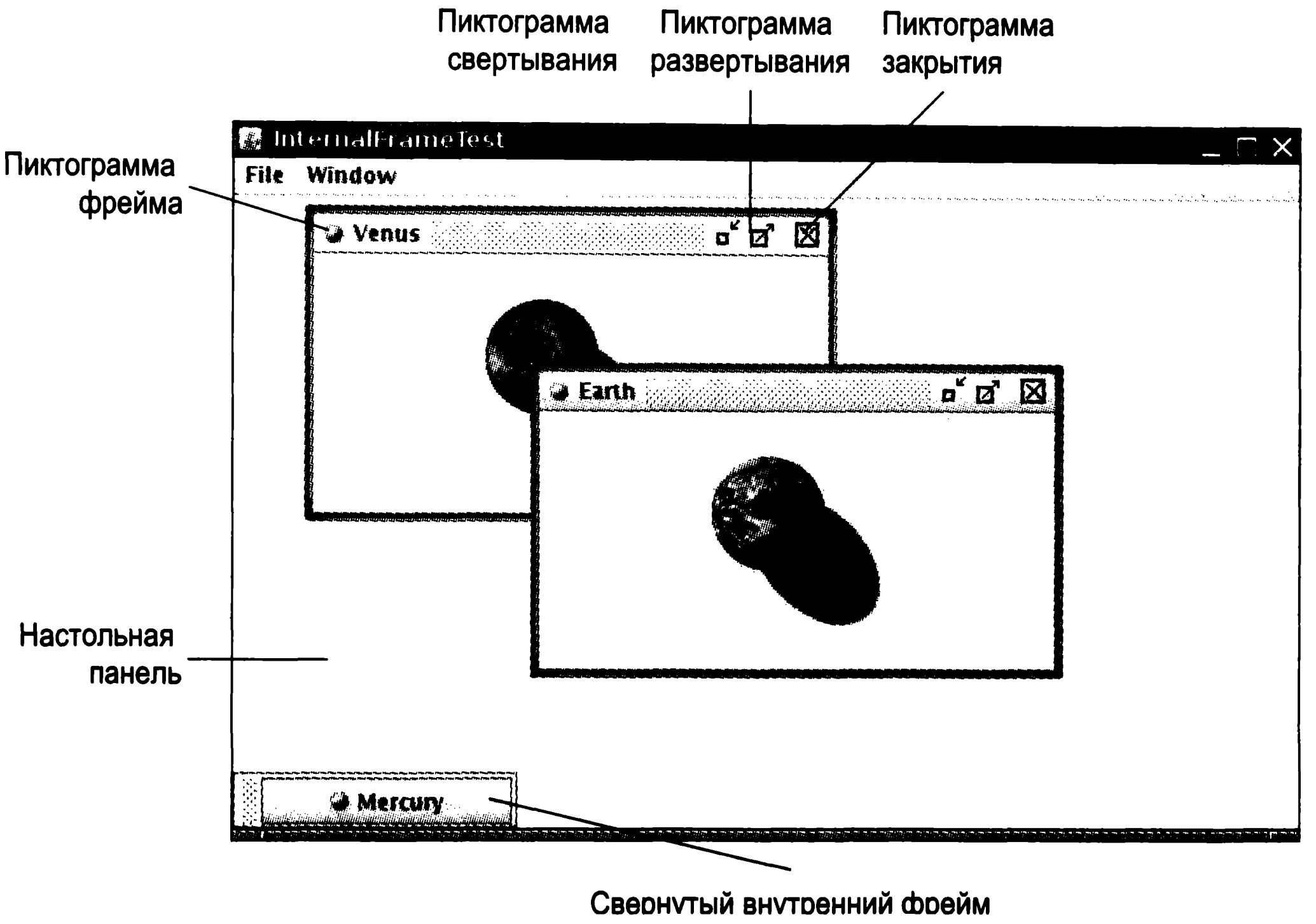


Рис. 6.48. Приложение с тремя внутренними фреймами

1. Создайте обычное окно фрейма типа `JFrame` для своего приложения.
2. Введите во фрейм типа `JFrame` настольную панель типа `JDesktopPane` следующим образом:

```
desktop = new JDesktopPane();
add(desktop, BorderLayout.CENTER);
```

3. Создайте окна внутренних фреймов типа `JInternalFrame`, указав по желанию пиктограммы для изменения размеров фрейма и его закрытия, как показано ниже. Как правило, указываются все пиктограммы.

```
JInternalFrame iframe = new JInternalFrame(title,
    true, // изменение размеров
    true, // закрытие
    true, // развертывание
    true); // свертывание
```

4. Введите во фрейм нужные компоненты:

```
iframe.add(c, BorderLayout.CENTER);
```

5. Установите для фрейма пиктограмму, которая будет отображаться в его левом верхнем углу:

```
iframe.setFrameIcon(icon);
```

НА ЗАМЕТКУ! В текущей версии визуального стиля Metal пиктограмма не отображается, когда фрейм находится в свернутом виде.

6. Установите размеры внутреннего фрейма. В исходном состоянии внутренний фрейм, как и стандартный, имеет размеры 0×0. Кроме того, с помощью метода `reshape()` нужно указать разное исходное положение для всех фреймов, чтобы они не закрывали друг друга. Этот же метод дает возможность установить размеры фреймов следующим образом:

```
iframe.reshape(nextFrameX, nextFrameY, width, height);
```

7. Подобно экземплярам обычного фрейма типа `JFrame`, сделайте каждый внутренний фрейм видимым, как показано ниже.

```
iframe.setVisible(true);
```

НА ЗАМЕТКУ! В ранних версиях Swing внутренние фреймы автоматически становились видимыми, и поэтому вызывать метод `setVisible()` не требовалось.

8. Введите фрейм на настольной панели типа `JDesktopPane` следующим образом:

```
desktop.add(iframe);
```

9. Новый фрейм, вероятнее всего, придется сделать *выбранным фреймом*, который, в отличие от других внутренних фреймов, имеет фокус ввода. В соответствии с визуальным стилем Metal строка заголовка выбранного фрейма выделяется синим цветом, тогда как строка заголовка других невыбранных фреймов — серым цветом. Для выбора фрейма воспользуйтесь методом `setSelected()`. Но свойство, определяющее выбранное состояние фрейма, может оказаться недоступным для установки. Это означает, что фрейм, выбранный в настоящий момент, может отказаться отдавать фокус ввода тому фрейму, который требуется выбрать. В таком случае метод `setSelected()` генерирует исключение типа `PropertyVetoException`, которое следует перехватить и обработать:

```
try
{
  iframe.setSelected(true);
}
catch (PropertyVetoException ex)
{
  // попытка выбрать фрейм не удалась
}
```

10. Расположите каждый последующий фрейм со смещением, чтобы фреймы не закрывали полностью друг друга. Расстояние между фреймами удобно выбрать равным высоте заголовка фрейма, которая получается следующим образом:

```
int frameDistance = iframe.getHeight() - iframe.getContentPane().
getHeight()
```

11. Воспользуйтесь полученным расстоянием, чтобы определить положение следующего внутреннего фрейма:

```
nextFrameX += frameDistance;
nextFrameY += frameDistance;
if (nextFrameX + width > desktop.getWidth())
  nextFrameX = 0;
if (nextFrameY + height > desktop.getHeight())
  nextFrameY = 0;
```

Каскадное и мозаичное расположение фреймов

В Windows предусмотрено несколько команд для *каскадного* расположения фреймов с перекрытием (рис. 6.49) и *мозаичного* их расположения без перекрытия (рис. 6.50). Но в классах `JDesktopPane` и `JInternalFrame` из библиотеки Swing не предусмотрено никаких средств для поддержки подобных операций. Поэтому в примере программы из листинга 6.31 демонстрируется, как самостоятельно организовать упорядочение фреймов, реализовав операции их каскадного и мозаичного расположения.

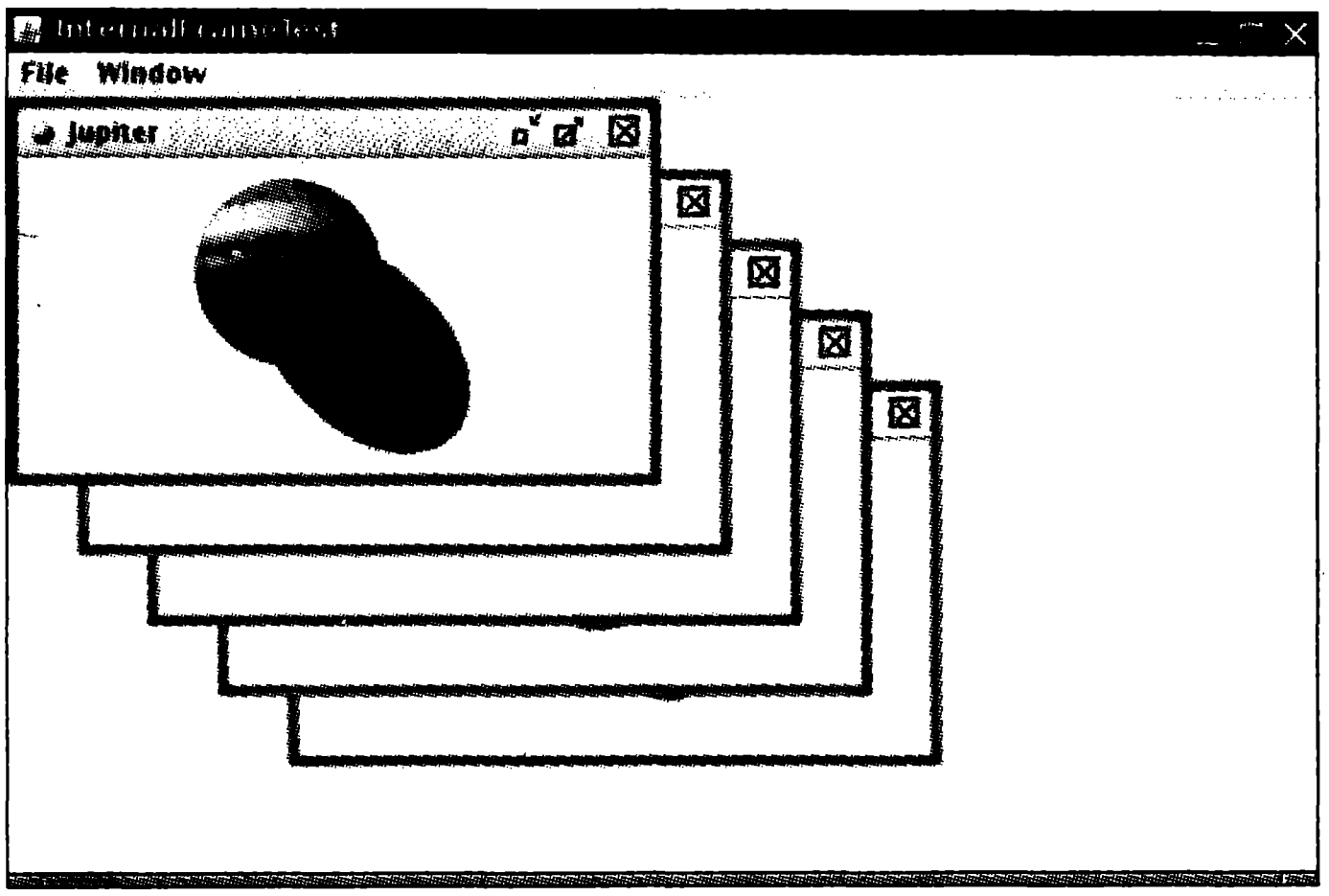


Рис. 6.49. Каскадное расположение внутренних фреймов

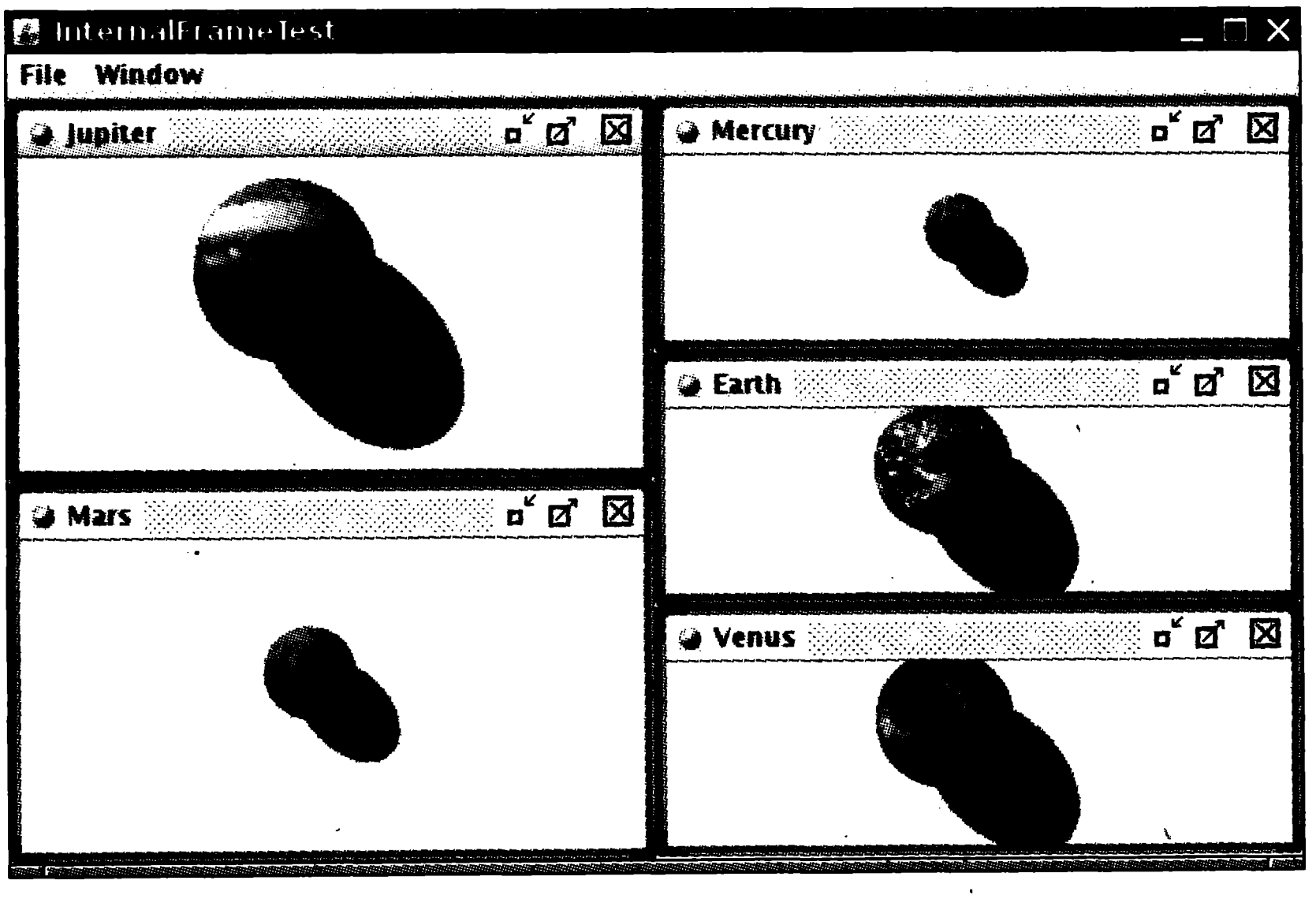


Рис. 6.50. Мозаичное расположение внутренних фреймов

Для каскадного расположения нужно задать одинаковые размеры всех фреймов, сместив положение каждого последующего фрейма на одинаковую величину

относительно предыдущего. Для того чтобы получить массив всех внутренних фреймов, вызывается метод `getAllFrames()` из класса `JDesktopPane`:

```
JInternalFrame[] frames = desktop.getAllFrames();
```

Но при этом нужно учитывать текущее состояние внутреннего фрейма. Каждый внутренний фрейм может находиться в одном из следующих трех состояний:

- свернутое;
- нормальное, допускающее изменение размеров;
- развернутое.

Для выявления свернутых фреймов, которые следует пропустить, служит метод `isIcon()`. Но если фрейм развернут, его нужно сначала установить в нормальное состояние, допускающее изменение размеров, с помощью метода `setMaximum(false)`. Но и это свойство может оказаться недоступным для установки, а следовательно, придется организовать перехват и обработку исключения типа `PropertyVetoException`. В приведенном ниже цикле все внутренние фреймы располагаются каскадом на настольной панели.

```
for (JInternalFrame frame : desktop.getAllFrames())
{
  if (!frame.isIcon())
  {
    try
    {
       // попытаться перевести фреймы из развернутого состояния
       // в нормальное, допускающее изменение размеров, хотя это
       // может быть и запрещено
       frame.setMaximum(false);
       frame.reshape(x, y, width, height);
       x += frameDistance;
       y += frameDistance;
       // перенести в следующий ряд по достижении правого края
       // настольной панели
       if (x + width > desktop.getWidth()) x = 0;
       if (y + height > desktop.getHeight()) y = 0;
    }
    catch (PropertyVetoException ex)
    {}
  }
}
```

Организовать мозаичное расположение фреймов труднее, особенно если количество фреймов таково, что они не вписываются полностью в квадратную сетку. С этой целью сначала подсчитывается количество несвернутых фреймов, а затем определяется количество рядов в первом столбце квадратной сетки, как показано ниже.

```
int rows = (int) Math.sqrt(frameCount);
```

Далее определяется количество заполненных столбцов:

```
int cols = frameCount / rows;
```

А последний незаполненный столбец определяется так:

```
int extra = frameCount % rows
```

Количество рядов в этом столбце равно `rows + 1`. В приведенном ниже цикле все внутренние фреймы располагаются мозаикой на настольной панели.

```
int width = desktop.getWidth() / cols;
int height = desktop.getHeight() / rows;
int r = 0;
int c = 0;
for (JInternalFrame frame : desktop.getAllFrames())
{
  if (!frame.isIcon())
  {
    try
    {
      frame.setMaximum(false);
      frame.reshape(c * width, r * height, width, height);
      r++;
      if (r == rows)
      {
        r = 0;
        c++;
        if (c == cols - extra)
        {
           // начать ввод дополнительного ряда
           rows++;
           height = desktop.getHeight() / rows;
        }
      }
    }
    catch (PropertyVetoException ex)
    {}
  }
}
```

В рассматриваемом здесь примере программы демонстрируется еще одна распространенная операция: перемещение выбора от текущего фрейма к следующему несвернутому фрейму. С этой целью обходятся все фреймы и вызывается метод `isSelected()` до тех пор, пока не будет найден текущий выбранный фрейм. Затем выявляется следующий несвернутый фрейм и делается попытка выбрать его с помощью метода

```
rames[next].setSelected(true);
```

Как и прежде, этот метод может генерировать исключение типа `PropertyVetoException`. Возврат к исходному фрейму означает, что других доступных для выбора фреймов не существует. Ниже приведен цикл для обхода всех внутренних фреймов, выявления среди них текущего выбранного фрейма и выбора следующего.

```
JInternalFrame[] frames = desktop.getAllFrames();
for (int i = 0; i < frames.length; i++)
{
  if (frames[i].isSelected())
  {
    // найти следующий несвернутый фрейм, который можно выбрать
    int next = (i + 1) % frames.length;
    while (next != i)
    {
      if (!frames[next].isIcon())
```

```
        {
          try
          {
            // все остальные фреймы свернуты или недоступны для выбора
            frames[next].setSelected(true);
            frames[next].toFront();
            frames[i].toBack();
            return;
          }
          catch (PropertyVetoException ex)
          {}
        }
        next = (next + 1) % frames.length;
      }
    }
}
```

Наложение запрета на установку свойств

Итак, рассмотрев исключения, возникающие в связи с запретом на установку свойств, выясним, каким образом подобный запрет накладывается во фреймах. Для контроля над установкой свойств в классе `JInternalFrame` используется общий механизм *JavaBeans*. Этот механизм подробно описывается в главе 8, а здесь показывается, каким образом во фреймах формируются запросы на изменения свойств.

Запреты обычно не накладываются на свертывание или передачу фокуса ввода, но нередко во фреймах проверяется, разрешено ли их *закрывать*. Для закрытия фрейма служит метод `setClosed()` из класса `JInternalFrame`. Но поскольку этот метод допускает наложение запрета, то в нем вызываются все зарегистрированные *приемники запрещаемых изменений* перед тем, как перейти непосредственно к изменениям. Таким образом, каждому приемнику предоставляется возможность сгенерировать исключение типа `PropertyVetoException` и завершить метод `setClosed()` до внесения в нем каких-нибудь изменений в установки свойств.

В рассматриваемом здесь примере программы появляется диалоговое окно, приведенное на рис. 6.51. В нем пользователю предлагается подтвердить намерение закрыть фрейм. Если пользователь не дает такого подтверждения, то фрейм остается открытым. Для организации такого уведомления пользователя выполните следующие действия.

1. Создайте объект приемника для каждого фрейма, как показано ниже. Этот объект должен быть экземпляром некоторого класса, реализующего интерфейс `VetoableChangeListener`. Такой приемник лучше всего ввести сразу же после создания фрейма. В рассматриваемом здесь примере программы для создания внутренних фреймов применяется класс фрейма, хотя для этой цели вполне подойдет и анонимный внутренний класс.

   ```
   iframe.addVetoableChangeListener(listener);
   ```

2. Реализуйте метод `vetoableChange()`, объявленный единственным в интерфейсе `VetoableChangeListener`. В качестве параметра ему передается объект типа `PropertyChangeEvent`. Чтобы получить имя изменяемого свойства, воспользуйтесь методом `getName()`. Так, если для наложения запрета вызывается метод `setClosed(true)`, изменяемое свойство будет называться `closed`. Как показано в главе 8, имя свойства получается путем удаления префикса **`set`** из имени метода и замены первой прописной буквы на строчную.

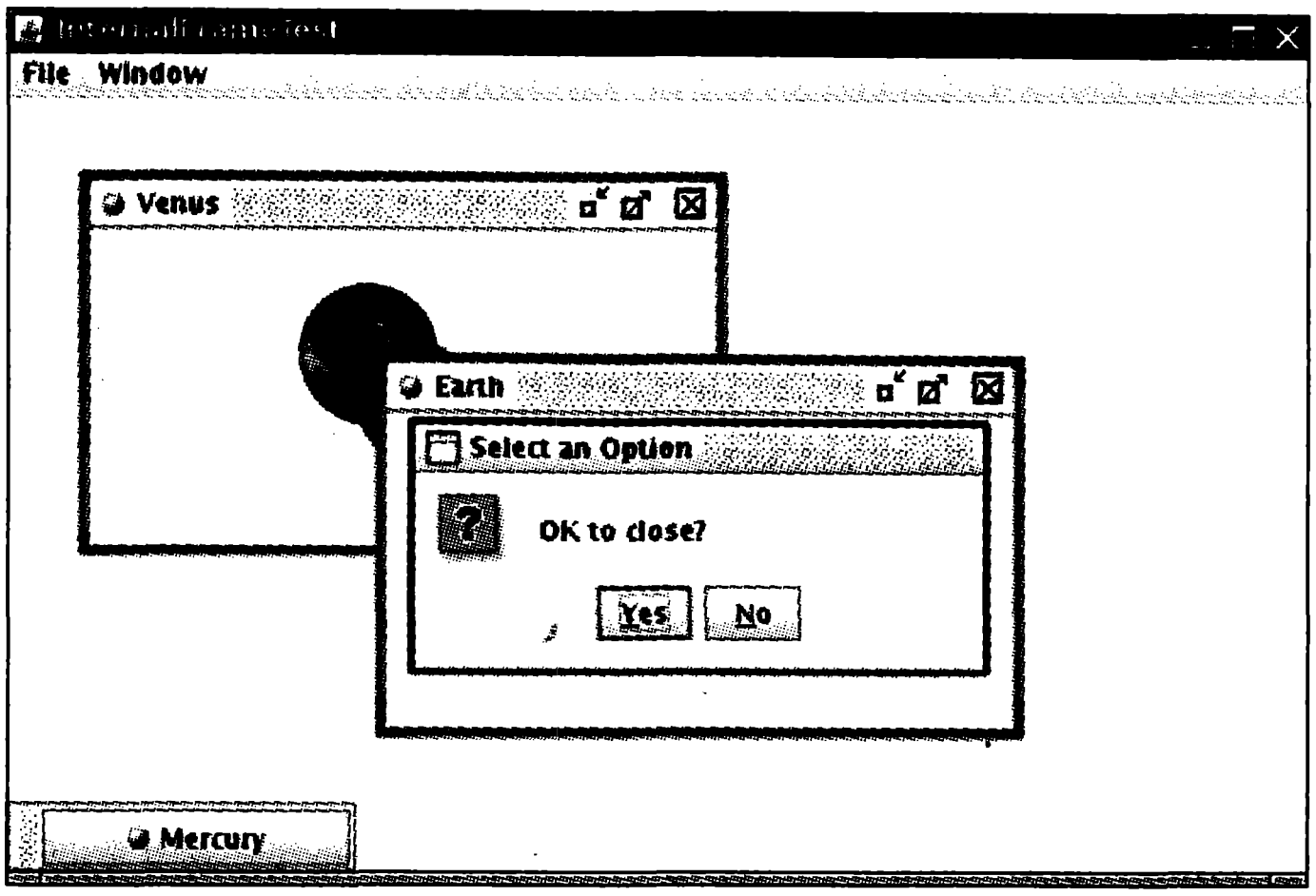


Рис. 6.51. Пользователь может наложить запрет на закрытие окна

3. Вызовите метод `getNewValue()`, чтобы получить новое предлагаемое значение свойства:

```
String name = event.getPropertyName();
Object value = event.getNewValue();
if (name.equals("closed") && value.equals(true))
{
   запросить у пользователя подтверждение
}
```

4. Создайте метод, генерирующий исключение типа `PropertyVetoException`, чтобы наложить запрет на изменение свойства. Если же накладывать запрет на изменение свойства не требуется, завершите метод обычным способом, как показано в приведенном ниже фрагменте кода.

```
class DesktopFrame extends JFrame
    implements VetoableChangeListener
{
   . . .
   public void vetoableChange(PropertyChangeEvent event)
       throws PropertyVetoException
   {
     . . .
     if (запрещено)
       throw new PropertyVetoException(причина_запрета, event);
     // если разрешено, произвести возврат обычным способом
   }
}
```

Диалоговые окна во внутренних фреймах

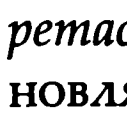
Для создания диалоговых окон во внутренних фреймах не рекомендуется использовать класс `JDialog`, поскольку такие окна обладают некоторыми недостатками.

- Они очень ресурсоемки, потому что создают новый фрейм в системе окон.
- Системе управления окнами неизвестно, как расположить диалоговое окно по отношению к породившему его внутреннему фрейму.

Вместо этого для создания простых диалоговых окон рекомендуется использовать методы типа `showInternalXxxDialog()` из класса `JOptionPane`. Они действуют аналогично методам типа `showXxxDialog()`, за исключением того, что диалоговое окно располагается непосредственно над внутренним фреймом. Для создания более сложных диалоговых окон следует использовать класс `JInternalFrame`, но в нем не предусмотрены инструменты для поддержки режимных (или модальных) окон.

В рассматриваемом здесь примере программы используется внутреннее диалоговое окно, в котором пользователю предлагается подтвердить закрытие фрейма, как выделено ниже полужирным.

```
int result = JOptionPane.show Internal ConfirmDialog(
   iframe, "OK to close?", "Select an Option", JOptionPane.YES_NO_OPTION);
```



НА ЗАМЕТКУ! Если требуется только *уведомить* о закрытии фрейма, то вместо механизма запрета лучше установить приемник типа `InternalFrameListener`, который действует аналогично приемнику типа `WindowsListener`. При закрытии внутреннего фрейма вместо метода `windowClosing()` вызывается метод `internalFrameClosing()`. Аналогичным образом применяются все шесть методов, уведомляющих о действиях с внутренним фреймом (открыт/закрыт, свернут/развернут, активен/неактивен).

Перетаскивание контуров

Разработчики часто жалуются на низкую производительность при использовании внутренних фреймов. Например, медленнее всего выполняется операция перетаскивания фрейма со сложным содержимым. Дело в том, что настольный диспетчер постоянно перерисовывает фрейм, что существенно замедляет работу. Подобная ситуация наблюдается в графических оболочках Windows или X Window с неудачно написанным видеодрайвером. В некоторых системах перетаскивание выполняется быстро, поскольку в видеоадаптерах данная операция поддерживается на аппаратном уровне. С этой целью изображение во фрейме отображается на разные места экрана в процессе перетаскивания.

Для повышения производительности можно воспользоваться так называемым *перетаскиванием контуров*. В этом случае при перетаскивании фрейма постоянно обновляются только его контуры. А внутренняя часть фрейма перерисовывается только тогда, когда фрейм достигнет своего конечного положения. Для того чтобы включить режим перетаскивания контуров, следует вызвать приведенный ниже метод. Такая установка равнозначна переключению режима постоянной перерисовки в классе `JSplitPane`.

```
desktop.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```

НА ЗАМЕТКУ! В ранних версиях библиотеки Swing для включения режима перетаскивания контуров приходилось вызывать следующий метод:

```
desktop.putClientProperty("JDesktopPane.dragMode", "outline");
```

В рассматриваемом здесь примере программы для включения/отключения режима перетаскивания контуров пользователю предоставляется пункт меню Window⇨Drag Outline (Окно⇨Перетаскивать контур) с устанавливаемым флажком.

НА ЗАМЕТКУ! Внутренними фреймами настольной панели управляет настольный диспетчер, реализуемый в классе `DesktopManager`. Для обычного программирования этот класс не требуется. Его следует применять лишь для реализации различных вариантов поведения настольной панели. С этой целью устанавливается новый настольный диспетчер, но рассмотрение подобных вопросов выходит за рамки данной книги.

В рассматриваемом здесь примере программы из листинга 6.31 демонстрируется настольная панель с внутренними фреймами, предназначенными для отображения HTML-страниц. При выборе пункта меню File⇨Open (Файл⇨Открыть) на экране появляется диалоговое окно для выбора локального HTML-файла. Если щелкнуть на выбранном HTML-файле, он отобразится в новом внутреннем фрейме. Попробуйте изменить расположение фреймов на экране с помощью пунктов меню Window⇨Cascade (Окно⇨Расположить каскадом) и Window⇨Tile (Окно⇨Расположить мозаикой).

Листинг 6.31. Исходный код из файла `internalFrame/DesktopFrame.java`

```
1   package internalFrame;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.beans.*;
6   import javax.swing.*;
7
8   /**
9    * Этот фрейм настольной панели содержит панели редакторов,
10   * на которых отображаются HTML-документы
11  */
12  public class DesktopFrame extends JFrame
13  {
14    private static final int DEFAULT_WIDTH = 600;
15    private static final int DEFAULT_HEIGHT = 400;
16    private static final String[] planets =
17       { "Mercury", "Venus", "Earth", "Mars", "Jupiter",
18         "Saturn", "Uranus", "Neptune", "Pluto", };
19
20    private JDesktopPane desktop;
21    private int nextFrameX;
22    private int nextFrameY;
23    private int frameDistance;
24    private int counter;
25
26    public DesktopFrame()
27    {
28      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
29
30      desktop = new JDesktopPane();
31      add(desktop, BorderLayout.CENTER);
32
33      // установить меню
34
35      JMenuBar menuBar = new JMenuBar();
36      setJMenuBar(menuBar);
37      JMenu fileMenu = new JMenu("File");
38      menuBar.add(fileMenu);
```

```
39        JMenuItem openItem = new JMenuItem("New");
40        openItem.addActionListener(new ActionListener()
41          {
42            public void actionPerformed(ActionEvent event)
43            {
44              createInternalFrame(new JLabel(new ImageIcon(getClass()
45                 .getResource(planets[counter] + ".gif"))),
46              planets[counter]);
47              counter = (counter + 1) % planets.length;
48            }
49          });
50        fileMenu.add(openItem);
51        JMenuItem exitItem = new JMenuItem("Exit");
52        exitItem.addActionListener(new ActionListener()
53          {
54            public void actionPerformed(ActionEvent event)
55            {
56              System.exit(0);
57            }
58          });
59        fileMenu.add(exitItem);
60        JMenu windowMenu = new JMenu("Window");
61        menuBar.add(windowMenu);
62        JMenuItem nextItem = new JMenuItem("Next");
63        nextItem.addActionListener(new ActionListener()
64          {
65            public void actionPerformed(ActionEvent event)
66            {
67              selectNextWindow();
68            }
69          });
70        windowMenu.add(nextItem);
71        JMenuItem cascadeItem = new JMenuItem("Cascade");
72        cascadeItem.addActionListener(new ActionListener()
73          {
74            public void actionPerformed(ActionEvent event)
75            {
76              cascadeWindows();
77            }
78          });
79        windowMenu.add(cascadeItem);
80        JMenuItem tileItem = new JMenuItem("Tile");
81        tileItem.addActionListener(new ActionListener()
82          {
83            public void actionPerformed(ActionEvent event)
84            {
85              tileWindows();
86            }
87          });
88        windowMenu.add(tileItem);
89        final JCheckBoxMenuItem dragOutlineItem =
90           new JCheckBoxMenuItem("Drag Outline");
91        dragOutlineItem.addActionListener(new ActionListener()
92          {
93            public void actionPerformed(ActionEvent event)
94            {
95              desktop.setDragMode(dragOutlineItem.isSelected()
96                 ? JDesktopPane.OUTLINE_DRAG_MODE
```

```
97                  : JDesktopPane.LIVE_DRAG_MODE);
98              }
99          });
100       windowMenu.add(dragOutlineItem);
101    }
102
103    /**
104     * Создает внутренний фрейм на настольной панели
105     * @param c Компонент, отображаемый во внутреннем фрейме
106     * @param t Заголовок внутреннего фрейма
107    */
108    public void createInternalFrame(Component c, String t)
109    {
110       final JInternalFrame iframe = new JInternalFrame(t,
111          true, // изменение размеров
112          true, // закрытие
113          true, // разворачивание
114          true); // сворачивание
115
116       iframe.add(c, BorderLayout.CENTER);
117       desktop.add(iframe);
118
119       iframe.setFrameIcon(new ImageIcon(getClass()
120          .getResource("document.gif")));
121       // ввести приемник, чтобы подтвердить закрытие фрейма
122       iframe.addVetoableChangeListener(new VetoableChangeListener()
123          {
124             public void vetoableChange(PropertyChangeEvent event)
125                throws PropertyVetoException
126             {
127                String name = event.getPropertyName();
128                Object value = event.getNewValue();
129
130                // проверить только попытки закрыть фрейм
132                if (name.equals("closed") && value.equals(true))
133                {
134                   // запросить у пользователя разрешение закрыть фрейм
135                   int result = JOptionPane.showInternalConfirmDialog(
136                      iframe, "OK to close?", "Select an Option",
137                      JOptionPane.YES_NO_OPTION);
138
139                   // если пользователь не дает разрешение,
140                   // наложить запрет на закрытие фрейма
141                   if (result != JOptionPane.YES_OPTION)
142                      throw new PropertyVetoException(
143                      "User canceled close", event);
144                }
145             }
146          });
147       // расположить фрейм
148       int width = desktop.getWidth() / 2;
149       int height = desktop.getHeight() / 2;
150       iframe.reshape(nextFrameX, nextFrameY, width, height);
151
152       iframe.show();
153
154       // выбрать фрейм, на что может быть наложен запрет
155       try
```

```
156     {
157       iframe.setSelected(true);
158     }
159     catch (PropertyVetoException ex)
160     {
161     }
162
163     frameDistance = iframe.getHeight() -
164        iframe.getContentPane().getHeight();
165
166     // вычислить расположение следующего фрейма
167
168     nextFrameX += frameDistance;
169     nextFrameY += frameDistance;
170     if (nextFrameX + width > desktop.getWidth()) nextFrameX = 0;
171     if (nextFrameY + height > desktop.getHeight()) nextFrameY = 0;
172   }
173
174   /**
175    * Располагает каскадом внутренние фреймы,
176    * не свернутые на настольной панели
177   */
178   public void cascadeWindows()
179   {
180     int x = 0;
181     int y = 0;
182     int width = desktop.getWidth() / 2;
183     int height = desktop.getHeight() / 2;
184
185     for (JInternalFrame frame : desktop.getAllFrames())
186     {
187       if (!frame.isIcon())
188       {
189         try
190         {
190           // попытаться сделать развернутые фреймы допускающими
191           // изменение размеров, что может быть запрещено
192           frame.setMaximum(false);
193           frame.reshape(x, y, width, height);
194
195           x += frameDistance;
196           y += frameDistance;
197           // перенести в следующий ряд по достижении правого
198           // края настольной панели
199           if (x + width > desktop.getWidth()) x = 0;
200           if (y + height > desktop.getHeight()) y = 0;
201         }
202         catch (PropertyVetoException ex)
203         {
204         }
205       }
206     }
207   }
208
209   /**
210    * Заголовки внутренних фреймов, не свернутых на настольной панели
211   */
212   public void tileWindows()
```

```
213   {
214     // подсчитать количество несвернутых фреймов
215     int frameCount = 0;
216     for (JInternalFrame frame : desktop.getAllFrames())
217     if (!frame.isIcon()) frameCount++;
218     if (frameCount == 0) return;
219
220     int rows = (int) Math.sqrt(frameCount);
221     int cols = frameCount / rows;
222     int extra = frameCount % rows;
223     // количество столбцов с дополнительными рядами
224
225     int width = desktop.getWidth() / cols;
226     int height = desktop.getHeight() / rows;
227     int r = 0;
228     int c = 0;
229     for (JInternalFrame frame : desktop.getAllFrames())
230     {
231       if (!frame.isIcon())
232       {
233         try
234         {
235           frame.setMaximum(false);
236           frame.reshape(c * width, r * height, width, height);
237           r++;
238           if (r == rows)
239           {
240             r = 0;
241             c++;
242             if (c == cols - extra)
243             {
244               // начать ввод дополнительного ряда
245               rows++;
246               height = desktop.getHeight() / rows;
247             }
248           }
249         }
250         catch (PropertyVetoException ex)
251         {
252         }
253       }
254     }
255   }
256
257   /**
258    * Перемещает вперед следующий несвернутый внутренний фрейм
259   */
260   public void selectNextWindow()
261   {
262     JInternalFrame[] frames = desktop.getAllFrames();
263     for (int i = 0; i < frames.length; i++)
264     {
265       if (frames[i].isSelected())
266       {
267         // найти следующий несвернутый фрейм, который можно выбрать
268         int next = (i + 1) % frames.length;
269         while (next != i)
270         {
```

```
271             if (!frames[next].isIcon())
272             {
273               try
274               {
275                 // все остальные фреймы свернуты, или их запрещено выбирать
276                 frames[next].setSelected(true);
277                 frames[next].toFront();
278                 frames[i].toBack();
279                 return;
280               }
281               catch (PropertyVetoException ex)
282               {
283               }
284             }
285             next = (next + 1) % frames.length;
286           }
287         }
288       }
289     }
290 }
```

javax.swing.JDesktopPane 1.2

- `JInternalFrame[] getAllFrames()`

  Получает все внутренние фреймы на данной панели.

- `void setDragMode(int mode)`

  Устанавливает на время перетаскивания фрейма отображение его содержимого или только контуров.

  Параметры: *mode* Принимает значение одной из следующих констант: `JDesktopPane.LIVE_DRAG_MODE` или `JDesktopPane.OUTLINE_DRAG_MODE`

javax.swing.JInternalFrame 1.2

- `JInternalFrame()`
- `JInternalFrame(String title)`
- `JInternalFrame(String title, boolean resizable)`
- `JInternalFrame(String title, boolean resizable, boolean closable)`
- `JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)`
- `JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)`

  Конструируют новый внутренний фрейм.

Параметры:	*title*	Символьная строка для вывода заголовка внутреннего фрейма
	resizable	Принимает логическое значение true, разрешающее изменение размеров внутреннего фрейма
	closable	Принимает логическое значение true, разрешающее закрытие внутреннего фрейма
	maximizable	Принимает логическое значение true, разрешающее разворачивание внутреннего фрейма
	iconifiable	Принимает логическое значение true, разрешающее сворачивание внутреннего фрейма

- `boolean isResizable()`
- `void setResizable(boolean b)`
- `boolean isClosable()`
- `void setClosable(boolean b)`
- `boolean isMaximizable()`
- `void setMaximizable(boolean b)`
- `boolean isIconifiable()`
- `void setIconifiable(boolean b)`

Получают или устанавливают свойства `resizable`, `closable`, `maximizable` или `iconifiable`. Если соответствующее свойство принимает логическое значение `true`, в строке заголовка фрейма отображается пиктограмма изменения размеров, закрытия, разворачивания или сворачивания внутреннего фрейма.

- `boolean isIcon()`
- `void setIcon(boolean b)`
- `boolean isMaximum()`
- `void setMaximum(boolean b)`
- `boolean isClosed()`
- `void setClosed(boolean b)`

Получают или устанавливают свойства `icon`, `maximum` или `closed`. Если соответствующее свойство принимает логическое значение `true`, то допускается разворачивание, сворачивание или закрытие внутреннего фрейма.

- `boolean isSelected()`
- `void setSelected(boolean b)`

Получают или устанавливают свойство `selected`. Если данное свойство принимает логическое значение `true`, то текущий внутренний фрейм становится выбранным на настольной панели.

- `void moveToFront()`
- `void moveToBack()`

  Перемещают внутренний фрейм на настольной панели вперед или назад.
- `void reshape(int x, int y, int width, int height)`

  Перемещает внутренний фрейм и изменяет его размеры.

  Параметры: *x*, *y* — Координаты верхнего левого внутреннего угла фрейма

  width, *height* — Ширина и высота фрейма
- `Container getContentPane()`
- `void setContentPane(Container c)`

  Получают или устанавливают панель содержимого для данного внутреннего фрейма.
- `JDesktopPane getDesktopPane()`

  Получает настольную панель для данного внутреннего фрейма.
- `Icon getFrameIcon()`
- `void setFrameIcon(Icon anIcon)`

  Получают или устанавливают пиктограмму, которая отображается в строке заголовка внутреннего фрейма.
- `boolean isVisible()`
- `void setVisible(boolean b)`

  Получают или устанавливают свойство, определяющее видимость внутреннего фрейма.
- `void show()`

  Делает видимым данный внутренний фрейм и перемещает его вперед.

javax.swing.JComponent 1.2

- `void addVetoableChangeListener(VetoableChangeListener listener)`

  Вводит приемник запрещаемых изменений, который уведомляется при попытке изменить свойство, если на это наложен запрет.

java.beans.VetoableChangeListener 1.1

- `void vetoableChange(PropertyChangeEvent event)`

  Вызывается, когда метод установки свойства, на изменение которого наложен запрет, соответственно уведомляет приемник запрещаемых изменений.

java.beans.PropertyChangeEvent 1.1

- `String getPropertyName()`

  Возвращает имя изменяемого свойства.
- `Object getNewValue()`

  Возвращает предлагаемое новое значение свойства.

java.beans.PropertyVetoException 1.1

- `PropertyVetoException(String reason, PropertyChangeEvent event)`
  Генерирует исключение для свойства, изменения в котором запрещены.

Параметры:	`reason`	Причина наложения запрета
	`event`	Событие, наступающее при попытке нарушить запрет

Слои

В версии Java SE 7 внедрено новое средство, позволяющее накладывать компоненты ГПИ слоями. В слое можно рисовать, принимая события из нижележащего слоя. Слои позволяют вводить дополнительные визуальные ориентиры в ГПИ. Например, с их помощью можно специально оформить введенные в настоящий момент данные, неверно введенные данные или недоступные компоненты.

Класс `JLayer` связывает компонент с объектом типа `LayerUI`, отвечающим за рисование и обработку событий. У конструктора класса `LayerUI` имеется один параметр типа, который должен соответствовать связанному компоненту. В качестве примера в приведенном ниже фрагменте кода показано, каким образом компонент типа `JPanel` дополняется новым слоем.

```
JPanel panel = new JPanel();
LayerUI<JPanel> layerUI = new PanelLayer();
JLayer layer = new JLayer(panel, layerUI);
frame.add(layer);
```

Следует, однако, иметь в виду, что в родительский компонент вводится слой, а не панель. Как показано ниже, класс `PanelLayer` является производным от класса `LayerUI`.

```
class PanelLayer extends LayerUI<Panel>
{
  public void paint(Graphics g, JComponent c)
  {
    . . .
  }
  . . .
}
```

В методе `paint()` допускается рисовать все, что угодно. Тем не менее для рисования компонента следует непременно сделать вызов `super.paint()`. Так, в приведенном ниже фрагменте кода весь компонент окрашивается прозрачным цветом.

```
public void paint(Graphics g, JComponent c)
{
  super.paint(g, c);
  Graphics2D g2 = (Graphics2D) g.create();
  g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, .3f));
  g2.setPaint(color));
  g2.fillRect(0, 0, c.getWidth(), c.getHeight());
  g2.dispose();
}
```

Для приема событий от связанного компонента или любых производных от него компонентов в классе `LayerUI` должна быть установлена маска событий в слое. Это нужно сделать в методе `installUI()` следующим образом:

```
class PanelLayer extends LayerUI<JPanel>
{
  public void installUI(JComponent c)
  {
    super.installUI(c);
    ((JLayer<?>) c).setLayerEventMask(AWTEvent.KEY_EVENT_MASK |
        AWTEvent.FOCUS_EVENT_MASK);
  }

  public void uninstallUI(JComponent c)
  {
    ((JLayer<?>) c).setLayerEventMask(0);
    super.uninstallUI(c);
  }
  . . .
}
```

Теперь события будут приниматься в методах типа `processXxxEvent()`. Так, в рассматриваемом здесь примере программы слой перерисовывается после каждого нажатия клавиш, как показано ниже.

```
public class PanelLayer extends LayerUI<JPanel>
{
  protected void processKeyEvent(KeyEvent e, JLayer<? extends JPanel> l)
  {
    l.repaint();
  }
}
```

В рассматриваемом здесь примере программы из листинга 6.32 имеются три поля для ввода значений основных (RGB) составляющих цвета фона. Как только пользователь изменит эти значения, цвет прозрачно отображается на панели. Кроме того, в данной программе перехватываются события, связанные с перемещением фокуса ввода, а текст в компоненте, обладающем фокусом ввода, выделяется полужирным шрифтом.

Листинг 6.32. Исходный код из файла `layer/ColorFrame.java`

```
1  package layer;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import javax.swing.plaf.*;
7
8  /**
9   * Фрейм с тремя текстовыми полями для установки цвета фона
10 */
11 public class ColorFrame extends JFrame
12 {
13   private JPanel panel;
14   private JTextField redField;
15   private JTextField greenField;
```

```
16   private JTextField blueField;
17
18   public ColorFrame()
19   {
20     panel = new JPanel();
21
22     panel.add(new JLabel("Red:"));
23     redField = new JTextField("255", 3);
24     panel.add(redField);
25
26     panel.add(new JLabel("Green:"));
27     greenField = new JTextField("255", 3);
28     panel.add(greenField);
29     panel.add(new JLabel("Blue:"));
30     blueField = new JTextField("255", 3);
31     panel.add(blueField);
32
33     LayerUI<JPanel> layerUI = new PanelLayer();
34     JLayer<JPanel> layer = new JLayer<JPanel>(panel, layerUI);
35
36     add(layer);
37     pack();
38   }
39
40   class PanelLayer extends LayerUI<JPanel>
41   {
42     public void installUI(JComponent c)
43     {
44       super.installUI(c);
45       ((JLayer<?>) c).setLayerEventMask(AWTEvent.KEY_EVENT_MASK |
46          AWTEvent.FOCUS_EVENT_MASK);
47     }
48
49     public void uninstallUI(JComponent c)
50     {
51        ((JLayer<?>) c).setLayerEventMask(0);
52        super.uninstallUI(c);
53     }
54
55     protected void processKeyEvent(
56        KeyEvent e, JLayer<? extends JPanel> l)
57     {
58       l.repaint();
59     }
60
61     protected void processFocusEvent(
62        FocusEvent e, JLayer<? extends JPanel> l)
63     {
64       if (e.getID() == FocusEvent.FOCUS_GAINED)
65       {
66         Component c = e.getComponent();
67         c.setFont(getFont().deriveFont(Font.BOLD));
68       }
69       if (e.getID() == FocusEvent.FOCUS_LOST)
70       {
71         Component c = e.getComponent();
72         c.setFont(getFont().deriveFont(Font.PLAIN));
73       }
74     }
```

```
76
77      public void paint(Graphics g, JComponent c)
78      {
79        super.paint(g, c);
80        Graphics2D g2 = (Graphics2D) g.create();
81        g2.setComposite(AlphaComposite.getInstance(
82           AlphaComposite.SRC_OVER, .3f));
83        int red = Integer.parseInt(redField.getText().trim());
84        int green = Integer.parseInt(greenField.getText().trim());
85        int blue = Integer.parseInt(blueField.getText().trim());
86        g2.setPaint(new Color(red, green, blue));
87        g2.fillRect(0, 0, c.getWidth(), c.getHeight());
88        g2.dispose();
89      }
90    }
91 }
```

javax.swing.JLayer<V extends Component> 7

- `JLayer(V view, LayerUI<V> ui)`

  Создает слой над заданным представлением *view*, делегируя рисование и обработку событий объекту *ui*.

- `void setLayerEventMask(long layerEventMask)`

  Позволяет отправить объекту типа `LayerUI` все совпадающие события, посланные связанному с ним компоненту или производным от него компонентам. Для составления маски событий в слое применяются в любом сочетании следующие константы из класса `LayerUI`:

  `COMPONENT_EVENT_MASK`
  `FOCUS_EVENT_MASK`
  `HIERARCHY_BOUNDS_EVENT_MASK`
  `HIERARCHY_EVENT_MASK`
  `INPUT_METHOD_EVENT_MASK`
  `KEY_EVENT_MASK`
  `MOUSE_EVENT_MASK`
  `MOUSE_MOTION_EVENT_MASK`
  `MOUSE_WHEEL_EVENT_MASK`

javax.swing.plaf.LayerUI<V extends Component> 7

- `void installUI(JComponent c)`
- `void uninstallUI(JComponent c)`

  Вызываются при установке или удалении объекта типа `LayerUI`, связанного с компонентом `c`. Этот метод следует переопределить для установки или удаления маски событий в слое.

- `void paint(Graphics g, JComponent c)`

  Вызывается при рисовании специально оформляемого компонента. Этот метод следует переопределить для вызова метода `super.paint()` и рисования специального оформления.

- `void processComponentEvent(ComponentEvent e, JLayer<? extends V> l)`
- `void processFocusEvent(FocusEvent e, JLayer<? extends V> l)`
- `void processHierarchyBoundsEvent(HierarchyEvent e, JLayer<? extends V> l)`
- `void processHierarchyEvent(HierarchyEvent e, JLayer<? extends V> l)`
- `void processInputMethodEvent(InputMethodEvent e, JLayer<? extends V> l)`
- `void processKeyEvent(KeyEvent e, JLayer<? extends V> l)`
- `void processMouseEvent(MouseEvent e, JLayer<? extends V> l)`
- `void processMouseMotionEvent(MouseEvent e, JLayer<? extends V> l)`
- `void processMouseWheelEvent(MouseWheelEvent e, JLayer<? extends V> l)`

  Вызываются при отправке указанного события данному объекту типа `LayerUI`.

В этой главе вы ознакомились с тем, как пользоваться усовершенствованными компонентами, предоставляемыми в библиотеке Swing. А в следующей главе мы обсудим вопросы, связанные с применением расширенных средств из библиотеки AWT, включая сложные операции рисования, манипулирование изображениями, печать и сопряжение с собственной оконной системой применяемой платформы.

ГЛАВА 7

Расширенные средства AWT

В этой главе...

- Конвейер визуализации
- Фигуры
- Участки
- Обводка
- Раскраска
- Преобразование координат
- Отсечение
- Прозрачность и композиция
- Указания по воспроизведению
- Чтение и запись изображений
- Манипулирование изображениями
- Вывод изображений на печать
- Буфер обмена
- Перетаскивание объектов
- Интегрирование с платформой

Для создания простых рисунков можно использовать методы из класса `Graphics`. Они эффективны для разработки простых аплетов и приложений, но их недостаточно для построения сложных фигур или полного контроля над внешним видом рисунков. В состав прикладного интерфейса Java 2D API входит библиотека классов для создания высококачественных рисунков. В этой главе рассматриваются

расширенные средства этой библиотеки, а также обсуждаются вопросы вывода получаемых изображений на печать и способы реализации функций печати в прикладных программах.

Кроме того, рассматриваются два способа обмена данными между программами: с помощью буфера обмена и механизма перетаскивания. Эти способы можно использовать для передачи данных между двумя приложениями на Java или прикладной программой на Java и собственной программой для конкретной платформы. И наконец, в главе представлены методики разработки прикладных программ на Java в стиле собственных приложений для конкретной платформы, включая предоставление начального экрана при запуске программы и пиктограммы на панели задач операционной системы.

Конвейер визуализации

В исходной версии JDK 1.0 применялся очень простой механизм для рисования фигур. Для этого достаточно было выбрать нужный цвет, режим рисования и вызвать подходящие методы из класса `Graphics`, например `drawRect()` или `fillOval()`. В прикладном интерфейсе Java 2D API поддерживается много больше возможностей для рисования двухмерной графики. Он, в частности, позволяет делать следующее.

- Легко формировать самые разные фигуры.
- Управлять *обводкой*, т.е. вычерчивать пером контуры фигур.
- Заполнять фигуры любым сплошным цветом, используя различные оттенки и узоры.
- Выполнять *преобразования* для перемещения, масштабирования, вращения и растягивания фигур.
- *Отсекать* фигуры таким образом, чтобы ограничить их произвольно выбираемым участком экрана.
- Выбирать *правила композиции*, чтобы описывать порядок сочетания пикселей новой и уже существующей фигур.
- Давать *указания по воспроизведению* для достижения компромисса между скоростью загрузки и качеством рисования.

Для того чтобы нарисовать фигуру, нужно выполнить следующие действия.

1. Получить объект класса `Graphics2D`. Это подкласс, производный от класса `Graphics`. Начиная с версии Java SE 1.2 такие методы, как `paint()` и `paintComponent()`, автоматически получают объект класса `Graphics2D`. Поэтому остается только выполнить приведение типов, как показано ниже.

```
public void paintComponent(Graphics g)
{
  Graphics2D g2 = (Graphics2D) g;
  . . .
}
```

2. Вызвать приведенный ниже метод `setRenderingHints()`, чтобы добавить указания по воспроизведению для достижения компромисса между скоростью и качеством рисования.

```
RenderingHints hints = . . .;
g2.setRenderingHints(hints);
```

3. Вызвать приведенный ниже метод `setStroke()`, чтобы задать *обводку* контура фигуры. Для обводки можно выбрать толщину, а также сплошную или пунктирную линию.

```
Stroke stroke = . . .;
g2.setStroke(stroke);
```

4. Вызвать приведенный ниже метод `setPaint()`, чтобы указать способ *раскраски*. Раскраска подразумевает закрашивание участков контура обводки или внутренней области фигуры. Она может состоять из одного сплошного цвета, нескольких меняющихся оттенков или мозаичных узоров.

```
Paint paint = . . .;
g2.setPaint(paint);
```

5. Вызвать метод `clip()`, чтобы задать *область отсечения* следующим образом:

```
Shape clip = . . .;
g2.clip(clip);
```

6. Вызвать приведенный ниже метод `setTransform()` для *преобразования* рисунка из пользовательского пространства в пространство конкретного устройства. Такое преобразование следует выполнять в тех случаях, когда фигуру проще создать в специальной системе координат, чем использовать для этой цели координаты, выражаемые в пикселях.

```
AffineTransform transform = . . .;
g2.transform(transform);
```

7. Вызвать приведенный ниже метод `setComposite()`, чтобы задать правило композиции, описывающее порядок объединения новых пикселей с уже существующими.

```
Composite composite = . . .;
g2.setComposite(composite);
```

8. Создать фигуру, как показано ниже. В прикладном интерфейсе Java 2D API предусмотрено немало объектов фигур и методов для их сочетания.

```
Shape shape = . . .;
```

9. Нарисовать или заполнить фигуру, как показано ниже. Под рисованием подразумевается очерчивание контуров фигуры, а под заполнением — закрашивание ее внутренней области.

```
g2.draw(shape);
g2.fill(shape);
```

Разумеется, на практике выполнять все эти действия обычно не требуется, и поэтому для многих параметров двухмерного графического контекста предусмотрены значения по умолчанию, изменять которые следует только в случае особой необходимости. В следующем разделе рассматриваются способы описания фигур, обводок, раскрасок, преобразований и правил композиции.

Различные методы типа `set()` просто устанавливают состояние двухмерного графического контекста и не относятся к конкретным рисункам. Аналогично фигура воспроизводится не во время создания объекта типа `Shape`, а при вызове методов `draw()` или `fill()`, когда новая фигура рассчитывается в *конвейере визуализации* (рис. 7.1).

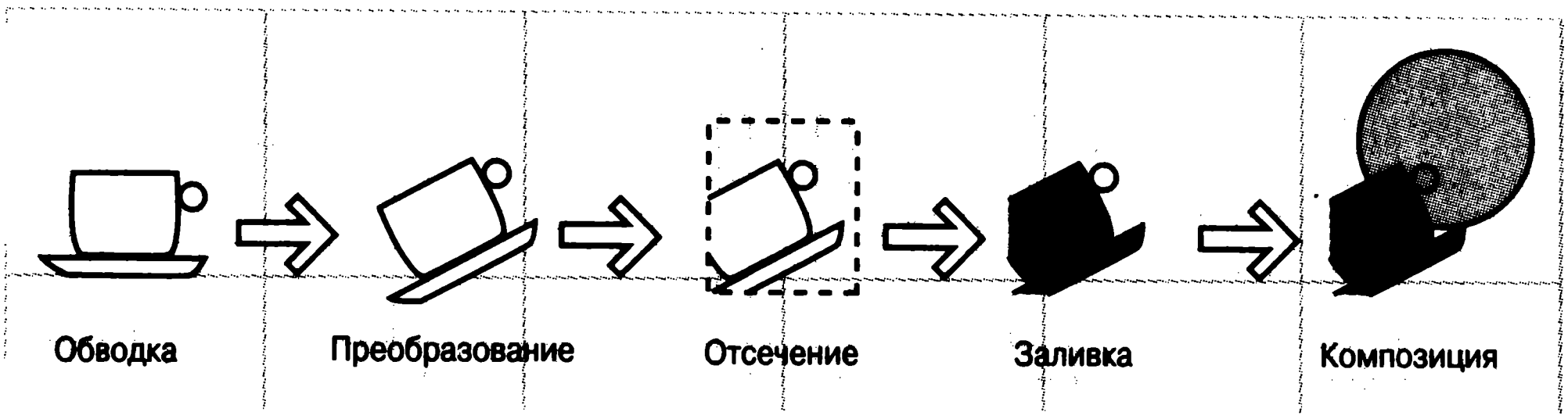


Рис. 7.1. Конвейер визуализации

Для воспроизведения фигуры в конвейере визуализации выполняются следующие действия.

1. Обводится контур фигуры.
2. Выполняется преобразование фигуры.
3. Происходит отсечение фигуры. Данный процесс прекращается, как только фигура перестает пересекать область отсечения.
4. Оставшаяся после отсечения часть фигуры заполняется.
5. Осуществляется композиция, т.е. составление пикселей фигуры и уже существующих пикселей в единый рисунок. (Как показано на рис. 7.1, уже существующие пиксели образуют круг, а фигура чашки накладывается на него.)

В следующем разделе сначала рассматриваются способы определения фигур, а затем установки двухмерного графического контекста.

java.awt.Graphics2D 1.2

- `void draw(Shape s)`
  Рисует контур указанной фигуры в соответствии с текущей раскраской.
- `void fill(Shape s)`
  Заполняет внутреннюю область фигуры в соответствии с текущей раскраской.

Фигуры

Ниже перечислены некоторые методы из класса `Graphics`, которые используются для рисования фигур.

```
drawLine()
drawRectangle()
drawRoundRect()
draw3Drect()
```

```
drawPolygon()
drawPolyline()
drawOval()
drawArc()
```

Для них существуют соответствующие методы заполнения фигур, которые были предусмотрены в классе `Graphics`, начиная с версии JDK 1.0. В прикладном интерфейсе Java 2D API используется совершенно другая, объектно-ориентированная модель, где вместо методов применяются перечисленные ниже классы. Эти классы реализуют интерфейс `Shape`.

```
Line2D
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
QuadCurve2D
CubicCurve2D
GeneralPath
```

И наконец, предусмотрен класс `Point2D`, описывающий точку с координатами *x* и *y*. Точки полезны для определения фигур, но сами по себе фигурами не являются. Для того чтобы нарисовать фигуру, нужно сначала создать экземпляр класса, реализующего интерфейс `Shape`, а затем вызвать метод `draw()` из класса `Graphics2D`.

Классы `Line2D`, `Rectangle2D`, `RoundRectangle2D`, `Ellipse2D` и `Arc2D` соответствуют методам `drawLine()`, `drawRectangle()`, `drawRoundRect()`, `drawOval()` и `drawArc()`. Для понятия "трехмерный прямоугольник" не предусмотрено соответствующего метода `draw3DRect()`. Но в прикладном интерфейсе Java 2D API поддерживаются два дополнительных класса для рисования кривых второго и третьего порядков, рассматриваемых далее. Для рисования многоугольников также не предусмотрено отдельного класса вроде `Polygon2D()`, а предлагается класс `GeneralPath`, описывающий контуры, состоящие из линий и кривых второго и третьего порядков. Класс `GeneralPath` можно использовать для описания многоугольника, как поясняется далее.

Перечисленные ниже классы наследуют общий класс `RectangularShape`. Как известно, эллипсы и дуги не являются прямоугольниками, но их можно вписать в прямоугольник (рис. 7.2).

```
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
```

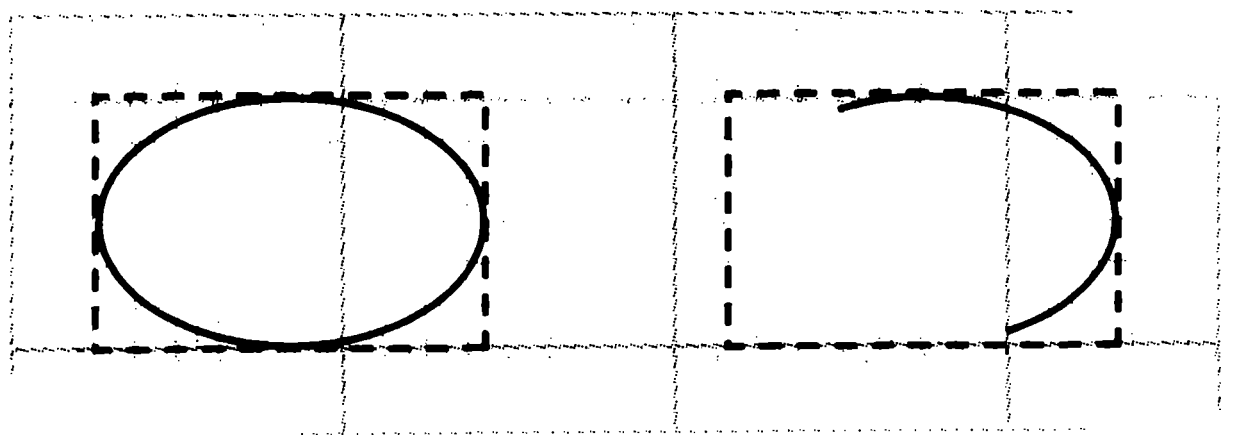


Рис. 7.2. Эллипс и дугу можно вписать в прямоугольник

У каждого из классов, название которого оканчивается на **2D**, имеются два подкласса для задания координат в виде числовых значений типа `float` и `double`. Так, в первом томе данной книги уже встречались выражения `Rectangle2D.Float` и `Rectangle2D.Double`. Такая же схема используется для обозначения других классов, например `Arc2D.Float` и `Arc2D.Double`.

Для внутреннего представления координат во всех классах, предназначенных для построения графики, используются числовые данные типа `float`, поскольку они требуют меньше места для хранения, чем числовые данные типа `double`. К тому же они поддерживают достаточную точность для геометрических расчетов. Но в языке Java обработка числовых данных типа `float` выполняется очень громоздкими и неуклюжими средствами. Поэтому большинство методов из классов для построения графики принимают параметры типа `double` и возвращают значения типа `double`. Выбирать тип числовых данных (`float` или `double`) и соответствующий конструктор класса приходится только при создании двухмерного объекта:

```
Rectangle2D floatRect = new Rectangle2D.Float(5F, 10F, 7.5F, 15F);
Rectangle2D doubleRect = new Rectangle2D.Double(5, 10, 7.5, 15);
```

Классы `Xxx2D.Float` и `Xxx2D.Double` являются производными от класса *Xxx*`2D`. После создания объекта нет никакой нужды запоминать подклассы, а можно просто сохранить созданный объект в переменной суперкласса, как показано в приведенном выше фрагменте кода.

Судя по названиям классов *Xxx*`2D.Float` и *Xxx*`2D.Double`, можно сделать вывод, что они являются внутренними для классов *Xxx*`2D`. Это простое синтаксическое правило позволяет избежать увеличения длины имен внешних классов. На рис. 7.3 схематически представлены отношения наследования между классами фигур без указания подклассов `Double` и `Float`. Устаревшие классы, унаследованные из прежних версий JDK, выделены серым цветом.

Применение классов фигур

О классах `Line2D`, `Rectangle2D` и `Ellipse2D` уже упоминалось в главе 7 первого тома данной книги, а в этом разделе речь пойдет о том, как пользоваться классами для построения остальных двухмерных фигур. В частности, для построения прямоугольника со скругленными углами в конструкторе класса `RoundRectangle2D` следует указать координаты верхнего левого угла, ширину, высоту и размеры области скругления углов по осям *x* и *y* (рис. 7.4). Например, в результате следующего вызова конструктора данного класса создается прямоугольник с радиусом скругления углов, равным **20**:

```
RoundRectangle2D r = new RoundRectangle2D.Double(150, 200, 100, 50, 20, 20);
```

Для построения дуги нужно указать ограничивающий прямоугольник, начальный угол, угол разворота дуги (рис. 7.5), а также один из видов замыкания дуги: `Arc2D.OPEN`, `Arc2D.PIE` или `Arc2D.CHORD`. Ниже приведен пример построения дуги, а на рис. 7.6 — виды замыкания дуги.

```
Arc2D a = new Arc2D(x, y, width, height, startAngle, arcAngle, closureType);
```

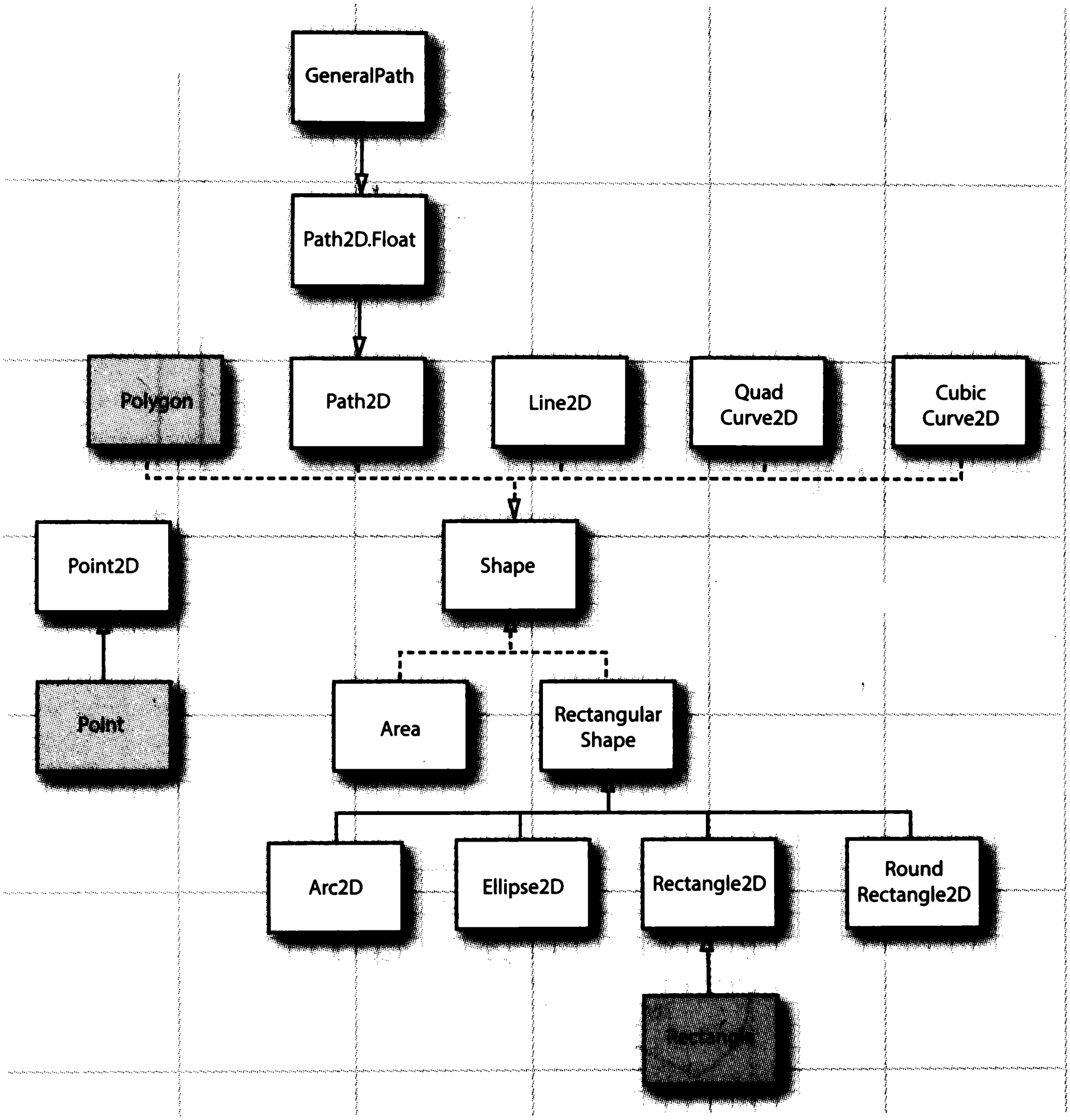


Рис. 7.3. Отношения наследования между классами фигур

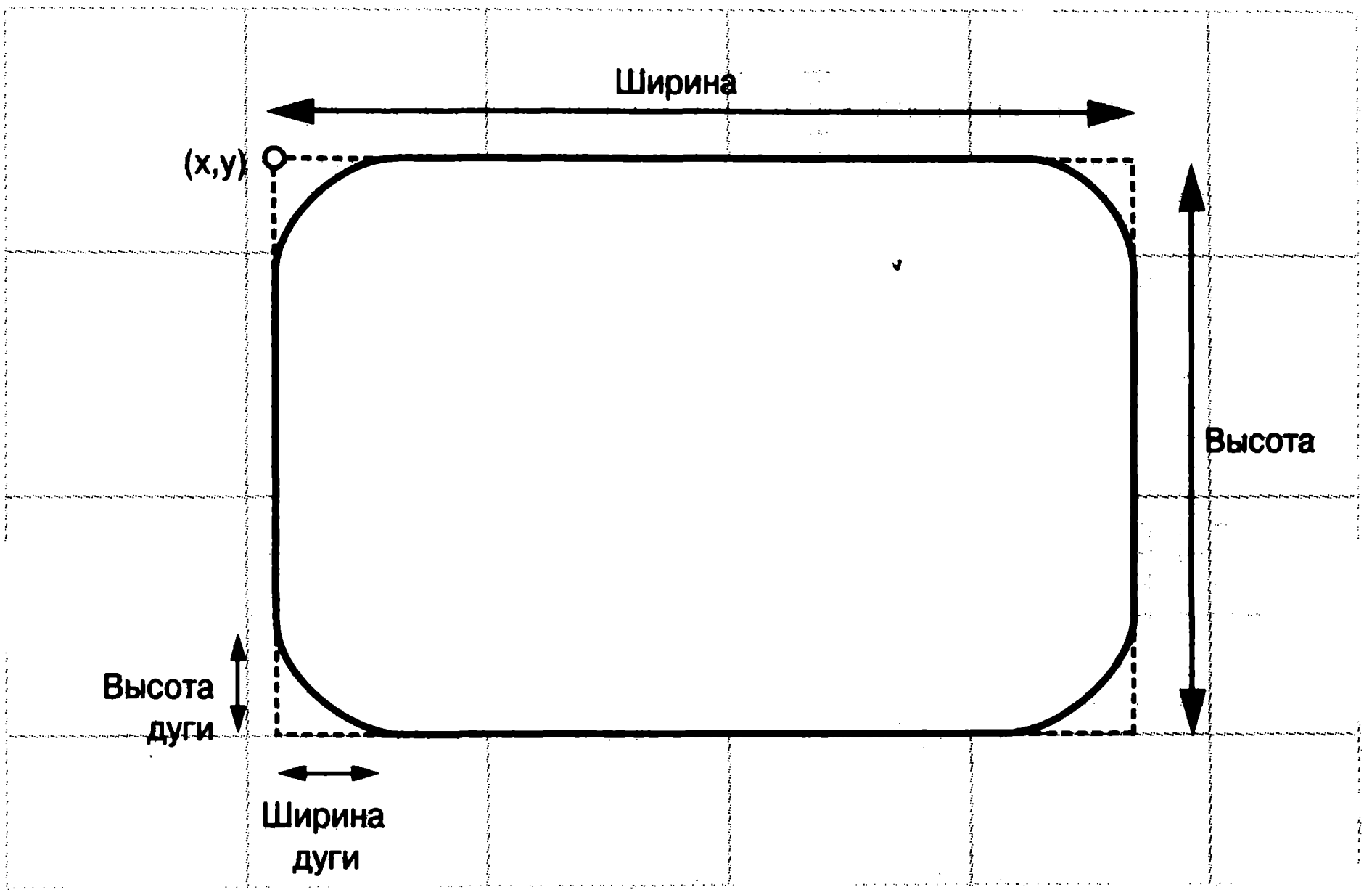


Рис. 7.4. Построение прямоугольника со скругленными углами средствами класса `RoundRectangle2D`

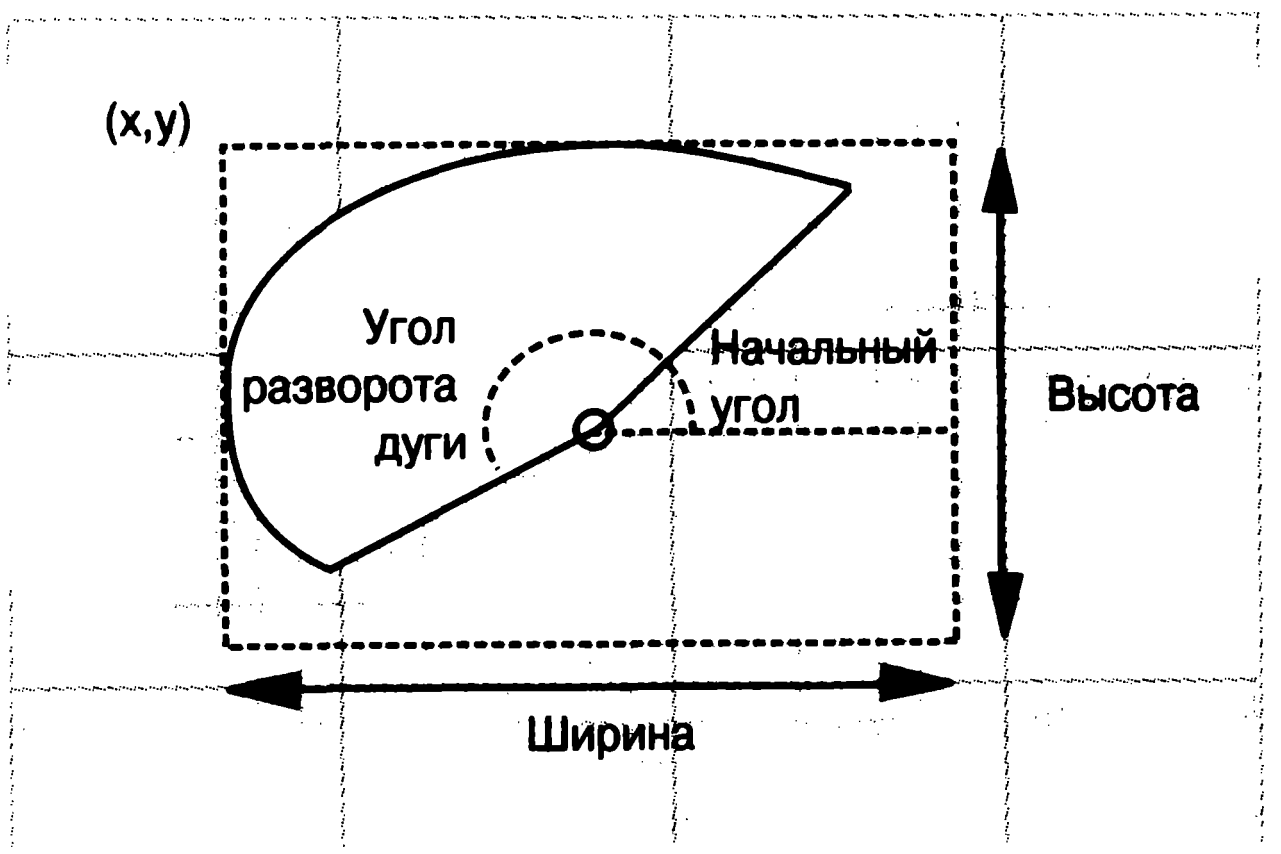


Рис. 7.5. Построение эллиптической дуги

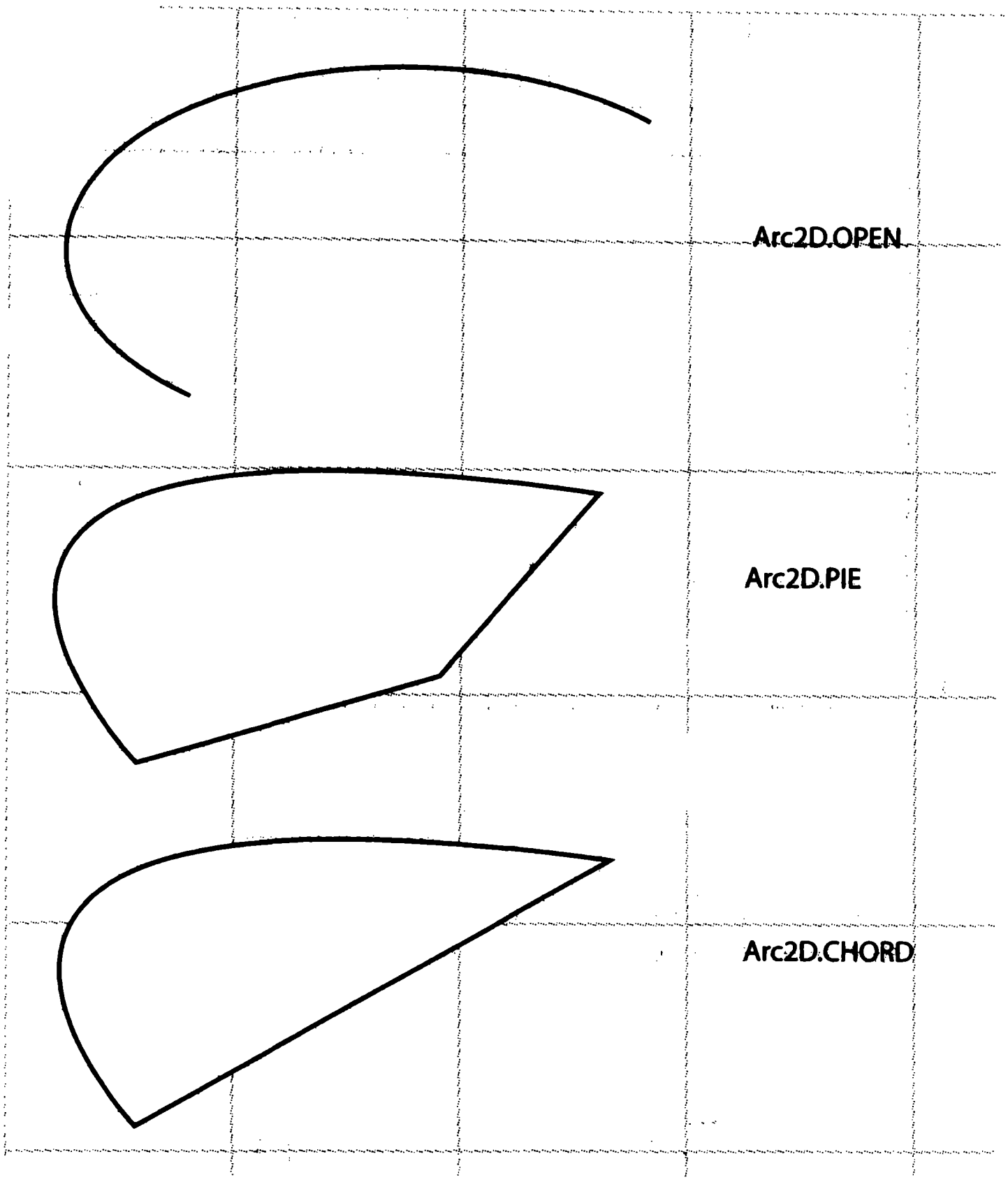


Рис. 7.6. Виды замыкания дуги

ВНИМАНИЕ! Если дуга имеет эллиптическую форму, рассчитать ее углы не так-то просто. По этому поводу в документации на прикладной интерфейс API заявляется следующее: "Углы задаются относительно неквадратного обрамляющего прямоугольника таким образом, чтобы линия, проведенная из центра эллипса к правому верхнем углу обрамляющего прямоугольника, образовывала угол 45° с обеими его сторонами. Если же обрамляющий прямоугольник заметно длиннее по одной оси, чем по другой, углы к началу и концу отрезка дуги будут скашиваться вдоль более длинной оси фрейма". К сожалению, в документации ничего не говорится о том, как рассчитывать это "скашивание". Поэтому покажем, как это делается на практике.

Допустим, центр дуги находится в точке начала отсчета, а точка с координатами (x, y) — на самой дуге. Угол скашивания в этом случае рассчитывается следующим образом:

```
skewedAngle = Math.toDegrees(Math.atan2(-y * height, x * width));
```

В итоге получается значение в пределах от **-180** до **180**. Подобным образом рассчитываются сначала начальный и конечный углы скашивания, а затем их разница. Если получится отрицательное значение, то прибавляется значение **360**. Далее остается лишь предоставить конструктору дуги значения начального угла и разницы двух углов скашивания в качестве параметров.

Запустив на выполнение пример программы, исходный код которой приведен в конце этого раздела, вы сможете сами убедиться, что описанный выше порядок расчета углов скашивания действительно дает правильные значения параметров для конструктора дуги, как будет показано на рис. 7.9.

В прикладном интерфейсе Java 2D API предусмотрены средства для построения кривых *второго* порядка (квадратичных) и *третьего* порядка (кубических). Здесь не рассматриваются математические аспекты построения этих кривых, но для демонстрации их возможностей предлагается запустить программу из листинга 7.1. Как показано на рис. 7.7 и 7.8, кривые второго и третьего порядка определяются двумя *конечными точками* и одной или двумя *управляющими точками* (для кривых второго и третьего порядка соответственно). При перемещении управляющих точек изменяется форма кривых. Для построения кривых второго и третьего порядка задаются координаты конечных и управляющих точек, как показано в приведенном ниже примере кода.

```
QuadCurve2D q = new QuadCurve2D.Double(
    startX, startY, control1X, control1Y, endX, endY);
CubicCurve2D c = new CubicCurve2D.Double(
    startX, startY, control11X, control11Y,
    control12X, control12Y, endX, endY);
```

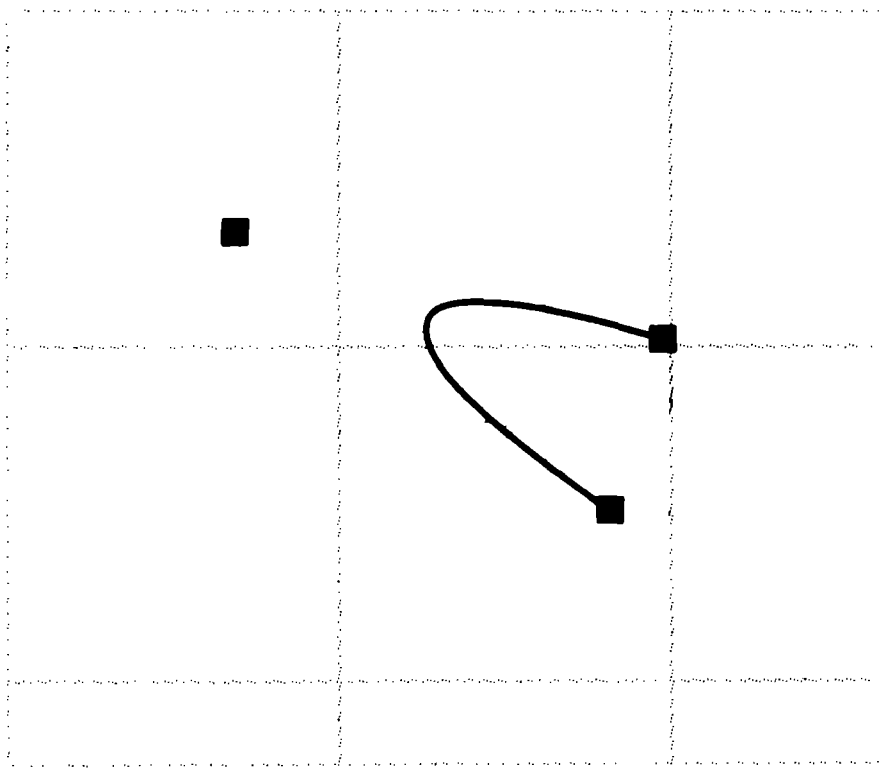


Рис. 7.7. Кривая второго порядка

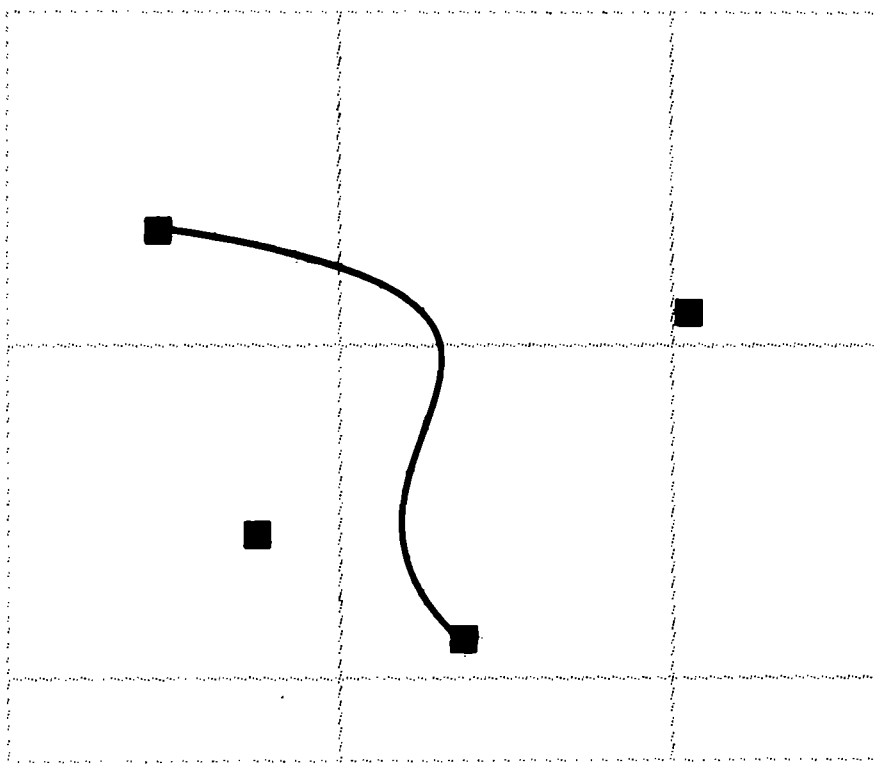


Рис. 7.8. Кривая третьего порядка

Кривые второго порядка не очень удобны, и поэтому они редко используются на практике, в отличие от кривых третьего порядка (например, кривых Безье, вычерчиваемых средствами класса `CubicCurve2D`). Сочетая несколько кривых третьего порядка таким образом, чтобы в точках соединения совпадали углы их наклона, можно строить сложные поверхности. Более подробно об этом можно узнать из книги *Computer Graphics: Principles and Practice, Second Edititon in C* Джеймса Фоли, Андриса ван Дама, Стивена Фэйнера и др. (James D. Foley, Andries van Dam, Steven K. Feiner et al.; издательство Addison Wesley, 1995 г.).

Для построения и сохранения произвольной кривой в виде последовательности линейных отрезков, кривых второго и третьего порядка предусмотрен класс `GeneralPath`. С этой целью методу `moveTo()` нужно передать координату первой точки строящейся кривой следующим образом:

```
GeneralPath path = new GeneralPath();
path.moveTo(10, 20);
```

Построение контура кривой возможно далее с помощью метода `lineTo()`, `quadTo()` или `curveTo()`. Эти методы продолжают кривую линией, кривой второго порядка или кривой третьего порядка соответственно. При вызове метода `lineTo()` следует указать конечную точку, а при вызове двух других методов построения

кривых — сначала управляющие точки и затем конечную точку, как показано ниже. И наконец, замыкается кривая с помощью метода `closePath()`, который проводит линию в обратном направлении к начальной точке контура кривой.

```
path.lineTo(20, 30);
path.curveTo(control1X, control1Y, control2X, control2Y, endX, endY);
```

Чтобы построить многоугольник, следует вызвать метод `moveTo()` для перехода к первой его вершине, а затем несколько раз вызвать метод `lineTo()` для соединения отрезками линии остальных вершин многоугольника. И наконец, нужно вызвать метод `closePath()`, чтобы замкнуть многоугольник. Подробнее об этом — далее, в листинге 7.1. Общий контур вычерчиваемой фигуры совсем не обязательно должен быть соединенным. Это означает, что метод `moveTo()` можно вызвать в любой удобный момент, чтобы начать построение нового отрезка контура.

К общему контуру можно добавить произвольные объекты типа `Shape`, используя для этой цели метод `append()`. При этом контур присоединяемой фигуры добавляется в конце общего контура вычерчиваемой фигуры. Второй параметр этого метода принимает логическое значение `true`, если новую фигуру необходимо соединить с последней точкой общего контура, а иначе — логическое значение `false`. Например, в приведенном ниже фрагменте кода контур прямоугольника добавляется к общему контуру вычерчиваемой фигуры, не соединяясь с этим контуром в его конце.

```
Rectangle2D r = . . .;
path.append(r, false);
```

Но при следующем вызове метода `append()` сначала проводится прямая линия, соединяющая конечную точку общего контура с начальной точкой прямоугольника `r`, а затем контур прямоугольника добавляется к общему контуру вычерчиваемой фигуры:

```
path.append(r, true);
```

На рис. 7.7 и 7.8 показаны результаты выполнения примера программы из листинга 7.1. В рабочем окне этой программы находится раскрывающийся список для выбора следующих видов вычерчиваемых фигур.

- Прямые линии.
- Прямоугольники, в том числе скругленные, а также эллипсы.
- Дуги (помимо самой дуги, отображаются контуры ограничивающего прямоугольника, маркер начального угла и угол разворота дуги).
- Многоугольники (вычерчиваемые средствами класса `GeneralPath`).
- Кривые второго и третьего порядка.

Изменить расположение управляющих точек можно мышью. Обратите внимание на то, что при перемещении этих точек фигура автоматически перерисовывается. Рассматриваемая здесь программа имеет довольно сложную структуру, поскольку она оперирует многими фигурами и поддерживает их автоматическую перерисовку при перетаскивании управляющих точек.

Абстрактный суперкласс `ShapeMaker` инкапсулирует общие функции всех классов для построения фигур. У каждой фигуры имеются управляющие точки, которые пользователь может перемещать, а метод `getPointCount()` возвращает их количество. Для расчета конкретной формы фигуры, исходя из текущего положения управляющих точек, служит следующий абстрактный метод:

```
Shape makeShape(Point2D[] points)
```

Метод `toString()` возвращает имя класса, и благодаря этому объекты типа `ShapeMaker` могут быть просто внесены в комбинированный список типа `JComboBox`. Для активизации режима перетаскивания управляющих точек в классе `ShapePanel` следует предусмотреть обработку событий от мыши. Так, если кнопка мыши нажата над прямоугольным маркером управляющей точки, этот маркер перемещается при последующем движении мыши.

Большинство классов для построения фигур имеют очень простую структуру. Их методы `makeShape()` строят и возвращают требующуюся фигуру. Но в классе `ArcMaker` требуется дополнительно рассчитать начальный и конечный углы скашивания, как пояснялось выше. Поэтому для демонстрации правильности выполненных расчетов возвращаемая фигура вычерчивается средствами класса `GeneralPath`, включая саму дугу, ограничивающий ее прямоугольник и линии, проведенные из центра дуги к управляющим точкам (рис. 7.9).

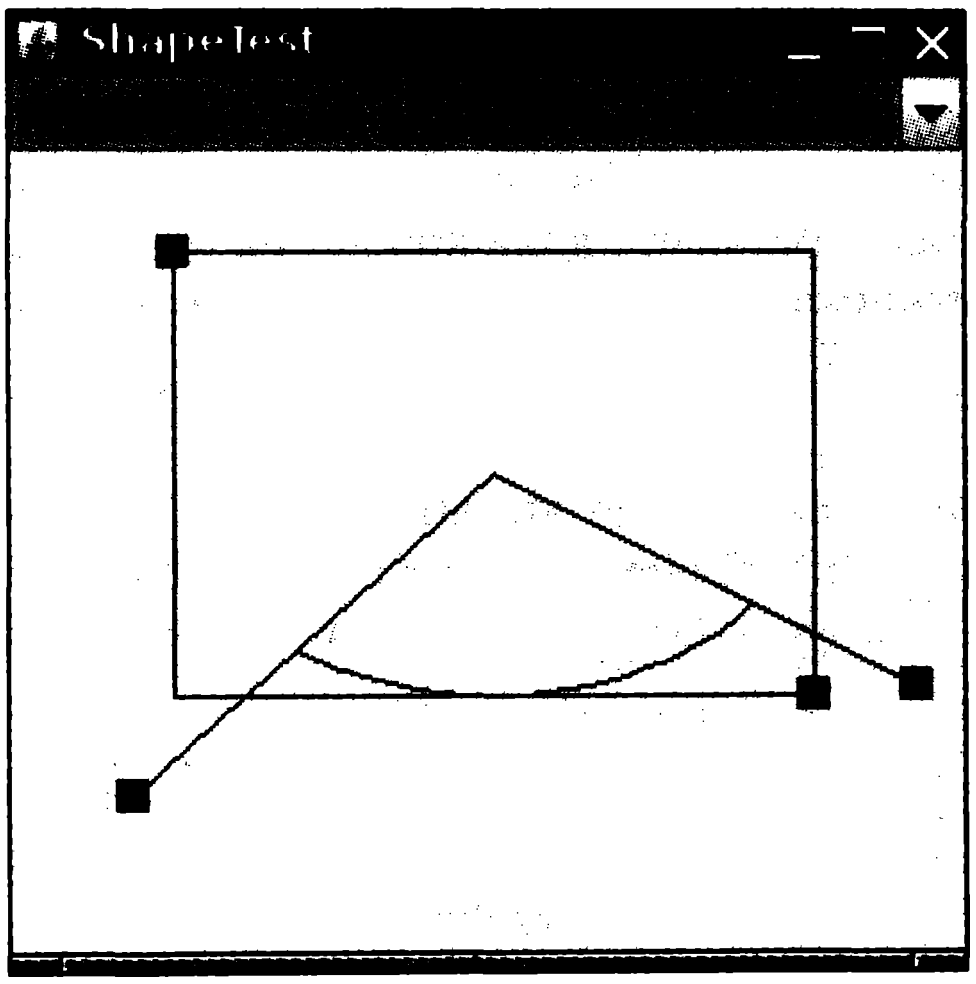


Рис. 7.9. Рабочее окно программы `ShapeTest` с построенной дугой и ее вспомогательными элементами

Листинг 7.1. Исходный код из файла `shape/ShapeTest.java`

```
1  package shape;
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   * В этой программе демонстрируется построение различных двухмерных форм
10  * @version 1.02 2007-08-16
11  * @author Cay Horstmann
12 */
13 public class ShapeTest
14 {
15   public static void main(String[] args)
```

```
16    {
17      EventQueue.invokeLater(new Runnable()
18        {
19          public void run()
20          {
21            JFrame frame = new ShapeTestFrame();
22            frame.setTitle("ShapeTest");
23            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24            frame.setVisible(true);
25          }
26        });
27    }
28  }
29
30  /**
31   * Этот фрейм содержит комбинированный список для выбора фигур,
32   * а также компонент для их рисования
33  */
34  class ShapeTestFrame extends JFrame
35  {
36    public ShapeTestFrame()
37    {
38      final ShapeComponent comp = new ShapeComponent();
39      add(comp, BorderLayout.CENTER);
40      final JComboBox<ShapeMaker> comboBox = new JComboBox<>();
41      comboBox.addItem(new LineMaker());
42      comboBox.addItem(new RectangleMaker());
43      comboBox.addItem(new RoundRectangleMaker());
44      comboBox.addItem(new EllipseMaker());
45      comboBox.addItem(new ArcMaker());
46      comboBox.addItem(new PolygonMaker());
47      comboBox.addItem(new QuadCurveMaker());
48      comboBox.addItem(new CubicCurveMaker());
49      comboBox.addActionListener(new ActionListener()
50        {
51          public void actionPerformed(ActionEvent event)
52          {
53            ShapeMaker shapeMaker =
54            comboBox.getItemAt(comboBox.getSelectedIndex());
55            comp.setShapeMaker(shapeMaker);
56          }
57        });
58      add(comboBox, BorderLayout.NORTH);
59      comp.setShapeMaker((ShapeMaker) comboBox.getItemAt(0));
60      pack();
61    }
62
63  }
64
65  /**
66   * Этот компонент рисует фигуру и дает пользователю возможность
67   * перемещать определяющие ее точки
68  */
69  class ShapeComponent extends JComponent
70  {
71    private static final int DEFAULT_WIDTH = 300;
72    private static final int DEFAULT_HEIGHT = 200;
73
```

```
74    private Point2D[] points;
75    private static Random generator = new Random();
76    private static int SIZE = 10;
77    private int current;
78    private ShapeMaker shapeMaker;
79
80    public ShapeComponent()
81    {
82      addMouseListener(new MouseAdapter()
83        {
84          public void mousePressed(MouseEvent event)
85          {
86            Point p = event.getPoint();
87            for (int i = 0; i < points.length; i++)
88            {
89              double x = points[i].getX() - SIZE / 2;
90              double y = points[i].getY() - SIZE / 2;
91              Rectangle2D r = new Rectangle2D.Double(x, y, SIZE, SIZE);
92              if (r.contains(p))
93              {
94                current = i;
95                return;
96              }
97            }
98          }
99
100         public void mouseReleased(MouseEvent event)
101         {
102           current = -1;
103         }
104       });
105     addMouseMotionListener(new MouseMotionAdapter()
106       {
107         public void mouseDragged(MouseEvent event)
108         {
109           if (current == -1) return;
110           points[current] = event.getPoint();
111           repaint();
112         }
113       });
114     current = -1;
115   }
116
117   /**
118    * Устанавливает построитель фигур и инициализирует
119    * его произвольным рядом точек
120    * @param aShapeMaker Построитель фигур, определяющий фигуру
121    *                    по ряду исходных точек
122   */
123   public void setShapeMaker(ShapeMaker aShapeMaker)
124   {
125     shapeMaker = aShapeMaker;
126     int n = shapeMaker.getPointCount();
127     points = new Point2D[n];
128     for (int i = 0; i < n; i++)
129     {
130       double x = generator.nextDouble() * getWidth();
131       double y = generator.nextDouble() * getHeight();
```

```
132         points[i] = new Point2D.Double(x, y);
133       }
134       repaint();
135    }
136
137    public void paintComponent(Graphics g)
138    {
139       if (points == null) return;
140       Graphics2D g2 = (Graphics2D) g;
141       for (int i = 0; i < points.length; i++)
142       {
143          double x = points[i].getX() - SIZE / 2;
144          double y = points[i].getY() - SIZE / 2;
145          g2.fill(new Rectangle2D.Double(x, y, SIZE, SIZE));
146       }
147
148       g2.draw(shapeMaker.makeShape(points));
149    }
150
151    public Dimension getPreferredSize()
152       { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
153 }
154
155 /**
156  * Построитель фигур по ряду исходных точек. Конкретные подклассы
157  * должны возвращать форму из метода makeShape()
158 */
159 abstract class ShapeMaker
160 {
161    public abstract Shape makeShape(Point2D[] p);
162    private int pointCount;
163
164    /**
165     * Конструирует построитель фигур
166     * @param aPointCount Количество исходных точек, требующихся
167     *                    для определения данной фигуры
168    */
169    public ShapeMaker(int aPointCount)
170    {
171       pointCount = aPointCount;
172    }
173
174    /**
175     * Получает количество исходных точек, требующихся
176     * для определения данной фигуры
177     * @return Возвращает количество исходных точек
178    */
179    public int getPointCount()
180    {
181       return pointCount;
182    }
183
184    /**
185     * Строит фигуру по заданному ряду исходных точек
186     * @param p Исходные точки, определяющие фигуру
187     * @return Возвращает Фигуру, определяемую исходными точками
188    */
189
```

```
190   public String toString()
191   {
192     return getClass().getName();
193   }
194
195 }
196
197 /**
198  * Проводит линию, соединяющую две заданные точки
199 */
200 class LineMaker extends ShapeMaker
201 {
202   public LineMaker()
203   {
204     super(2);
205   }
206
207   public Shape makeShape(Point2D[] p)
208   {
209     return new Line2D.Double(p[0], p[1]);
210   }
211 }
212
213 /**
214   * Строит прямоугольник, соединяющий две заданные угловые точки
215 */
216 class RectangleMaker extends ShapeMaker
217 {
218   public RectangleMaker()
219   {
220     super(2);
221   }
222
223   public Shape makeShape(Point2D[] p)
224   {
225     Rectangle2D s = new Rectangle2D.Double();
226     s.setFrameFromDiagonal(p[0], p[1]);
227     return s;
228   }
229 }
230
231 /**
232  * Строит прямоугольник со скругленными углами, соединяющий
233  * две заданные угловые точки
234 */
235 class RoundRectangleMaker extends ShapeMaker
236 {
237   public RoundRectangleMaker()
238   {
239     super(2);
240   }
241
242   public Shape makeShape(Point2D[] p)
243   {
244     RoundRectangle2D s =
245        new RoundRectangle2D.Double(0, 0, 0, 0, 20, 20);
246     s.setFrameFromDiagonal(p[0], p[1]);
247     return s;
```

```
248    }
249 }
250
251 /**
252    * Строит эллипс, вписанный в ограничивающий прямоугольник
253    * с двумя заданными угловыми точками
254 */
255 class EllipseMaker extends ShapeMaker
256 {
257    public EllipseMaker()
258    {
259       super(2);
260    }
261
262    public Shape makeShape(Point2D[] p)
263    {
264       Ellipse2D s = new Ellipse2D.Double();
265       s.setFrameFromDiagonal(p[0], p[1]);
266       return s;
267    }
268 }
269
270 /**
271  * Строит дугу, вписанную в ограничивающий прямоугольник с двумя
272  * заданными угловыми точками. Начальный и конечный углы дуги задают
273  * линии, проведенные из центра ограничивающего прямоугольника в две
274  * заданные точки. Для демонстрации правильности расчета углов
275  * возвращаемая фигура состоит из дуги, ограничивающего
276  * прямоугольника и линий, образующих эти углы
277 */
278 class ArcMaker extends ShapeMaker
279 {
280    public ArcMaker()
281    {
282       super(4);
283    }
284
285    public Shape makeShape(Point2D[] p)
286    {
287       double centerX = (p[0].getX() + p[1].getX()) / 2;
288       double centerY = (p[0].getY() + p[1].getY()) / 2;
289       double width = Math.abs(p[1].getX() - p[0].getX());
290       double height = Math.abs(p[1].getY() - p[0].getY());
291
292       double skewedStartAngle =
293          Math.toDegrees(Math.atan2(-(p[2].getY() - centerY) * width,
294          (p[2].getX() - centerX) * height));
295       double skewedEndAngle =
296          Math.toDegrees(Math.atan2(-(p[3].getY() - centerY) * width,
297          (p[3].getX() - centerX) * height));
298       double skewedAngleDifference = skewedEndAngle - skewedStartAngle;
299       if (skewedStartAngle < 0) skewedStartAngle += 360;
300       if (skewedAngleDifference < 0) skewedAngleDifference += 360;
301
302       Arc2D s = new Arc2D.Double(0, 0, 0, 0, skewedStartAngle,
303          skewedAngleDifference, Arc2D.OPEN);
304       s.setFrameFromDiagonal(p[0], p[1]);
305
```

```
306       GeneralPath g = new GeneralPath();
307       g.append(s, false);
308       Rectangle2D r = new Rectangle2D.Double();
309       r.setFrameFromDiagonal(p[0], p[1]);
310       g.append(r, false);
311       Point2D center = new Point2D.Double(centerX, centerY);
312       g.append(new Line2D.Double(center, p[2]), false);
313       g.append(new Line2D.Double(center, p[3]), false);
314       return g;
315    }
316 }
317
318 /**
319    * Строит многоугольник, определяемый шестью угловыми точками
320 */
321 class PolygonMaker extends ShapeMaker
322 {
323    public PolygonMaker()
324    {
325       super(6);
326    }
327
328    public Shape makeShape(Point2D[] p)
329    {
330       GeneralPath s = new GeneralPath();
331       s.moveTo((float) p[0].getX(), (float) p[0].getY());
332       for (int i = 1; i < p.length; i++)
333       s.lineTo((float) p[i].getX(), (float) p[i].getY());
334       s.closePath();
335       return s;
336    }
337 }
338
339 /**
340    * Строит кривую второго порядка, определяемую двумя конечными
341    * точками и одной управляющей точкой
342    */
343 class QuadCurveMaker extends ShapeMaker
344 {
345    public QuadCurveMaker()
346    {
347       super(3);
348    }
349
350    public Shape makeShape(Point2D[] p)
351    {
352       return new QuadCurve2D.Double(p[0].getX(), p[0].getY(),
353          p[1].getX(), p[1].getY(), p[2].getX(), p[2].getY());
354    }
355 }
356
357 /**
358    * Строит кривую третьего порядка, определяемую двумя конечными
359    * точками и двумя управляющими точками
360 */
361 class CubicCurveMaker extends ShapeMaker
362 {
363    public CubicCurveMaker()
```

```
364    {
365       super(4);
366    }
367
368    public Shape makeShape(Point2D[] p)
369    {
370       return new CubicCurve2D.Double(p[0].getX(), p[0].getY(),
371          p[1].getX(), p[1].getY(),p[2].getX(),
372          p[2].getY(), p[3].getX(), p[3].getY());
373    }
374 }
```

java.awt.geom.RoundRectangle2D.Double 1.2

- RoundRectangle2D.Double(double *x*, double *y*, double *width*, double *height*, double *arcWidth*, double *arcHeight*)

  Строит прямоугольник со скругленными углами, исходя из заданных размеров ограничивающего прямоугольника и дуги скругления. Наглядное представление о параметрах *arcWidth* и *arcHeight* дает рис. 7.4.

java.awt.geom.Arc2D.Double 1.2

- Arc2D.Double(double *x*, double *y*, double *w*, double *h*, double *startAngle*, double *arcAngle*, int *type*)

  Строит дугу, исходя из заданного ограничивающего прямоугольника, начального и конечного угла, также вида замыкания дуги. Наглядное представление о параметрах startAngle и arcAngle дает рис. 7.5. Что касается параметра type, то он может принимать значение одной из следующих констант: Arc2D.OPEN, Arc2D.PIE или Arc2D.CHORD.

java.awt.geom.QuadCurve2D.Double 1.2

- QuadCurve2D.Double(double *x1*, double *y1*, double *ctrlx*, double *ctrly*, double *x2*, double *y2*)

  Строит кривую второго порядка по начальной, управляющей и конечной точкам.

java.awt.geom.CubicCurve2D.Double 1.2

- CubicCurve2D.Double(double *x1*, double *y1*, double *ctrlx1*, double *ctrly1*, double *ctrlx2*, double *ctrly2*, double *x2*, double *y2*)

  Создает кривую третьего порядка по начальной, двум контрольным и конечной точкам.

java.awt.geom.GeneralPath 1.2

- `GeneralPath()`

  Строит пустой общий контур

java.awt.geom.Path2D.Float 6

- `void moveTo(float x, float y)`

  Строит *текущую точку* с координатами *x* и *y*, т.е. начальную точку следующего отрезка линии.

- `void lineTo(float x, float y)`
- `void quadTo(float ctrlx, float ctrly, float x, float y)`
- `void curveTo(float ctrl1x, float ctrl1y, float ctrl2x, float ctrl2y, float x, float y)`

  Рисуют прямую линию, кривую второго или третьего порядка от текущей точки к конечной с координатами *x* и *y*, после чего конечная точка становится текущей.

java.awt.geom.Path2D 6

- `void append(Shape s, boolean connect)`

  Присоединяет контур заданной фигуры к общему контуру. Если параметр *connect* принимает логическое значение `true`, то текущая точка общего контура соединяется прямой линией с начальной точкой присоединяемой фигуры.

- `void closePath()`

  Замыкает контур, рисуя прямую линию от текущей точки к первой точке контура.

Участки

В предыдущем разделе обсуждались средства построения сложных фигур с помощью прямых линий и кривых второго и третьего порядка. Используя достаточное количество линий и кривых, можно нарисовать практически любую фигуру. Например, контуры символов в шрифтах, которые отображаются на экране и на отпечатке, состоят из прямых линий и кривых третьего порядка.

Но иногда описать фигуру легче, составляя ее из *участков*, например, прямоугольных, многоугольных или овальных. В прикладном интерфейсе Java 2D API поддерживаются четыре метода, выполняющих геометрические операции построения нового участка из двух исходных участков. Результаты выполнения этих операций приведены на рис. 7.10.

- Метод `add()` составляет участок, который содержит все точки первого или второго исходного участка. Это операция геометрического сложения участков.
- Метод `subtract()` составляет участок, который содержит все точки первого исходного участка, не входящие во второй исходный участок. Это операция геометрического вычитания участков.
- Метод `intersect()` составляет участок, который содержит только точки, принадлежащие одновременно первому и второму исходному участкам. Это операция геометрического пересечения участков.
- Метод `exclusiveOr()` составляет участок, который содержит все точки, принадлежащие первому или второму исходному участку, но не обоим вместе. Это операция геометрического исключения участков.

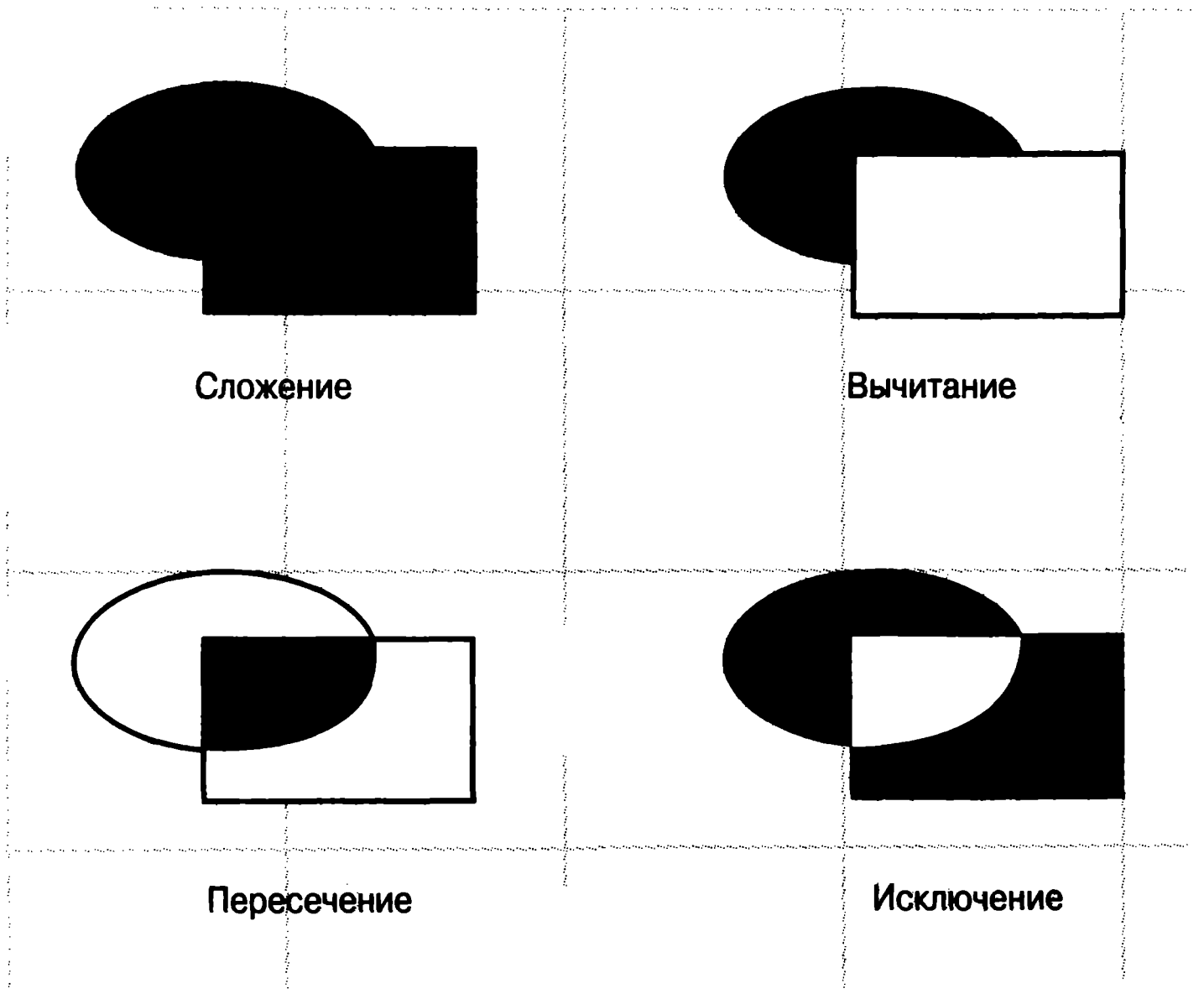


Рис. 7.10. Геометрические операции построения нового участка из двух исходных участков

Для того чтобы построить участок сложной геометрической формы, нужно создать сначала исходный участок следующим образом:

```
Area a = new Area();
```

Затем следует объединить этот участок с фигурой нужной формы, выполнив одну из упомянутых выше геометрических операции:

```
a.add(new Rectangle2D.Double(. . .));
a.subtract(path);
. . .
```

Класс `Area` для построения участков реализует интерфейс `Shape`. В процессе построения участка можно очертить его границу с помощью метода `draw()` или заполнить внутреннюю его часть с помощью метода `fill()` из класса `Graphics2D`.

java.awt.geom.Area

- `void add(Area other)`
- `void subtract(Area other)`
- `void intersect(Area other)`
- `void exclusiveOr(Area other)`

  Выполняют геометрические операции построения нового участка, объединяя текущий участок с другим участком, определяемым параметром *other*. Получающийся в итоге участок становится текущим.

Обводка

Метод `draw()` из класса `Graphics2D` рисует границу фигуры, используя выбранный в настоящий момент вид *обводки*. По умолчанию для обводки выбирается сплошная линия толщиной в один пиксель. Для изменения вида обводки предусмотрен метод `setStroke()`, которому в качестве параметра передается экземпляр класса, реализующего интерфейс `Stroke`. В прикладном интерфейсе Java 2D API определен только один такой класс, а именно `BasicStroke`. В этом разделе рассматриваются функциональные возможности данного класса.

Для обводки можно задать линию произвольной толщины. Например, для выбора линии толщиной 10 пикселей и последующей обводки служит следующий код:

```
g2.setStroke(new BasicStroke(10.0F));
g2.draw(new Line2D.Double(. . .));
```

Если толщина линии обводки превышает один пиксель, то для формирования концов линии можно использовать один из перечисленных ниже стилей (рис. 7.11).

- **Торцевой конец линии.** Линия обводки обрывается в конечной точке.
- **Скругленный конец линии.** В конце линии обводки добавляется полукруг.
- **Квадратный конец линии.** В конце линии обводки добавляется полуквадрат.

Для соединения двух толстых линий обводки можно также указать один из следующих стилей (рис. 7.12).

- **Косой стык.** Соединяет линии обводки по прямой линии, перпендикулярной биссектрисе угла между ними.
- **Скругленный стык.** Соединяет линии обводки, образуя скругленный конец линии.
- **Угловой стык.** Соединяет линии обводки, образуя клинообразный конец линии.

Угловой стык не совсем подходит для линий обводки, которые пересекаются под небольшими углами. Так, если две линии обводки пересекаются под углами, меньшими *предела среза*, то вместо углового используется косой стык. Благодаря этому предотвращается образование чрезмерно длинных клиньев. По умолчанию предел среза составляет 10°. Стили окончания и соединения линий обводки можно указать в конструкторе класса `BasicStroke`, как показано ниже.

Рис. 7.11. Стили окончания линии обводки

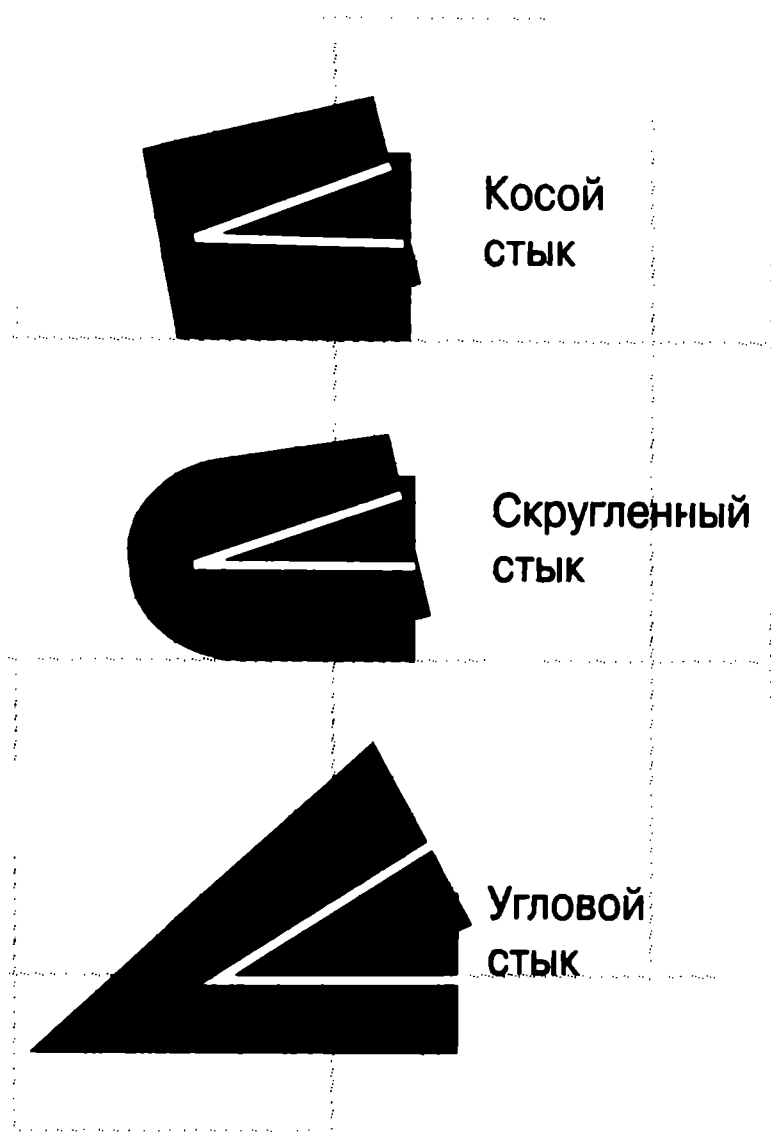


Рис. 7.12. Стили соединения линий обводки

```
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_ROUND,
     BasicStroke.JOIN_ROUND));
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT,
     BasicStroke.JOIN_MITER, 15.0F /* miter limit */));
```

И наконец, для пунктирных линий обводки можно выбрать конкретный *пунктир*. В примере программы из листинга 7.2 используется пунктир, соответствующий сигналу SOS в азбуке Морзе. Пунктир задается в виде массива типа `float[]`, который содержит длины рисуемых и пробельных отрезков обводки (рис. 7.13).

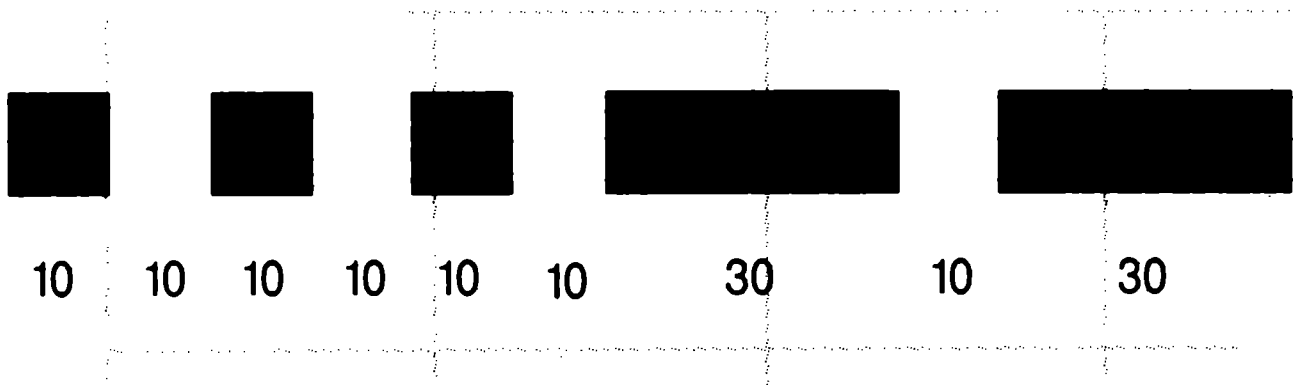


Рис. 7.13. Пунктир для обводки

При создании объекта типа `BasicStroke`, помимо самого пунктира, указывается также *фаза пунктира*, т.е. место начала пунктира на линии обводки. Обычно фаза пунктира устанавливается равной нулю. В приведенном ниже примере кода показано, каким образом задается пунктирный стиль линий обводки.

```
float[] dashPattern = { 10, 10, 10, 10, 10, 10, 30, 10, 30, ... };
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT,
    BasicStroke.JOIN_MITER, 10.0F /* предел среза */,
    dashPattern, 0 /* фаза пунктира */));
```

НА ЗАМЕТКУ! Стили окончания линий распространяются и на все штрихи пунктира.

В примере программы, исходный код которой приведен в листинге 7.2, предоставляется возможность указать стиль окончания линий обводки, стиль их соединения и пунктир (рис. 7.14). Кроме того, концы линий обводки можно перемещать, чтобы проверить действенность предела среза. Для этого следует выбрать угловой стык, а затем перетащить мышью отрезок линии обводки таким образом, чтобы две линии образовали очень острый угол. По достижения предела среза угловой стык автоматически превращается в косой.

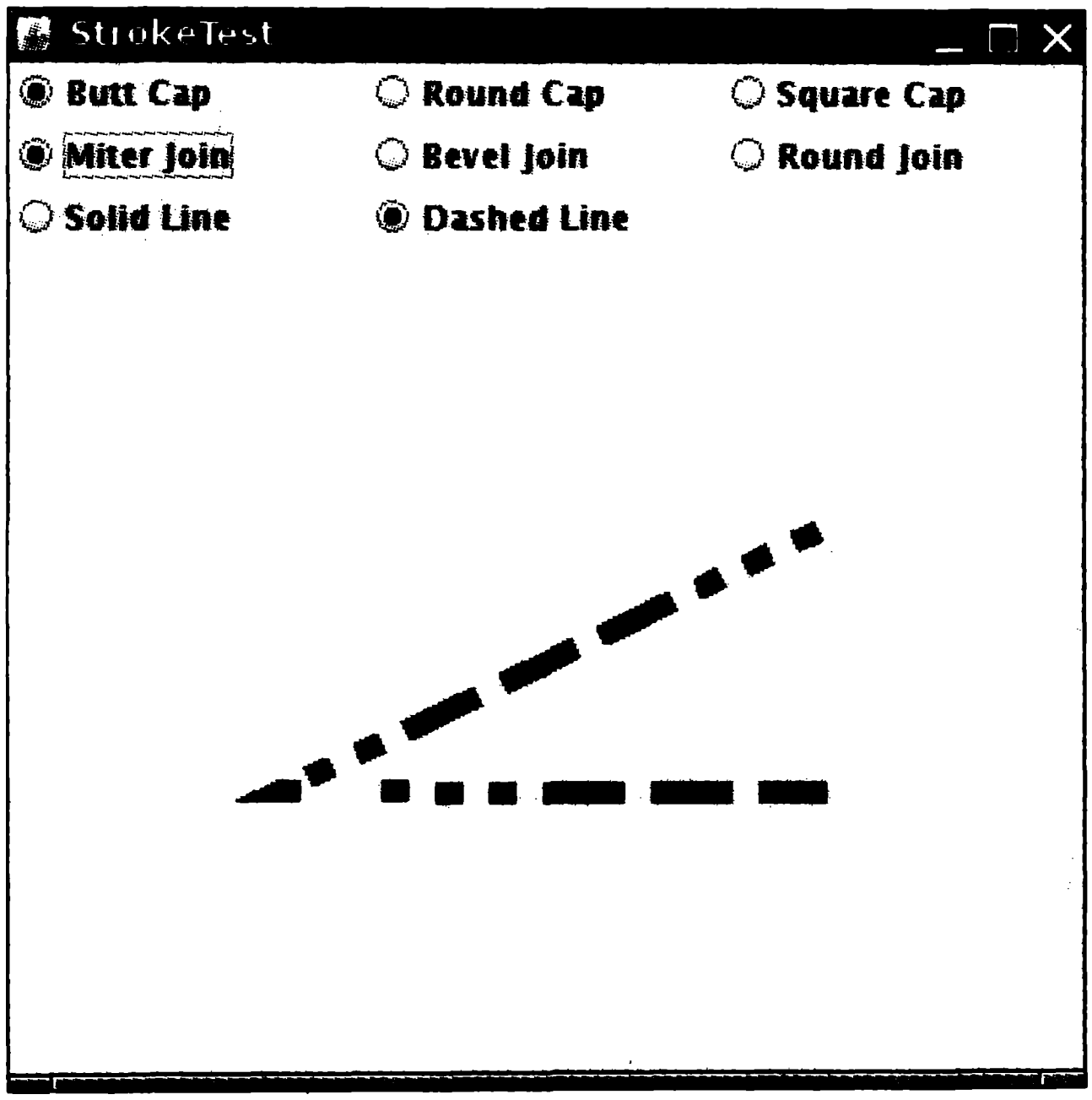


Рис. 7.14. Рабочее окно программы `StrokeTest`

Рассматриваемая здесь программа похожа на программу из листинга 7.1. Обработчик событий от мыши реагирует на щелчок в конечной точке линии обводки и при перемещении курсора мыши будет передвигать вслед за ним конец линии обводки. Группа кнопок-переключателей уведомляет о выборе пользователем стиля окончания и соединения линии обводки, а также сплошной или пунктирной линии. Метод `paintComponent()` из класса `StrokePanel` используется для построения общего контура типа `GeneralPath` из двух отрезков, соединяющих три точки, которые пользователь может перемещать с помощью мыши. Затем создается объект типа `BasicStroke`, исходя из выбора, сделанного пользователем. И наконец, линия обводки рисуется по сформированному контуру.

Листинг 7.2. Исходный код из файла `stroke/StrokeTest.java`

```
1   package stroke;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.awt.geom.*;
6   import javax.swing.*;
7
8   /**
9    * В этой программе демонстрируются различные виды обводки
10  * @version 1.03 2007-08-16
11  * @author Cay Horstmann
12  */
13  public class StrokeTest
14  {
15    public static void main(String[] args)
16    {
17      EventQueue.invokeLater(new Runnable()
18        {
19          public void run()
20          {
21            JFrame frame = new StrokeTestFrame();
22            frame.setTitle("StrokeTest");
23            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24            frame.setVisible(true);
25          }
26        });
27    }
28  }
29
30  /**
31   * В этом фрейме пользователь может выбрать стиль окончания и соединения
32   * линий обводки, а также увидеть получающуюся в итоге линию обводки
33  */
34  class StrokeTestFrame extends JFrame
35  {
36    private StrokeComponent canvas;
37    private JPanel buttonPanel;
38
39    public StrokeTestFrame()
40    {
41      canvas = new StrokeComponent();
42      add(canvas, BorderLayout.CENTER);
43
44      buttonPanel = new JPanel();
45      buttonPanel.setLayout(new GridLayout(3, 3));
46      add(buttonPanel, BorderLayout.NORTH);
47
48      ButtonGroup group1 = new ButtonGroup();
49      makeCapButton("Butt Cap", BasicStroke.CAP_BUTT, group1);
50      makeCapButton("Round Cap", BasicStroke.CAP_ROUND, group1);
51      makeCapButton("Square Cap", BasicStroke.CAP_SQUARE, group1);
52
53      ButtonGroup group2 = new ButtonGroup();
54      makeJoinButton("Miter Join", BasicStroke.JOIN_MITER, group2);
55      makeJoinButton("Bevel Join", BasicStroke.JOIN_BEVEL, group2);
```

```
56      makeJoinButton("Round Join", BasicStroke.JOIN_ROUND, group2);
57
58      ButtonGroup group3 = new ButtonGroup();
59      makeDashButton("Solid Line", false, group3);
60      makeDashButton("Dashed Line", true, group3);
61    }
62
63    /**
64     * Создает кнопку-переключатель для выбора стиля окончания линии
65     * @param label Метка кнопки-переключателя
66     * @param style Стиль окончания линии
66     * @param group Группа кнопок-переключателей
67    */
68    private void makeCapButton(
69       String label, final int style, ButtonGroup group)
70    {
71      // выбрать первую кнопку-переключатель в группе
72      boolean selected = group.getButtonCount() == 0;
73      JRadioButton button = new JRadioButton(label, selected);
74      buttonPanel.add(button);
75      group.add(button);
76      button.addActionListener(new ActionListener()
77        {
78          public void actionPerformed(ActionEvent event)
79          {
80            canvas.setCap(style);
81          }
82        });
83      pack();
84    }
85
86    /**
87     * Создает кнопку-переключатель для выбора стиля соединения линий
88     * @param label Метка кнопки-переключателя
89     * @param style Стиль соединения линий
90     * @param group Группа кнопок-переключателей
91    */
92    private void makeJoinButton(
93       String label, final int style, ButtonGroup group)
94    {
95      // выбрать первую кнопку-переключатель в группе
96      boolean selected = group.getButtonCount() == 0;
97      JRadioButton button = new JRadioButton(label, selected);
98      buttonPanel.add(button);
99      group.add(button);
100     button.addActionListener(new ActionListener()
101       {
102         public void actionPerformed(ActionEvent event)
103         {
104           canvas.setJoin(style);
105         }
106       });
107   }
108
109   /**
110    * Создает кнопку-переключатель для выбора сплошных или
111    * пунктирных линий обводки
112    * @param label Метка кнопки-переключателя
```

```
113     * @param style Принимает логическое значение false, если линия
114     * сплошная, или логическое значение true, если линия пунктирная
115     * @param group Группа кнопок-переключателей
116    */
117    private void makeDashButton(
116      String label, final boolean style, ButtonGroup group)
117    {
118      // выбрать первую кнопку-переключатель в группе
119      boolean selected = group.getButtonCount() == 0;
120      JRadioButton button = new JRadioButton(label, selected);
121      buttonPanel.add(button);
122      group.add(button);
123      button.addActionListener(new ActionListener()
124        {
125          public void actionPerformed(ActionEvent event)
126          {
127            canvas.setDash(style);
128          }
129        });
130    }
131 }
132
133 /**
134  * Этот компонент рисует две соединенные линии, используя разные
135  * объекты обводки и давая пользователю возможность перетаскивать
136  * три точки, определяющие линии обводки
137 */
138 class StrokeComponent extends JComponent
139 {
140   private static final int DEFAULT_WIDTH = 400;
141   private static final int DEFAULT_HEIGHT = 400;
142   private static int SIZE = 10;
143
144   private Point2D[] points;
145   private int current;
146   private float width;
147   private int cap;
148   private int join;
149   private boolean dash;
150
151   public StrokeComponent()
152   {
153     addMouseListener(new MouseAdapter()
154       {
155         public void mousePressed(MouseEvent event)
156         {
157           Point p = event.getPoint();
158           for (int i = 0; i < points.length; i++)
159           {
160             double x = points[i].getX() - SIZE / 2;
161             double y = points[i].getY() - SIZE / 2;
162             Rectangle2D r = new Rectangle2D.Double(x, y, SIZE, SIZE);
163             if (r.contains(p))
164             {
165               current = i;
166               return;
167             }
168           }
```

```
169         }
170
171         public void mouseReleased(MouseEvent event)
172         {
173           current = -1;
174         }
175       });
176
177     addMouseMotionListener(new MouseMotionAdapter()
178       {
179         public void mouseDragged(MouseEvent event)
180         {
181           if (current == -1) return;
182           points[current] = event.getPoint();
183           repaint();
184         }
185       });
188
189     points = new Point2D[3];
190     points[0] = new Point2D.Double(200, 100);
191     points[1] = new Point2D.Double(100, 200);
192     points[2] = new Point2D.Double(200, 200);
193     current = -1;
194     width = 8.0F;
195   }
196
197   public void paintComponent(Graphics g)
198   {
199     Graphics2D g2 = (Graphics2D) g;
200     GeneralPath path = new GeneralPath();
201     path.moveTo((float) points[0].getX(), (float) points[0].getY());
202     for (int i = 1; i < points.length; i++)
203     path.lineTo((float) points[i].getX(), (float) points[i].getY());
204     BasicStroke stroke;
205     if (dash)
206     {
207       float miterLimit = 10.0F;
208       float[] dashPattern = {
209         10F, 10F, 10F, 10F, 10F, 10F, 30F,
210         10F, 30F, 10F, 30F, 10F, 10F,
211         10F, 10F, 10F, 10F, 30F };
212       float dashPhase = 0;
213       stroke = new BasicStroke(
214         width, cap, join, miterLimit, dashPattern, dashPhase);
215     }
216     else stroke = new BasicStroke(width, cap, join);
217     g2.setStroke(stroke);
218     g2.draw(path);
219   }
220
221   /**
222    * Устанавливает стиль соединения линий
223    * @param j Стиль соединения линий
224   */
225   public void setJoin(int j)
226   {
227     join = j;
228     repaint();
```

```
229    }
230
231    /**
232     * Устанавливает стиль окончания линий
233     * @param c Стиль окончания линий
234    */
235    public void setCap(int c)
236    {
237     cap = c;
238     repaint();
239    }
240
241    /**
242     * Устанавливает сплошные или пунктирные линии
243     * @param d Принимает логическое значение false, если линии
244     * сплошные, или логическое значение true, если линии пунктирные
245    */
246    public void setDash(boolean d)
247    {
248      dash = d;
249      repaint();
250    }
251
252    public Dimension getPreferredSize()
253       { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
254 }
```

java.awt.Graphics2D 1.2

- `void setStroke(Stroke s)`

  Устанавливает в качестве обводки для данного графического контекста указанный объект, класс которого реализует интерфейс `Stroke`.

java.awt.BasicStroke 1.2

- `BasicStroke(float width)`
- `BasicStroke(float width, int cap, int join)`
- `BasicStroke(float width, int cap, int join, float miterlimit)`
- `BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dashPhase)`

  Конструируют объект обводки с заданными свойствами.

  Параметры:

width	Толщина обводки
cap	Стиль окончания линии: CAP_BUTT, CAP_ROUND или CAP_SQUARE
join	Стиль соединения линий: JOIN_BEVEL, JOIN_MITER или

- `BasicStroke(float width)`
- `BasicStroke(float width, int cap, int join)`
- `BasicStroke(float width, int cap, int join, float miterlimit)`
- `BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dashPhase)`

Конструируют объект обводки с заданными свойствами.

Параметры:		
	width	Толщина обводки
	cap	Стиль окончания линии: CAP_BUTT, CAP_ROUND или CAP_SQUARE
	join	Стиль соединения линий: JOIN_BEVEL, JOIN_MITER или JOIN_ROUND
	miterlimit	Предел среза (в градусах), ниже которого угловой стык превращается в косой
	dash	Пунктир в виде массива длин отображаемых и пробельных отрезков обводки
	dashPhase	Фаза пунктира — длина отрезка, предшествующего первой точке линии обводки, при условии, что пунктир уже применяется до этой точки

Раскраска

Внутренняя часть фигуры заполняется путем *раскраски*. Для установки стиля раскраски служит объект, класс которого реализует интерфейс `Paint`. Этот объект передается методу `setPaint()`. В прикладном интерфейсе Java 2D API для этой цели предусмотрены следующие три класса.

- Класс `Color` служит для заполнения фигуры сплошным цветом. Для этого достаточно вызвать метод `setPaint()`, указав объект типа `Color`:

  ```
  g2.setPaint(Color.red);
  ```

- Класс `GradientPaint` служит для заполнения фигуры градиентом, т.е. цветом, постепенно изменяющим свой оттенок (рис. 7.15).
- Класс `TexturePaint` служит для заполнения фигуры текстурой, т.е. повторяющимся рисунком (рис. 7.16).

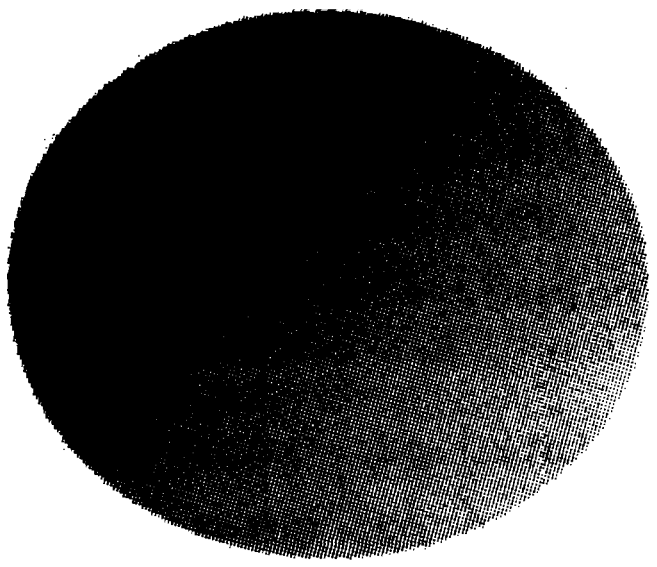


Рис. 7.15. Градиентная раскраска

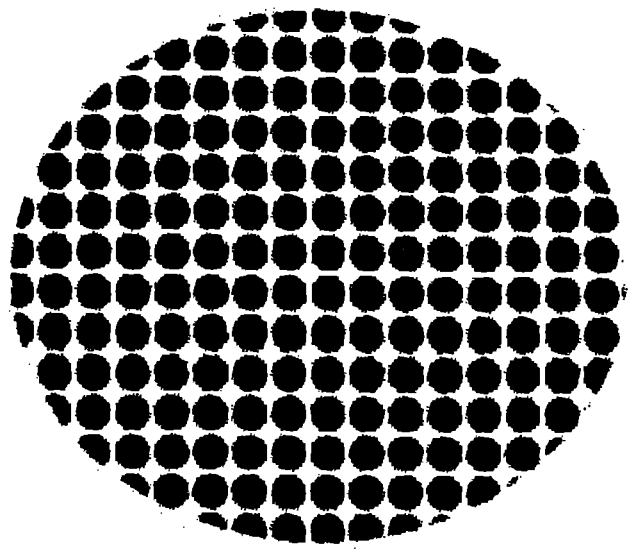


Рис. 7.16. Текстурная раскраска

Для создания объекта типа `GradientPaint` нужно указать две точки изменения градиента и цвета, которые должны сменять друг друга в этих точках, как показано ниже. Постепенное изменение цвета осуществляется по прямой, соединяющей эти точки. Цвета остаются постоянными вдоль прямых, перпендикулярных к линии соединения. Для точек, которые находятся за пределами соединительной линии, задаются цвета ее конечных точек соответственно.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW));
```

Если же вызвать конструктор класса `GradientPaint` с параметром `cyclic`, принимающим логическое значение `true`, то изменение цвета будет происходить *циклически* за пределами конечных точек соединительной линии.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW, true));
```

Для создания объекта типа `TexturePaint` нужно указать объект типа `BufferImage` повторяющегося рисунка и *прямоугольник привязки* следующим образом:

```
g2.setPaint(new TexturePaint(bufferedImage, anchorRectangle));
```

Класс `BufferedImage` будет более подробно рассматриваться чуть позже в этой главе, когда дойдет черед до обсуждения приемов обработки изображений. А до тех пор достаточно сказать, что самый простой способ получить буферизованное изображение в виде объекта типа `BufferedImage` — ввести его из файла, как показано ниже.

```
bufferedImage = ImageIO.read(new File("blue-ball.gif"));
```

Прямоугольник привязки повторяется бесконечное количество раз в направлениях, параллельных осям *x* и *y*, образуя текстуру в виде мозаики. Исходный рисунок масштабируется таким образом, чтобы вписаться в прямоугольник привязки, а затем повторяется в каждом элементе мозаики.

java.awt.Graphics2D 1.2

- `void setPaint(Paint s)`
- Устанавливает в качестве раскраски для данного графического контекста указанный объект, класс которого реализует интерфейс `Paint`.

java.awt.GradientPaint 1.2

- GradientPaint(float *x1*, float *y1*, Color *color1*, float *x2*, float *y2*, Color *color2*)
- GradientPaint(float *x1*, float *y1*, Color *color1*, float *x2*, float *y2*, Color *color2*, boolean *cyclic*)
- GradientPaint(Point2D *p1*, Color *color1*, Point2D *p2*, Color *color2*)
- GradientPaint(Point2D *p1*, Color *color1*, Point2D *p2*, Color *color2*, boolean *cyclic*)

  Создают объект градиентной раскраски. С его помощью фигура заполняется таким образом, что начальная точка с координатами *x1* и *y1* имеет цвет *color1*, конечная точка с координатами *x2* и *y2* — цвет *color2*, а в промежутке между ними цвет изменяется линейно путем интерполяции. Цвета остаются постоянными вдоль прямой, перпендикулярной линии, соединяющей начальную и конечную точки. По умолчанию градиентная раскраска не выполняется циклически. Это означает, что точки, находящиеся за соответствующими пределами градиента, окрашиваются таким же самым цветом, как и начальная и конечная точки градиента. Если же градиентная раскраска выполняется *циклически*, то интерполяция цветов продолжается, возвращаясь сначала к цвету начальной точки, а затем непрерывно повторяясь в обоих направлениях.

java.awt.TexturePaint 1.2

- TexturePaint(BufferedImage *texture*, Rectangle2D *anchor*)

  Создает объект текстурной раскраски. Прямоугольник привязки определяет мозаичное заполнение раскрашиваемого участка. Он повторяется бесконечное число раз в направлении обеих осей координат, *x* и *y*, масштабируя заданное изображение текстуры таким образом, чтобы оно заполняло каждый элемент мозаики.

Преобразование координат

Допустим, требуется нарисовать такой объект, как автомобиль, по известным исходным данным (высоте, расстоянию между осями, общей длине автомашины и в метрах). Конечно, все координаты рисуемых частей автомобиля можно выразить в пикселях, подсчитав количество пикселей на метр. Но все это можно сделать намного проще, запросив соответствующий графический контекст и выполнив в нем преобразование координат следующим образом:

```
g2.scale(pixelsPerMeter, pixelsPerMeter);
g2.draw(new Line2D.Double(координаты_в_метрах)); // преобразовать в
        // пиксели и нарисовать соответственно масштабированную линию
```

Метод `scale()` из класса `Graphics2D` выполняет масштабное *преобразование координат* в заданном графическом контексте. При этом *пользовательские координаты* (единицы измерения, указанные пользователем) преобразуются в *аппаратные координаты* (пиксели). На рис. 7.17 приведен пример такого преобразования.

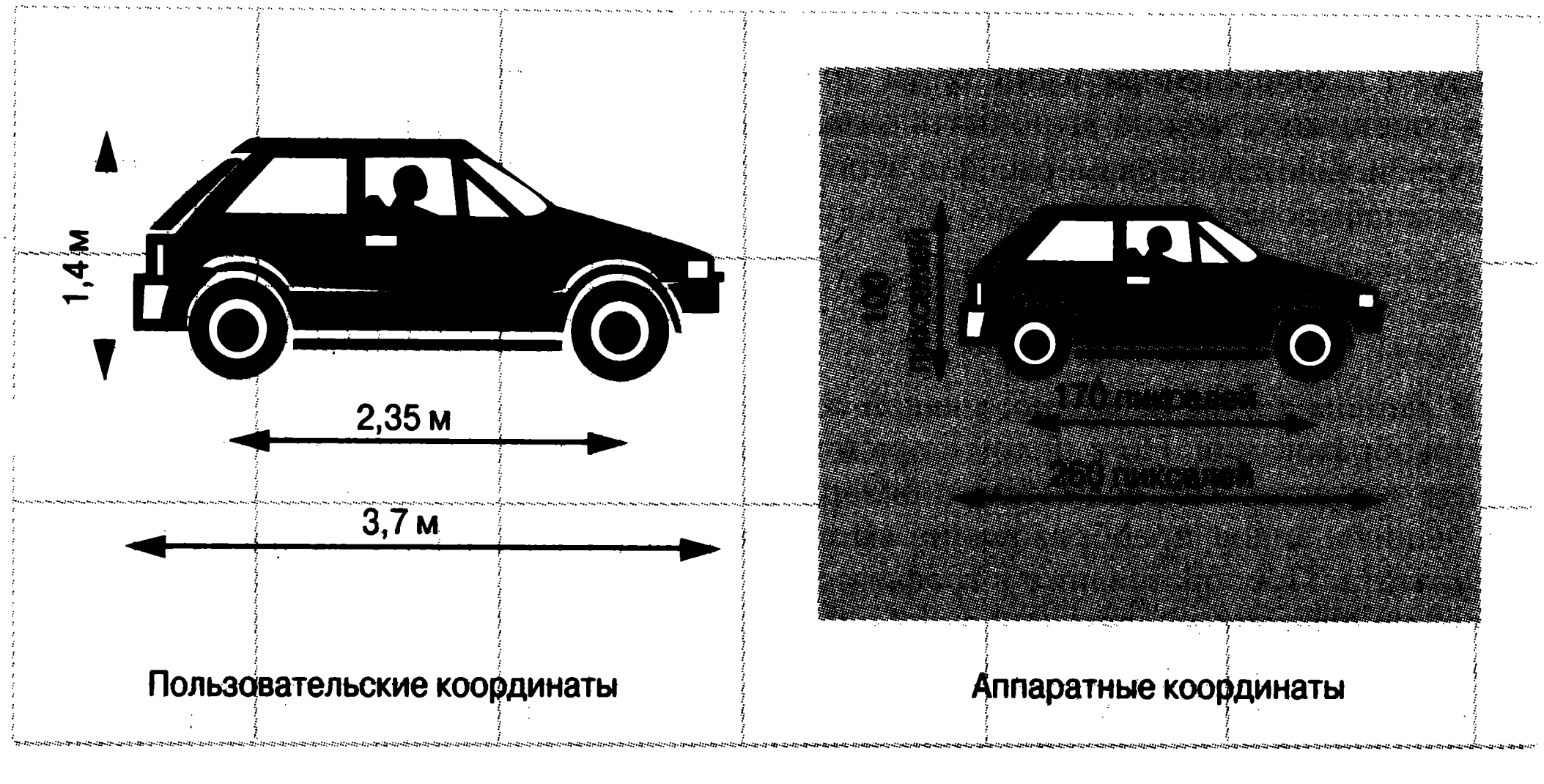


Рис. 7.17. Пользовательские и аппаратные координаты

Преобразование координат очень удобно применять на практике, поскольку оно позволяет оперировать общепринятыми единицами измерения. А все хлопоты по их преобразованию в пиксели берет на себя заданный графический контекст.

В Java предусмотрено четыре основных вида преобразований координат.

- **Масштабирование.** Увеличение или сокращение всех расстояний от фиксированной точки.
- **Вращение.** Поворот всех точек фигуры вокруг фиксированной точки.
- **Перемещение.** Передвижение всех точек фигуры на фиксированное расстояние.
- **Сдвиг.** Фиксация одной линии и перемещение всех параллельных ей линий фигуры на расстояние, пропорциональное расстоянию от данной линии до фиксированной.

На рис. 7.18 показаны результаты перечисленных выше видов преобразования над единичным квадратом.

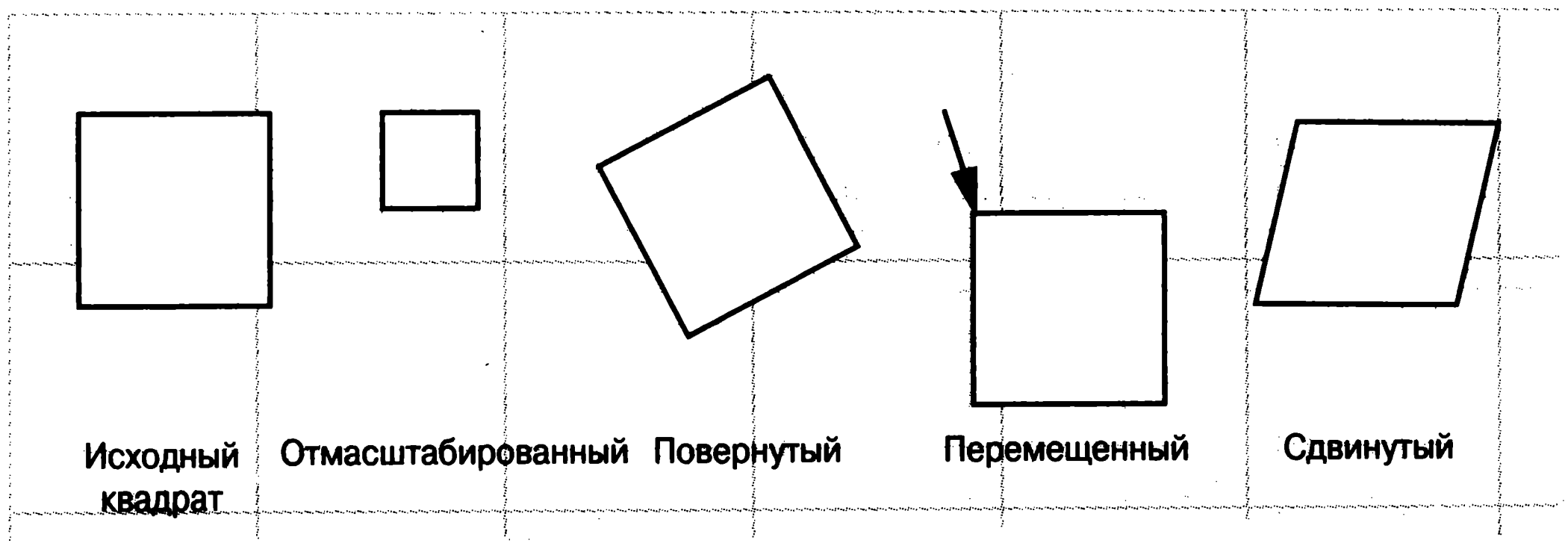


Рис. 7.18. Основные виды преобразования координат

Методы `scale()`, `rotate()`, `translate()` и `shear()` из класса `Graphics2D` реализуют перечисленные выше виды преобразования. Применяя эти методы в разных сочетаниях, можно выполнять *составные* преобразования, например, повернуть фигуру и удвоить ее размеры. Для этого достаточно выполнить операции вращения и масштабирования следующим образом:

```
g2.rotate(angle);
g2.scale(2, 2);
g2.draw(. . .);
```

В данном случае не важно, в какой именно последовательности будут выполняться преобразования. Но в целом порядок выполнения большинства преобразований *имеет* значение. Так, если нужно повернуть и сдвинуть изображение, сначала нужно решить, какое преобразование выполнить первым, чтобы добиться желаемого результата. Преобразования в графическом контексте выполняются в обратном порядке по сравнению с тем, как они были указаны. Это означает, что преобразование, указанное последним, выполняется первым.

Допускается любое количество преобразований координат. Рассмотрим следующую последовательность преобразований:

```
g2.translate(x, y);
g2.rotate(a);
g2.translate(-x, -y);
```

Последнее преобразование (которое выполняется первым) перемещает всю фигуру таким образом, чтобы точка с координатами (*x*, *y*) совпала с точкой начала отсчета. Второе преобразование поворачивает фигуру на угол `a` вокруг точки начала отсчета. А последнее преобразование снова перемещает всю фигуру таким образом, чтобы точка с координатами (*x*, *y*) совпала с точкой начала отсчета. В итоге фигура поворачивается вокруг точки с координатами (*x*, *y*), как показано на рис. 7.19. Вращение фигуры вокруг точки, отличающейся от начала отсчета, выполняется довольно часто, поэтому для данной операции предусмотрен следующий метод:

```
g2.rotate(a, x, y);
```

Как известно из теории матриц, вращение, масштабирование, перемещение, сдвиг и различные их сочетания могут быть выражены в виде матриц преобразования следующим образом:

$$\begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Такие преобразования называются *аффинными*, а в прикладном интерфейсе Java 2D API они описываются с помощью класса `AffineTransform`. Объект аффинного преобразования можно непосредственно создать, если известны компоненты матрицы преобразования, как показано ниже.

```
AffineTransform t = new AffineTransform(a, b, c, d, e, f);
```

Для реализации отдельных типов преобразований предусмотрены также фабричные методы `getRotateInstance()`, `getScaleInstance()`, `getTranslateInstance()` и `getShearInstance()`, назначение которых можно легко определить по их названию. Например, при вызове метода

```
t = AffineTransform.getScaleInstance(2.0F, 0.5F);
```

возвращается преобразование, которое соответствует такой матрице:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

И наконец, методы экземпляра `setToRotate()`, `setToScale()`, `setToTranslation()` и `setToShear()` служат для указания нового вида преобразования. Ниже приведен характерный тому пример.

```
t.setToRotation(angle); // установить вращение на заданный угол
```

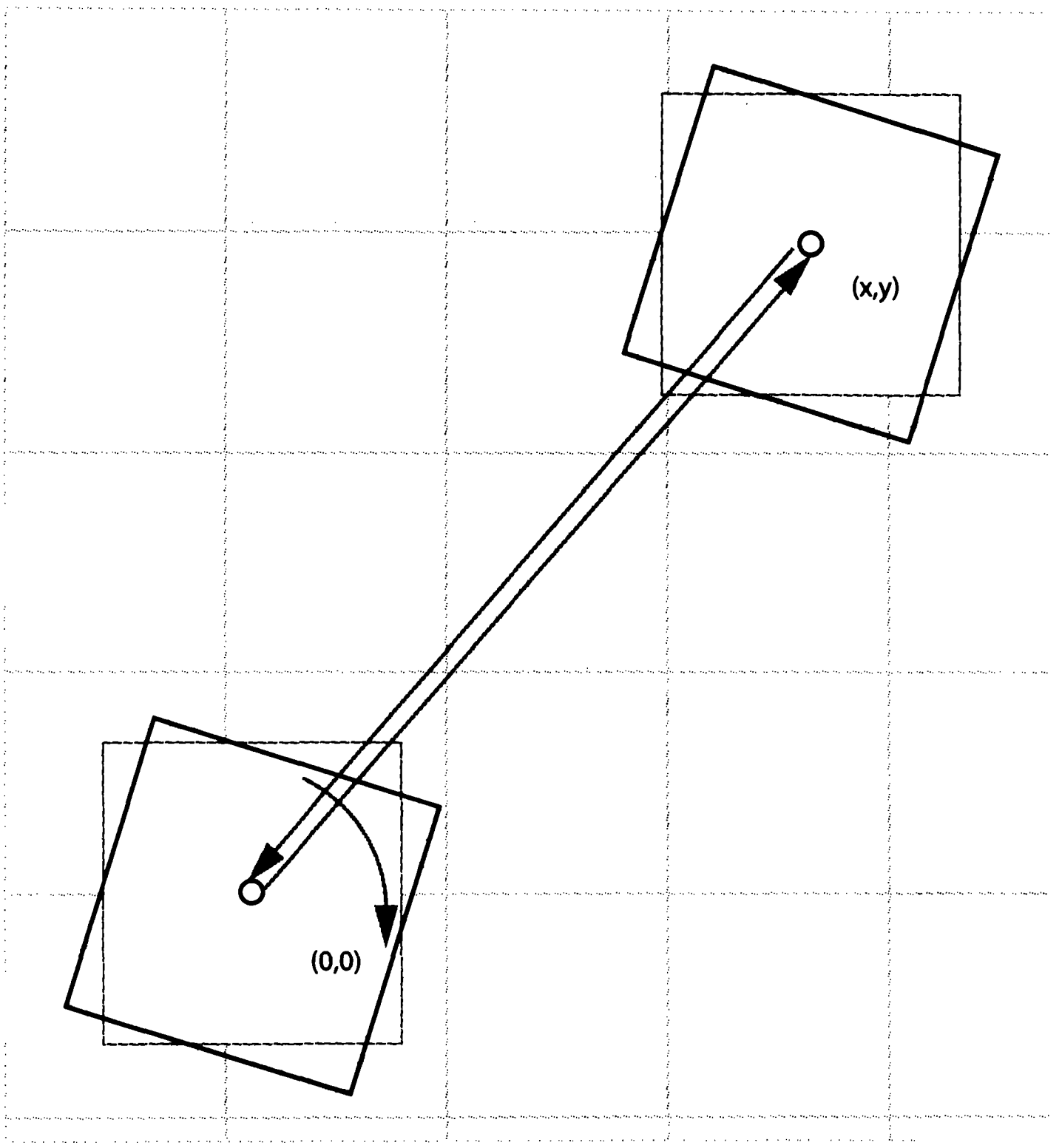


Рис. 7.19. Составные преобразования

Для замены текущего преобразования координат в графическом контексте аффинным преобразованием, представленным объектом типа `AffineTransform`, служит метод

```
g2.setTransform(t); // заменить текущее преобразование
```

Но на практике вызывать метод `setTransform()` все же не рекомендуется, поскольку он заменяет любое преобразование, которое может присутствовать в графическом контексте. Например, графический контекст для печати страницы с альбомной ориентацией уже содержит преобразование вращением на 90°. А при вызове метода `setTransform()` это преобразование игнорируется. В таком случае вместо

метода setTransform() рекомендуется вызвать метод transform(), как показано ниже. Этот метод объединяет текущее преобразование с новым объектом типа AffineTransform, представляющим аффинное преобразование.

```
g2.transform(t); // составить текущее преобразование вместе с афинным
```

Если же требуется выполнить какое-нибудь преобразование временно, то сначала следует сохранить предыдущее преобразование, затем составить вместе с ним новое преобразование и, наконец, восстановить прежнее преобразование, как показано в приведенном ниже фрагменте кода.

```
AffineTransform oldTransform = g2.getTransform(); // сохранить
                                                  // прежнее преобразование
g2.transform(t); // выполнить временное преобразование
рисовать в графическом контексте g2
g2.setTransform(oldTransform); // восстановить прежнее преобразование
```

java.awt.geom.AffineTransform 1.2

- AffineTransform(double *a*, double *b*, double *c*, double *d*, double *e*, double *f*)
- AffineTransform(float *a*, float *b*, float *c*, float *d*, float *e*, float *f*)

  Конструируют аффинное преобразование по приведенной ниже матрице.

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

- AffineTransform(double[] *m*)
- AffineTransform(float[] *m*)

  Конструируют аффинное преобразование по приведенной ниже матрице.

$$\begin{bmatrix} m[0] & m[2] & m[4] \\ m[1] & m[3] & m[5] \\ 0 & 0 & 1 \end{bmatrix}$$

- static AffineTransform getRotateInstance(double *a*)

  Задает преобразование вращением против часовой стрелки вокруг точки начала отсчета на угол *a*, указываемый в радианах. Ниже приведена матрица этого преобразования.

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  Если значение параметра *a* находится в пределах от **0** до **π/2**, то вращение происходит в направлении от положительной оси *x* до положительной оси *y*.

- `static AffineTransform getRotateInstance(double a, double x, double y)`

  Задает преобразование вращением против часовой стрелки вокруг точки с координатами (*x*, *y*) на угол *a*, указываемый в радианах.

- `static AffineTransform getScaleInstance(double sx, double sy)`

  Задает преобразование масштабированием с масштабными коэффициентами *sx* и *sy* по осям *x* и *y* соответственно Ниже приведена матрица этого преобразования.

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- `static AffineTransform getShearInstance (double shx, double shy)`

  Задает преобразование сдвигом с коэффициентами *shx* и *shy* по осям *x* и *y*. Ниже приведена матрица этого преобразования.

$$\begin{bmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- `static AffineTransform getTranslateInstance(double tx, double ty)`

  Задает преобразование перемещением на расстояния *tx* и *ty* по осям *x* и *y*. Ниже приведена матрица этого преобразования.

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

- `void setToRotation(double a)`
- `void setToRotation(double a, double x, double y)`
- `void setToScale(double sx, double sy)`
- `void setToShear(double sx, double sy)`
- `void setToTranslation(double tx, double ty)`

  Устанавливают аффинное преобразование в соответствие с указанными параметрами. Эти параметры интерпретируются таким же образом, как и в упомянутых выше методах типа `getXxxInstance()` для основных преобразований.

java.awt.Graphics2D 1.2

- `void setTransform(AffineTransform t)`

  Заменяет существующее преобразование координат в данном графическом контексте аффинным преобразованием `t`.

- `void transform(AffineTransform t)`

  Составляет преобразование координат, существующее в данном графическом контексте, вместе с аффинным преобразованием *t*.

- `void rotate(double a)`
- `void rotate(double a, double x, double y)`
- `void scale(double sx, double sy)`
- `void shear(double sx, double sy)`
- `void translate(double tx, double ty)`

  Составляют преобразование координат, существующее в данном графическом контексте, вместе с основным преобразованием по указанным параметрам. Эти параметры интерпретируются таким же образом, как и в упомянутых выше методах типа `getXxxInstance()` для основных преобразований.

Отсечение

Для выполнения всех графических операций только внутри ограниченного участка в графическом контексте предусмотрена *фигура отсечения*:

```
g2.setClip(clipShape); // лучше воспользоваться методом, рассматриваемым ниже
g2.draw(shape); // рисовать только внутри фигуры отсечения
```

Но на практике вызывать метод `setClip()` не рекомендуется, поскольку он заменяет все существующие фигуры отсечения в данном графическом контексте. Как будет показано далее, графический контекст для печати уже содержит прямоугольник отсечения, который позволяет избежать появления данных на полях страницы. В этом случае вместо метода `setClip()` лучше вызвать метод `clip()` следующим образом:

```
g2.clip(clipShape); // лучше вызвать этот метод
```

Метод `clip()` образует пересечение существующей фигуры отсечения с указанной фигурой. Если же фигуру отсечения требуется применить лишь временно, то сначала следует сохранить прежнюю фигуру, затем добавить новую и, наконец, восстановить прежнюю. Ниже приведен характерный тому пример.

```
Shape oldClip = g2.getClip(); // сохранить прежнее отсечение
g2.clip(clipShape); // осуществить временное отсечение
рисовать в графическом контексте g2
g2.setClip(oldClip); // восстановить прежнее отсечение
```

На рис. 7.20 возможности отсечения демонстрируются на примере рисования довольно непростого штрихового рисунка, который отсекается сложной фигурой, а именно контуром ряда символов.

Для получения контуров символов требуется *контекст воспроизведения шрифтов*. С этой целью сначала вызывается метод `getFontRenderContext()` из класса `Graphics2D`:

```
FontRenderContext context = g2.getFontRenderContext();
```

Затем создается объект расположения текста типа `TextLayout`, для чего используется символьная строка, шрифт и контекст воспроизведения шрифтов:

```
TextLayout layout = new TextLayout("Hello", font, context);
```

Рис. 7.20. Отсечение штрихового рисунка фигурами букв

Объект расположения текста описывает последовательность символов, расположение и оформление которых определяется выбранным контекстом воспроизведения шрифтов. Как известно, одни и те же символы могут по-разному выглядеть на экране и на печатной странице.

Но важнее другое: метод `getOutline()` возвращает объект типа `Shape`, описывающий фигуру контура символов в расположении текста. Эта фигура начинается в точке начала отсчета с координатами (**0**, **0**). Но если такое расположение не подходит, то при вызове метода `getOutline()` задается аффинное преобразование, позволяющее указать, в какой именно точке должен появиться контур:

```
AffineTransform transform = AffineTransform.getTranslateInstance(0, 100);
Shape outline = layout.getOutline(transform);
```

А затем контур присоединяется к фигуре отсечения следующим образом:

```
GeneralPath clipShape = new GeneralPath();
clipShape.append(outline, false);
```

И наконец, устанавливается фигура отсечения и рисуются линии штриховки, как показано ниже. В итоге линии появляются только внутри символов.

```
g2.setClip(clipShape);
Point2D p = new Point2D.Double(0, 0);
for (int i = 0; i < NLINES; i++)
{
  double x = . . .;
  double y = . . .;
  Point2D q = new Point2D.Double(x, y);
  g2.draw(new Line2D.Double(p, q)); // отсечь линии штриховки
}
```

java.awt.Graphics 1.0

- `void setClip(Shape s)` **1.2**

  Задает фигуру *s* в качестве текущей фигуры отсечения.

- `Shape getClip()` **1.2**

  Возвращает текущую фигуру отсечения.

java.awt.Graphics2D 1.2

- `void clip(Shape s)`

  Образует пересечение текущей фигуры отсечения с фигурой *s*.

- `FontRenderContext getFontRenderContext()`

  Возвращает контекст воспроизведения шрифтов, требующийся для создания объекта типа `TextLayout`.

java.awt.font.TextLayout 1.2

- `TextLayout(String s, Font f, FontRenderContext context)`

  Создает объект `TextLayout`, исходя из заданной символьной строки, шрифта и контекста воспроизведения шрифтов.

- `float getAdvance()`

  Возвращает ширину данного расположения текста.

- `float getAscent()`
- `float getDescent()`

  Возвращают высоту данного расположения текста относительно базовой линии.

- `float getLeading()`

  Возвращает расстояние между соседними строками для шрифта, применяемого в данном расположении текста.

Прозрачность и композиция

В стандартной цветовой модели RGB каждый цвет описывается по трем его основным составляющим: красной, зеленой и синей. Но иногда отдельные участки изображения удобно сделать *прозрачными* или частично прозрачными. При наложении рисунка на уже существующее изображение прозрачные пиксели не закрывают находящиеся под ними пиксели, а частично прозрачные пиксели смешиваются с нижележащими пикселями. На рис. 7.21 приведен результат наложения частично прозрачного прямоугольника на уже имеющееся изображение. Обратите внимание на то, что детали изображения, находящиеся под прямоугольником, по-прежнему видны.

Рис. 7.21. Наложение частично прозрачного прямоугольника на изображение

В прикладном интерфейсе Java 2D API прозрачность описывается с помощью *альфа-канала*. Каждый пиксель, кроме красной, зеленой и синей составляющей цвета, имеет значение прозрачности в альфа-канале, изменяющееся в предела от **0** (полностью прозрачен) до **1** (совершенно непрозрачен). Например, прямоугольник на рис. 7.21 заполнен бледно-желтым цветом с прозрачностью 50% следующим образом:

```
new Color(0.7F, 0.7F, 0.0F, 0.5F);
```

А теперь рассмотрим пример наложения двух фигур. Для этого требуется смешать или составить цвета и значения прозрачности в альфа-канале исходных и целевых пикселей. Исследователи Портер (Porter) и Дафф (Duff), работающие в области компьютерной графики, сформулировали двенадцать возможных *правил композиции*, которые реализованы в Java 2D API. Следует отметить, что только два из этих правил имеют практическое применение. Если эти правила покажутся слишком сложными, то вместо них рекомендуется использовать простое правило `SRC_OVER`, которое применяется по умолчанию для объектов типа `Graphics2D` и дает интуитивно понятные результаты.

Рассмотрим вкратце теоретические основы правил композиции. Допустим, *исходный пиксель* имеет значение прозрачности в альфа-канале a_S, а *целевой пиксель* — значение прозрачности a_D. На рис. 7.22 показана диаграмма, схематически поясняющая правила композиции для этих значений.

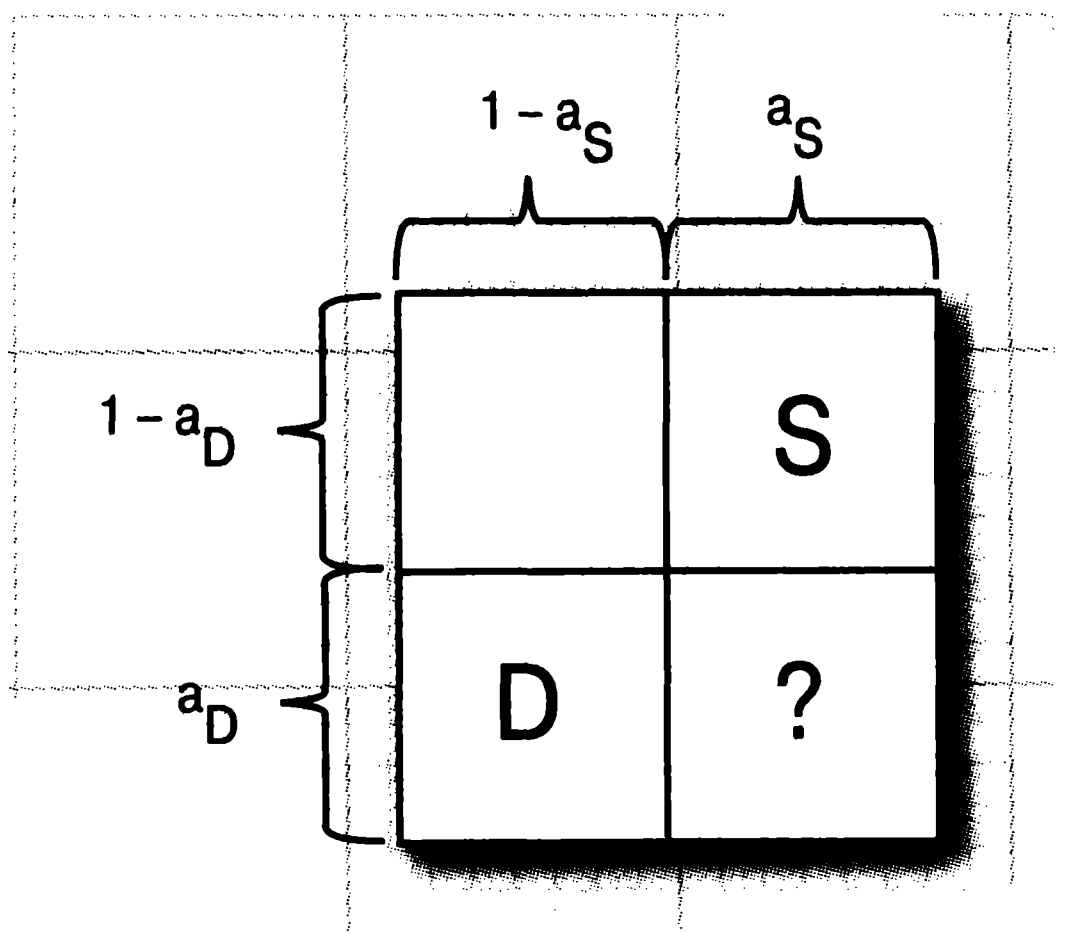


Рис. 7.22. Схематическое представление составляемого правила композиции

Портер и Дафф считают, что значение прозрачности в альфа-канале выражает вероятность использования цвета пикселя при объединении изображений. Для исходного пикселя исходный цвет будет использоваться с вероятностью a_S и не использоваться с вероятностью $1 - a_S$. Это же справедливо и для целевого пикселя. Допустим, при составлении цветов эти вероятности независимы. На рис. 7.22 показаны четыре возможных ситуации. Если нужно преимущественно использовать цвет исходного пикселя, а не целевого (эта ситуация обозначается буквой **S**), то вероятность такого события равна $a_S \cdot (1 - a_D)$. Аналогично вычисляется вероятность $a_D \cdot (1 - a_S)$ преимущественного использования цвета целевого, а не исходного пикселя (эта ситуация обозначается

буквой **D**). Что же делать, если исходное и целевое изображения "считают" свой цвет преимущественным? Именно в этой ситуации используются правила композиции Портера–Даффа. Если предпочтение отдается исходному цвету, в таком случае правый нижний угол диаграммы помечается буквой **S**, а само правило композиции называется `SRC_OVER`. По этому правилу исходный и целевой цвета составляются с весом a_S и $a_D \cdot (1 - a_S)$ соответственно.

Визуальный эффект применения этого правила композиции состоит в том, что при смешении цветов исходного и целевого пикселей приоритет отдается цвету исходного пикселя. В частности, если $a_S = 1$, то цвет целевого пикселя вообще не принимается во внимание. А если $a_S = 0$, то исходный пиксель становится совершенно прозрачным, тогда как цвет целевого пикселя не изменяется.

В зависимости от того, как расставлены буквы на диаграмме, можно сформировать другие правила композиции. Они перечислены в табл. 7.1 и схематически показаны на рис. 7.23. Изображения на этом рисунке представляют результаты применения правил композиции к прямоугольному участку исходного изображения со значением прозрачности **0,75** в альфа-канале и эллиптическому участку целевого изображения со значением прозрачности **1,0** в альфа-канале.

Как можно заметить, большинство этих правил вряд ли применяются на практике. Так, например, правило композиции `DST_IN` представляет собой крайний случай, когда во внимание совсем не принимается цвет исходного пикселя, тогда как его альфа-канал используется для изменения целевого пикселя. В отличие от него, правило композиции `SRC` может быть очень полезно, потому что оно предписывает использовать исходный цвет, исключая смешение с целевым цветом. Более подробно о правилах Портера–Даффа можно прочитать в упоминавшейся ранее книге *Computer Graphics: Principles and Practice, Second Edititon in C* Джеймса Фоли, Андриса ван Дама, Стивена Фэйнера и др.

Для создания объекта правила композиции, класс которого реализует интерфейс `Composite`, служит метод `setComposite()` из класса `Graphics2D`. В состав прикладного интерфейса Java 2D API входит только один такой класс — `AlphaComposite`. В этом классе реализованы все правила Портера–Даффа, представленные на рис. 7.23.

Для создания экземпляра правила типа `AlphaComposite` служит фабричный метод `getInstance()` из класса `AlphaComposite`. В качестве параметров этого метода указываются правило композиции и значение прозрачности в альфа-канале для пикселей исходного изображения, как показано в приведенном ниже фрагменте кода.

```
int rule = AlphaComposite.SRC_OVER;
float alpha = 0.5f;
g2.setComposite(AlphaComposite.getInstance(rule, alpha));
g2.setPaint(Color.blue);
g2.fill(rectangle);
```

В этом фрагменте кода прямоугольник заполняется синим цветом со значением прозрачности **0,5** в альфа-канале. В соответствии с заданным правилом композиции `SRC_OVER` этот прямоугольник прозрачно накладывается на уже существующее изображение.

В примере программы, исходный код которой приведен в листинге 7.3, предоставляется возможность исследовать правила композиции. Для этого достаточно выбрать конкретное правило из комбинированного списка и установить ползунковым регулятором значение прозрачности в альфа-канале для объекта типа `AlphaComposite`.

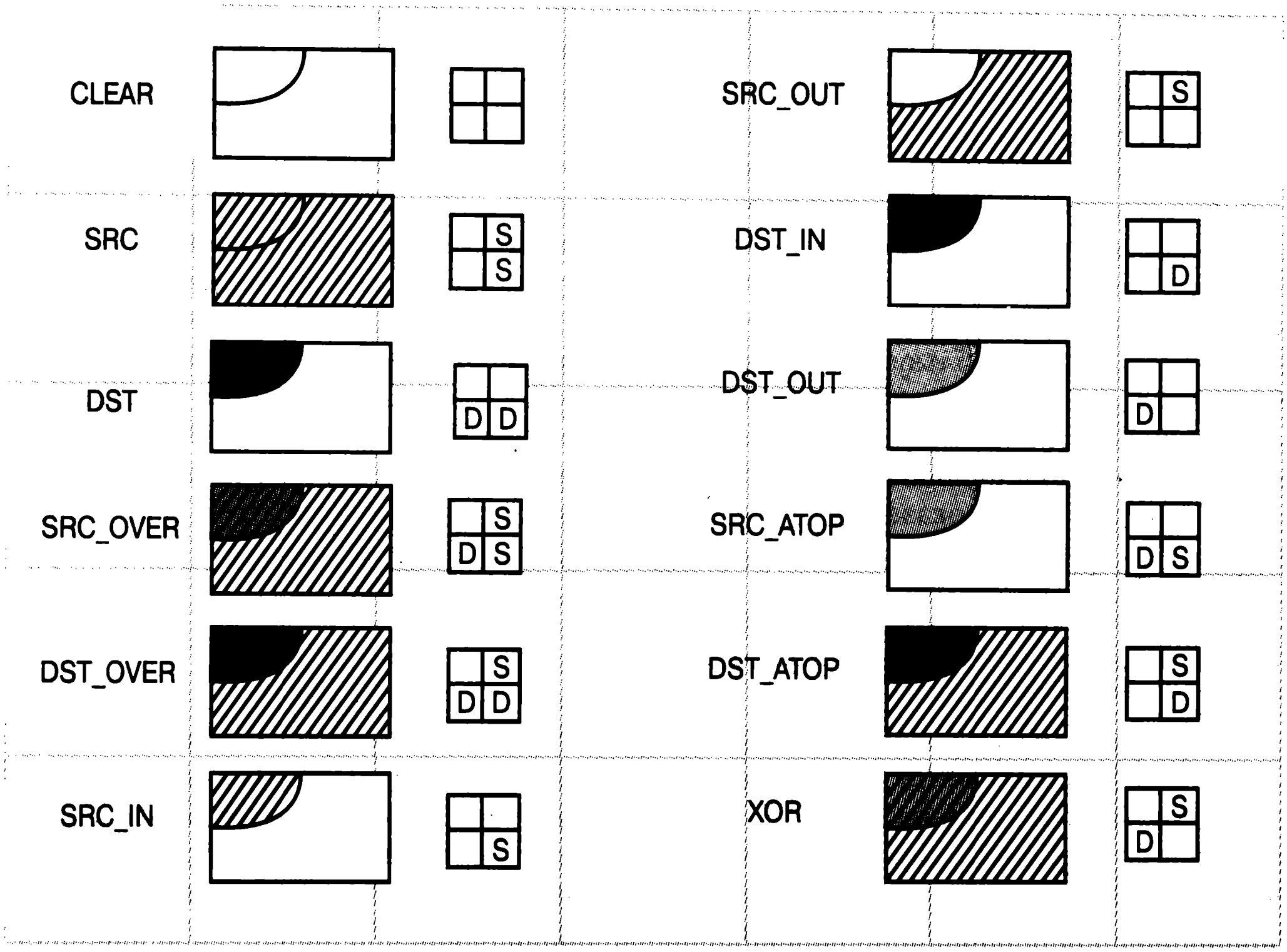


Рис. 7.23. Правила композиции Портера–Даффа

Таблица 7.1. Правила композиции Портера–Даффа

Правило	Описание
CLEAR	Исходные пиксели очищают целевые
SRC	Исходные пиксели перезаписывают целевые и пустые пиксели
DST	Исходные пиксели не оказывают никакого влияния на целевые пиксели
SRC_OVER	Исходные пиксели смешиваются с целевыми и перезаписывают пустые пиксели
DST_OVER	Исходные пиксели не оказывают никакого влияния на целевые пиксели и перезаписывают пустые пиксели
SRC_IN	Исходные пиксели перезаписывают целевые пиксели
SRC_OUT	Исходные пиксели очищают целевые пиксели и перезаписывают пустые пиксели
DST_IN	Прозрачность в альфа-канале исходного изображения видоизменяет целевое изображение
DST_OUT	Дополнение до прозрачности в альфа-канале исходного изображения видоизменяет целевое изображение
SRC_ATOP	Исходные пиксели смешиваются с целевыми
DST_ATOP	Прозрачность в альфа-канале исходного изображения видоизменяет целевое изображение. А исходные пиксели перезаписывают пустые пиксели
XOR	Дополнение до прозрачности в альфа-канале исходного изображения видоизменяет целевое изображение. А исходные пиксели перезаписывают пустые пиксели

В нижней части рабочего окна данной программы приводится краткое описание каждого правила композиции. Обратите внимание на то, что это описание автоматически составляется по диаграммам правил композиции. Например, строка **"DS"** во втором ряду диаграммы приводит к появлению в описании правила композиции строки "blends with destination" (смешивается с целевым изображением).

Но эта программа не гарантирует, что используемый графический контекст, соответствующий экрану, имеет альфа-канал. Когда пиксели накладываются на целевое изображение без альфа-канала, цвета пикселей умножаются на значение прозрачности в альфа-канале, а затем это значение отбрасывается. В ряде правил композиции Портера–Даффа используются значения прозрачности в альфа-канале целевого изображения, и это указывает на важную роль альфа-канала. Поэтому для составления изображений (в данном случае фигур) используется буферизованное изображение с цветовой моделью ARGB. После составления результирующее изображение выводится на экран, как показано в приведенном ниже фрагменте кода.

```
BufferedImage image = new BufferedImage(getWidth(), getHeight(),
BufferedImage.TYPE_INT_ARGB);
Graphics2D gImage = image.createGraphics();
// а теперь рисовать на изображении gImage
g2.drawImage(image, null, 0, 0);
```

В листингах 7.3 и 7.4 представлен исходный код классов фрейма и компонента. А класс `Rule` из листинга 7.5 предоставляет краткое описание каждого правила композиции, как показано на рис. 7.24. После запуска программы переместите ползунок регулятора слева направо, чтобы посмотреть результат применения выбранного правила композиции для наложения фигур при разных значениях прозрачности в альфа-канале. Обратите особое внимание на то, что правила `DST_IN` и `DST_OUT` отличаются лишь направлением изменения цвета целевого изображения (!) при изменении значения прозрачности в альфа-канале исходного изображения.

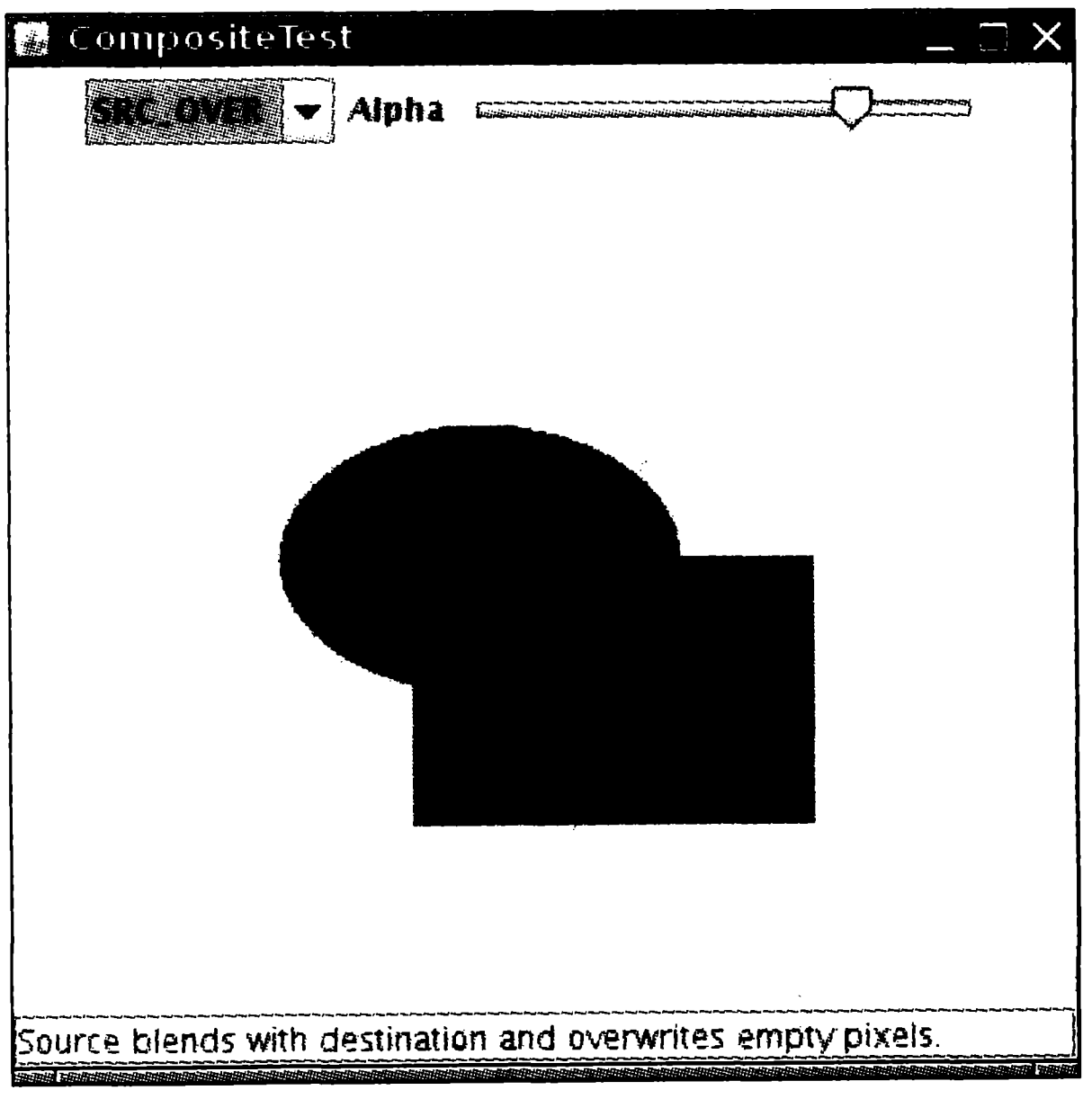


Рис. 7.24. Рабочее окно программы `CompositeTest`

Листинг 7.3. Исходный код из файла `composite/CompositeTestFrame.java`

```
1  package composite;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import javax.swing.event.*;
7
8  /**
9   * Этот фрейм содержит комбинированный список для выбора правил,
10  * композиции, ползунок для изменения прозрачности в альфа-канале
11  * исходного изображения, а также компонент для отображения
12  * результатов составления изображений
13 */
14 class CompositeTestFrame extends JFrame
15 {
16   private static final int DEFAULT_WIDTH = 400;
17   private static final int DEFAULT_HEIGHT = 400;
18
19   private CompositeComponent canvas;
20   private JComboBox<Rule> ruleCombo;
21   private JSlider alphaSlider;
22   private JTextField explanation;
23
24   public CompositeTestFrame()
25   {
26     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
27
28     canvas = new CompositeComponent();
29     add(canvas, BorderLayout.CENTER);
30
31     ruleCombo = new JComboBox<>(new Rule[]
32        { new Rule("CLEAR", " ", " "),
33          new Rule("SRC", " S", " S"),
34          new Rule("DST", " ", "DD"),
35          new Rule("SRC_OVER", " S", "DS"),
36          new Rule("DST_OVER", " S", "DD"),
37          new Rule("SRC_IN", " ", " S"),
38          new Rule("SRC_OUT", " S", " "),
39          new Rule("DST_IN", " ", " D"),
40          new Rule("DST_OUT", " ", "D "),
41          new Rule("SRC_ATOP", " ", "DS"),
42          new Rule("DST_ATOP", " S", " D"),
43          new Rule("XOR", " S", "D "), });
44     ruleCombo.addActionListener(new ActionListener()
45       {
46         public void actionPerformed(ActionEvent event)
47         {
48           Rule r = (Rule) ruleCombo.getSelectedItem();
49           canvas.setRule(r.getValue());
50           explanation.setText(r.getExplanation());
51         }
52       });
53
54     alphaSlider = new JSlider(0, 100, 75);
55     alphaSlider.addChangeListener(new ChangeListener()
56       {
57         public void stateChanged(ChangeEvent event)
```

```
58          {
59             canvas.setAlpha(alphaSlider.getValue());
60          }
61       });
62     JPanel panel = new JPanel();
63     panel.add(ruleCombo);
64     panel.add(new JLabel("Alpha"));
65     panel.add(alphaSlider);
66     add(panel, BorderLayout.NORTH);
67
68     explanation = new JTextField();
69     add(explanation, BorderLayout.SOUTH);
70
71     canvas.setAlpha(alphaSlider.getValue());
72     Rule r = ruleCombo.getItemAt(ruleCombo.getSelectedIndex());
73     canvas.setRule(r.getValue());
74     explanation.setText(r.getExplanation());
75   }
76 }
```

Листинг 7.4. Исходный код из файла `composite/CompositeComponent.java`

```
1  package composite;
2
3  import java.awt.*;
4  import java.awt.geom.*;
5  import java.awt.image.*;
6  import javax.swing.*;
7
8  /**
9   * Этот компонент рисует две формы, составленные по правилу композиции
10 */
11 class CompositeComponent extends JComponent
12 {
13   private int rule;
14   private Shape shape1;
15   private Shape shape2;
16   private float alpha;
17
18   public CompositeComponent()
19   {
20     shape1 = new Ellipse2D.Double(100, 100, 150, 100);
21     shape2 = new Rectangle2D.Double(150, 150, 150, 100);
22   }
23
24   public void paintComponent(Graphics g)
25   {
26     Graphics2D g2 = (Graphics2D) g;
27
28     BufferedImage image = new BufferedImage(getWidth(), getHeight(),
29        BufferedImage.TYPE_INT_ARGB);
30     Graphics2D gImage = image.createGraphics();
31     gImage.setPaint(Color.red);
32     gImage.fill(shape1);
33     AlphaComposite composite = AlphaComposite.getInstance(rule, alpha);
34     gImage.setComposite(composite);
35     gImage.setPaint(Color.blue);
36     gImage.fill(shape2);
```

```
37      g2.drawImage(image, null, 0, 0);
38    }
39
40    /**
40     * Устанавливает правило композиции
41     * @param r Правило композиции (в виде константы
42     *          из класса AlphaComposite)
43    */
44    public void setRule(int r)
45    {
46      rule = r;
47      repaint();
48    }
49
50    /**
51     * Устанавливает значение прозрачности в
52     * альфа-канале исходного изображения
53     * @param a Значение прозрачности в пределах от 0 до 100
54    */
55    public void setAlpha(int a)
56    {
57      alpha = (float) a / 100.0F;
58      repaint();
59    }
60  }
```

Листинг 7.5. Исходный код из файла composite/Rule.java

```
1   package composite;
2
3   import java.awt.*;
4
5   /**
6    * Этот класс описывает правило композиции Портера-Даффа
7   */
8   class Rule
9   {
10    private String name;
11    private String porterDuff1;
12    private String porterDuff2;
13
14    /**
15     * Составляет правило композиции Портера-Даффа
16     * @param n Название правила композиции
17     * @param pd1 Первый ряд квадратов в правиле композиции Портера-Дафа
18     * @param pd2 Второй ряд в правиле композиции Портера-Дафа
19    */
20    public Rule(String n, String pd1, String pd2)
21    {
22      name = n;
23      porterDuff1 = pd1;
24      porterDuff2 = pd2;
25    }
26
27    /**
28     * Получает объяснение композиции по данному правилу
29     * @return Возвращает объяснение композиции по данному правилу
30    */
```

```
31   public String getExplanation()
32   {
33     StringBuilder r = new StringBuilder("Source ");
34     if (porterDuff2.equals(" ")) r.append("clears");
35     if (porterDuff2.equals(" S")) r.append("overwrites");
36     if (porterDuff2.equals("DS")) r.append("blends with");
37     if (porterDuff2.equals(" D")) r.append("alpha modifies");
38     if (porterDuff2.equals("D ")) r.append("alpha complement modifies");
39     if (porterDuff2.equals("DD")) r.append("does not affect");
40     r.append(" destination");
41     if (porterDuff1.equals(" S")) r.append(" and overwrites empty pixels");
42     r.append(".");
43     return r.toString();
44   }
45
46   public String toString()
47   {
48     return name;
49   }
50
51   /**
52    * Получает значение по данному правилу в классе AlphaComposite
53    * @return Возвращает значение константы из класса AlphaComposite
54    * или значение -1, если соответствующая константа отсутствует
55   */
56   public int getValue()
57   {
58     try
59     {
60       return (Integer) AlphaComposite.class.getField(name).get(null);
61     }
62     catch (Exception e)
63     {
64       return -1;
65     }
66   }
67 }
```

java.awt.Graphics2D 1.2

- `void setComposite(Composite s)`

  Устанавливает указанный объект, класс которого реализует интерфейс `Composite`, в качестве правила композиции для данного графического контекста.

java.awt.AlphaComposite 1.2

- `static AlphaComposite getInstance(int rule)`
- `static AlphaComposite getInstance(int rule, float sourceAlpha)`

  Создают объект композиции на основе заданного правила и значения в альфа-канале. Параметр *rule*, задающий правило композиции, может принимать одно из следующих значений: `CLEAR`, `DST`, `SRC`, `SRC_OVER`, `DST_OVER`, `SRC_ATOP`, `SRC_IN`, `SRC_OUT`, `DST_ATOP`, `DST_IN`, `DST_OUT`, `XOR`.

Указания по воспроизведению

Как следует из предыдущих разделов, процесс воспроизведения графики довольно сложный. И хотя прикладной интерфейс Java 2D API чаще всего действует необычайно быстро, иногда возникает потребность в контроле над достижением компромисса между быстродействием и качеством. Получить такой контроль можно, задав так называемые указания по воспроизведению. В частности, метод `setRenderingHint()` из класса `Graphics2D` позволяет задать лишь одно такое указание. А ключи и значения указаний по воспроизведению объявляются в классе `RenderingHints`. В табл. 7.2 вкратце перечислены возможные варианты выбора ключей и значений. Значения, оканчивающиеся на `_DEFAULT`, обозначают выбираемые по умолчанию варианты, обеспечивающие в конкретной реализации достижение наилучшего компромисса между производительностью и качеством.

Таблица 7.2. Указания по воспроизведению

Ключ	Значение	Описание
KEY_ANTIALIASING	VALUE_ANTIALIAS_ON VALUE_ANTIALIAS_OFF VALUE_ANTIALIAS_DEFAULT	Включение/отключение сглаживания фигур
KEY_TEXT_ANTIALIASING	**VALUE_TEXT_ANTIALIAS_ON** **VALUE_TEXT_ANTIALIAS_OFF** **VALUE_TEXT_ANTIALIAS_DEFAULT** **VALUE_TEXT_ANTIALIAS_GASP 6** **VALUE_TEXT_ANTIALIAS_LCD_HRGB 6** **VALUE_TEXT_ANTIALIAS_LCD_HBGR 6** **VALUE_TEXT_ANTIALIAS_LCD_VRGB 6** **VALUE_TEXT_ANTIALIAS_LCD_VBGR 6**	Включение/отключение сглаживания шрифтов. Значение **VALUE_TEXT_ANTIALIAS_GASP** подразумевает просмотр gasp-таблицы шрифта, чтобы решить, следует ли сглаживать шрифт конкретного размера. А значения с префиксом **LCD** вызывают принудительное воспроизведение подпикселей на устройствах отображения конкретного типа (в данном случае — жидкокристаллических)
KEY_FRACTIONALMETRICS	**VALUE_FRACTIONALMETRICS_ON** **VALUE_FRACTIONALMETRICS_OFF** **VALUE_FRACTIONALMETRICS_DEFAULT**	Включение/отключение вычисления дробных размеров шрифтов. Дробные размеры шрифтов позволяют улучшить расположение выделяемых ими символов
KEY_RENDERING	**VALUE_RENDER_QUALITY** **VALUE_RENDER_SPEED** **VALUE_RENDER_DEFAULT**	Выбор алгоритма рисования для достижения лучшего качества или большей скорости
KEY_STROKE_CONTROL 1.3	**VALUE_STROKE_NORMALIZE** **VALUE_STROKE_PURE** **VALUE_STROKE_DEFAULT**	Выбор между возможностью передать контроль над расположением линий обводки графическому ускорителю (что чревато их смещением на целую половину пикселя) и вычислять их расположение строго по правилу, которое требует, чтобы линии обводки непременно проходили по центру пикселей

Окончание табл. 7.2

Ключ	Значение	Описание
KEY_DITHERING	VALUE_DITHER_ENABLE VALUE_DITHER_DISABLE VALUE_DITHER_DEFAULT	Включение/отключение имитации полутонов. Алгоритм имитации полутонов основан на специальном распределении значений цвета расположенных рядом пикселей. (Следует, однако, иметь в виду, что сглаживание может мешать имитации полутонов.)
KEY_ALPHA_INTERPOLATION	VALUE_ALPHA_INTERPOLATION_QUALITY VALUE_ALPHA_INTERPOLATION_SPEED VALUE_ALPHA_INTERPOLATION_DEFAULT	Включение/отключение точного вычисления при составлении изображений по альфа-каналу
KEY_COLOR_RENDERING	VALUE_COLOR_RENDER_QUALITY VALUE_COLOR_RENDER_SPEED VALUE_COLOR_RENDER_DEFAULT	Выбор между качеством или быстродействием при воспроизведении цветов. Такой выбор приходится делать только в том случае, если используются разные цветовые пространства
KEY_INTERPOLATION	VALUE_INTERPOLATION_NEAREST_NEIGHBOR VALUE_INTERPOLATION_BILINEAR VALUE_INTERPOLATION_BICUBIC	Выбор правила интерполяции пикселей при масштабировании или вращении изображений

Наиболее полезным из указаний по воспроизведению является сглаживание, подразумевающее удаление всевозможных неровностей при рисовании различных линий. Как показано на рис. 7.25, наклонная линия рисуется в виде лестницы из расположенных уступом пикселей. Иногда, особенно на мониторах с низким разрешением, она выглядит совершенно неприемлемо. Но если каждый пиксель рисуемой линии воспроизвести не только двумя контрастными цветами (например, черным и белым), а полутонами, значения которых пропорциональны занимаемой площади пикселя, то такая линия будет выглядеть намного более гладкой. Такой алгоритм называется *сглаживанием*. Безусловно, он требует больше времени, расходуемого на расчет значений всех переходных цветов.

В качестве примера ниже показано, каким образом можно указать в коде на применение сглаживания.

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
```

Сглаживание целесообразно применять и для шрифтов, как показано ниже. Что же касается остальных указаний по воспроизведению, то они применяются реже.

```
g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
    Rendering-Hints.VALUE_TEXT_ANTIALIAS_ON);
```

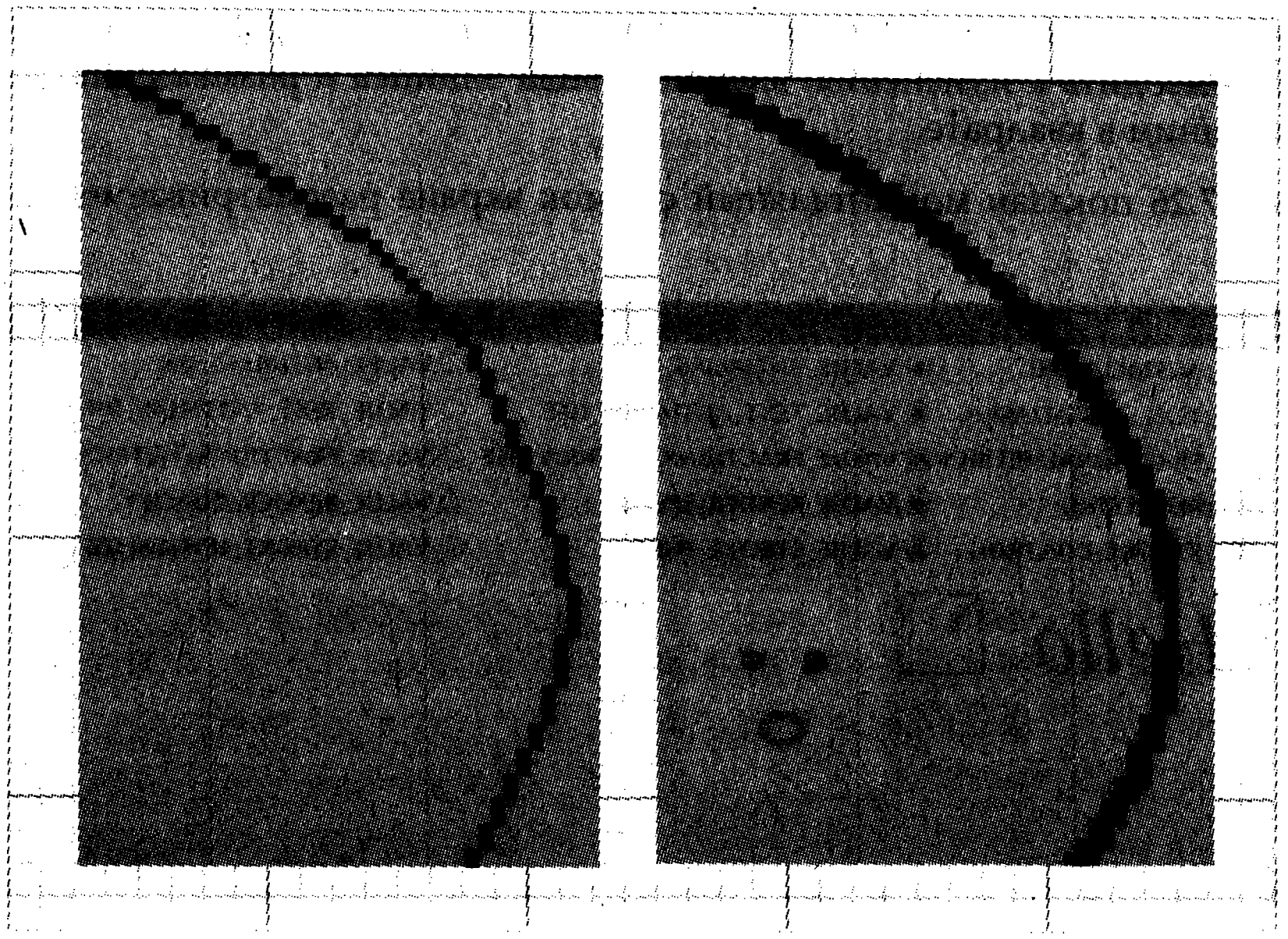


Рис. 7.25. Сглаживание

Пары "ключ–значение" для указаний по воспроизведению можно собрать в единое отображение и задавать их одновременно с помощью метода `setRenderingHints()`. Для этого подойдет любой класс коллекций, реализующий интерфейс `Map`, но может вполне пригодиться и сам класс `RenderingHints`. Ведь он также реализует интерфейс `Map` и формирует отображение по умолчанию, если параметр его конструктора принимает пустое значение `null`, как показано в приведенном ниже фрагменте кода.

```
RenderingHints hints = new RenderingHints(null);
hints.put(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
hints.put(RenderingHints.KEY_TEXT_ANTIALIASING, RenderingHints.VALUE_TEXT_
                                                ANTIALIAS_ON);
g2.setRenderingHints(hints);
```

Именно такой прием применяется в примере программы из листинга 7.6. В этой программе демонстрируется несколько указаний по воспроизведению, которые, на наш взгляд, наиболее полезны. В этой связи необходимо обратить особое внимание на следующее.

- Сглаживание фигуры делает плавной форму эллипса.
- Сглаживание текста делает плавным воспроизводимый текст.
- На некоторых платформах буквы будут располагаться чуть ближе друг к другу из-за дробной системы типографских параметров набора текста.
- При выборе значения **`VALUE_RENDER_QUALITY`** сглаживается масштабируемое изображение. (Такого же эффекта можно было бы добиться, установив значение **`VALUE_INTERPOLATION_BICUBIC`** по ключу **`KEY_INTERPOLATION`**.)

- При отключении сглаживания выбор значения VALUE_STROKE_NORMALIZE будет приводить к изменению внешнего вида эллипса и размещению диагональной линии в квадрате.

На рис. 7.26 показан моментальный снимок экрана рассматриваемой здесь программы.

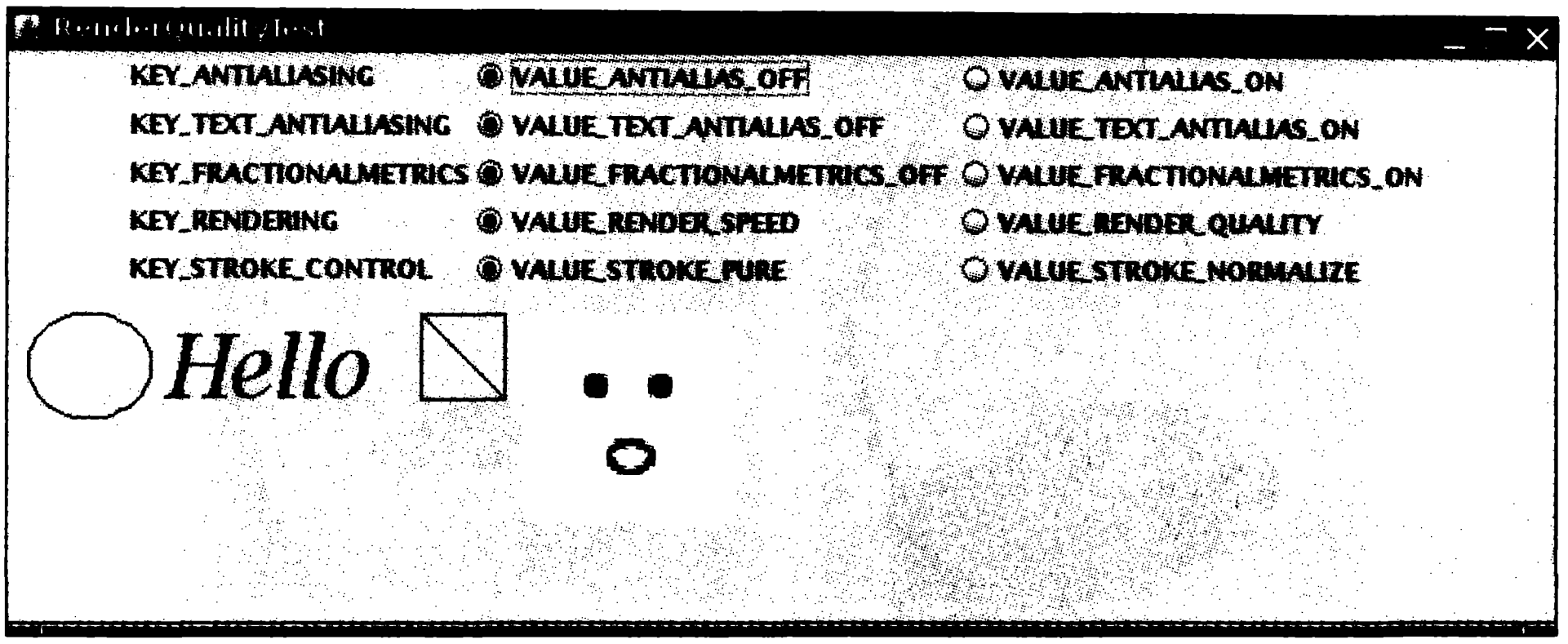


Рис. 7.26. Проверка действия указаний по воспроизведению

Листинг 7.6. Исходный код из файла renderQuality/RenderQualityTestFrame.java

```
1   package renderQuality;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.awt.geom.*;
6   import javax.swing.*;
7
8   /**
9   * Этот фрейм содержит кнопки-переключатели для выбора указаний
10  * по воспроизведению, а также воспроизводимое по ним изображение
11  */
12  public class RenderQualityTestFrame extends JFrame
13  {
14    private RenderQualityComponent canvas;
15    private JPanel buttonBox;
16    private RenderingHints hints;
17    private int r;
18
19    public RenderQualityTestFrame()
20    {
21      buttonBox = new JPanel();
22      buttonBox.setLayout(new GridBagLayout());
23      hints = new RenderingHints(null);
24
25      makeButtons("KEY_ANTIALIASING", "VALUE_ANTIALIAS_OFF",
26         "VALUE_ANTIALIAS_ON");
27      makeButtons("KEY_TEXT_ANTIALIASING", "VALUE_TEXT_ANTIALIAS_OFF",
28          "VALUE_TEXT_ANTIALIAS_ON");
29      makeButtons("KEY_FRACTIONALMETRICS", "VALUE_FRACTIONALMETRICS_OFF",
```

```
30          "VALUE_FRACTIONALMETRICS_ON");
31       makeButtons("KEY_RENDERING", "VALUE_RENDER_SPEED",
32          "VALUE_RENDER_QUALITY");
33       makeButtons("KEY_STROKE_CONTROL", "VALUE_STROKE_PURE",
34          "VALUE_STROKE_NORMALIZE");
35       canvas = new RenderQualityComponent();
36       canvas.setRenderingHints(hints);
37
38       add(canvas, BorderLayout.CENTER);
39       add(buttonBox, BorderLayout.NORTH);
40       pack();
41    }
42
43    /**
44     * Создает ряд кнопок-переключателей для выбора ключей и
45     * и значений конкретных указаний по воспроизведению
46     * @param key Имя ключа
47     * @param value1 Имя первого значения по данному ключу
48     * @param value2 Имя второго значения по данному ключу
49    */
50    void makeButtons(String key, String value1, String value2)
51    {
52       try
53       {
54          final RenderingHints.Key k = (RenderingHints.Key)
55             RenderingHints.class.getField(key).get(50 null);
56          final Object v1 = RenderingHints.class.getField(value1).get(null);
57          final Object v2 = RenderingHints.class.getField(value2).get(null);
58          JLabel label = new JLabel(key);
59
60          buttonBox.add(label, new GBC(0, r).setAnchor(GBC.WEST));
61          ButtonGroup group = new ButtonGroup();
62          JRadioButton b1 = new JRadioButton(value1, true);
63
64          buttonBox.add(b1, new GBC(1, r).setAnchor(GBC.WEST));
65          group.add(b1);
66          b1.addActionListener(new ActionListener()
67             {
68                public void actionPerformed(ActionEvent event)
69                {
70                   hints.put(k, v1);
71                   canvas.setRenderingHints(hints);
72                }
73             });
74          JRadioButton b2 = new JRadioButton(value2, false);
75
76          buttonBox.add(b2, new GBC(2, r).setAnchor(GBC.WEST));
77          group.add(b2);
78          b2.addActionListener(new ActionListener()
79           {
80             public void actionPerformed(ActionEvent event)
81             {
82                hints.put(k, v2);
83                canvas.setRenderingHints(hints);
84             }
85           });
86          hints.put(k, v1);
```

```
87          r++;
88        }
89        catch (Exception e)
90        {
91          e.printStackTrace();
92        }
93      }
94  }
95
96  /**
97   * Этот компонент производит рисунок, наглядно показывающий
98   * действие различных указаний по воспроизведению
99  */
100 class RenderQualityComponent extends JComponent
101 {
102   private static final int DEFAULT_WIDTH = 750;
103   private static final int DEFAULT_HEIGHT = 150;
104
105   private RenderingHints hints = new RenderingHints(null);
106   private Image image;
107
108   public RenderQualityComponent()
109   {
110     image = new ImageIcon(getClass().getResource("face.gif")).getImage();
111   }
112
113   public void paintComponent(Graphics g)
114   {
115     Graphics2D g2 = (Graphics2D) g;
116     g2.setRenderingHints(hints);
117
118     g2.draw(new Ellipse2D.Double(10, 10, 60, 50));
119     g2.setFont(new Font("Serif", Font.ITALIC, 40));
120     g2.drawString("Hello", 75, 50);
121
122     g2.draw(new Rectangle2D.Double(200, 10, 40, 40));
123     g2.draw(new Line2D.Double(201, 11, 239, 49));
124
125     g2.drawImage(image, 250, 10, 100, 100, null);
126   }
127
128   /**
129    * Устанавливает указания по воспроизведению и
130    * выполняет перерисовку
131    * @param h Указания по воспроизведению
132   */
133   public void setRenderingHints(RenderingHints h)
134   {
135     hints = h;
136     repaint();
137   }
138
139   public Dimension getPreferredSize()
140      { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
141 }
```

java.awt.Graphics2D 1.2

- `void setRenderingHint(RenderingHints.Key key, Object value)`

  Устанавливает указания по воспроизведению для данного графического контекста.

- `void setRenderingHints(Map m)`

  Устанавливает все указания по воспроизведению, пары "ключ-значение" которых хранятся в заданном отображении.

java.awt.RenderingHints 1.2

- `RenderingHints(Map<RenderingHints.Key, ?> m)`

  Создает отображение для хранения указаний по воспроизведению. Если параметр *m* принимает пустое значение `null`, то предоставляется отображение, реализуемое по умолчанию.

Чтение и запись изображений

До выпуска версии 1.4 возможности для чтения и записи файлов изображений в Java SE были весьма ограничены. Например, считывать можно было только файлы формата GIF и JPEG, а официальная поддержка записи изображений в файлы вообще отсутствовала.

Теперь ситуация выглядит значительно лучше. Еще в версии Java SE 1.4 появился пакет `javax.imageio` с готовыми средствами для чтения и записи файлов изображений нескольких наиболее типичных форматов, а также интегрированной средой разработки собственных средств для чтения и записи файлов других форматов. Что касается версии Java SE 6, то в ней поддерживаются такие форматы файлов изображений, как GIF, JPEG, PNG, BMP (растровый формат для Windows) и WBMP (Wireless Bitmap — растровый формат для беспроводных сетей). В прежних версиях из-за патентных ограничений запись изображений в файлы формата GIF не поддерживалась.

Основные функциональные возможности, доступные в библиотеке для чтения и записи изображений, чрезвычайно просты. Так, для загрузки изображения из файла применяется статический метод `read()` из класса `ImageIO`:

```
File f = . . .;
BufferedImage image = ImageIO.read(f);
```

Класс `ImageIO` выбирает соответствующее средство чтения, исходя из типа файла. Он проверяет расширение файла и соответствующее значение в заголовке файла. Если для чтения данного файла нельзя найти подходящее средство или оно не в состоянии расшифровать содержимое файла, то метод `read()` возвращает пустое значение `null`.

Так же просто осуществляется запись изображения в файл, как показано в приведенном ниже фрагменте кода. В данном случае строка `format` определяет формат изображения (например, JPEG или PNG), а класс `ImageIO` выбирает соответствующее средство записи и сохраняет изображение в файле.

```
File f = . . .;
String format = . . .;
ImageIO.write(image, format, f);
```

Получение средств чтения и записи изображений по типам файлов

Для выполнения расширенных операций записи и чтения изображений, которые выходят за пределы простого использования статических методов `read()` и `write()` из класса `ImageIO`, нужно прежде всего получить объекты типа `ImageReader` и `ImageWriter` соответственно. В классе `ImageIO` перечисляются средства чтения и записи, отвечающие одному из следующих условий.

- Формат изображения, например JPEG.
- Расширение файла, например **jpg**.
- Тип MIME, например **image/jpeg**.

НА ЗАМЕТКУ! Стандарт MIME (Multipurpose Internet Mail Extensions — многоцелевые расширения почты в Интернете) определяет общие форматы данных (например, `image/jpeg`, `application/pdf`). HTML-версию документа RFC (Requests for Comments — запросы на комментарии) с описанием стандарта MIME можно найти по адресу `http://www.oac.uci.edu/indiv/ehood/MIME`.

Например, с помощью приведенного ниже фрагмента кода можно получить средство чтения файлов формата JPEG. Методы `getImageReaderBySuffix()` и `getImageReaderByMIMEType()` возвращают средства чтения файлов с указанным расширением или типом MIME.

```
ImageReader reader = null;
Iterator<ImageReader> iter = ImageIO.getImageReadersByFormatName("JPEG");
if (iter.hasNext()) reader = iter.next();
```

Класс `ImageIO` может обнаружить несколько средств чтения, каждое из которых способно обработать файлы данного типа. В этом случае средство чтения выбирается исходя из более подробных сведений, которые можно получить с помощью интерфейса *поставщика услуг* следующим образом:

```
ImageReaderSpi spi = reader.getOriginatingProvider();
```

Затем можно получить название изготовителя и номер версии:

```
String vendor = spi.getVendor();
String version = spi.getVersion();
```

Возможно, эти сведения помогут сделать выбор подходящего средства чтения или хотя бы составить список доступных средств, чтобы пользователи могли выбрать наиболее подходящее из них по своему усмотрению. Но для начала будем считать, что для чтения изображений подходит первое же перечисленное средство.

В примере программы из листинга 7.7 требуется найти расширения файлов для всех доступных средств чтения, чтобы использовать их в фильтре файлов. Для этой цели можно вызвать статический метод `ImageIO.getReaderFileSuffixes()`, как показано ниже.

```
String[] extensions = ImageIO.getWriterFileSuffixes();
chooser.setFileFilter(new FileNameExtensionFilter("Image files",
extensions));
```

Что касается сохранения изображений в файлах, то для этого потребуется больше усилий. Хотелось бы, конечно, предоставить пользователю меню со всеми поддерживаемыми форматами изображений. Но, к сожалению, метод `getWriterFormaNames()` из класса `IOImage` для этой цели не подходит, потому что он возвращает не совсем обычный перечень с лишними вариантами названий форматов, например, следующий:

```
jpg, BMP, bmp, JPG, jpeg, wbmp, png, JPEG, PNG, WBMP, GIF, gif
```

Но это не совсем то, что должно отображаться в предполагаемом меню. Ведь это должен быть перечень только предпочитаемых названий форматов. Поэтому для этой цели используется вспомогательный метод `getWriterFormats()` (см. листинг 7.7). Сначала в этом методе отыскивается первое средство записи, ассоциируемое с названием каждого формата, а затем у него запрашиваются имеющиеся названия форматов в надежде, что первым будет перечислен наиболее подходящий формат. Для средства записи изображений в файлы формата JPEG такой подход вполне пригоден, поскольку первым это средство, естественно, перечислит формат JPEG. (А вот средство записи изображений в файлы формата PNG перечислит первым не название формата PNG, а его строчный эквивалент `png`. Можно надеяться, что в ближайшем будущем этот недостаток будет устранен, а до тех пор все строчные буквы в названии форматов придется преобразовать в прописные.) После выбора названия предпочтительного формата все его альтернативные названия просто удаляются из исходного набора, и так происходит до тех пор, пока не будут обработаны все названия форматов.

Чтение и запись файлов с несколькими изображениями

Некоторые файлы, например анимационного формата GIF, могут содержать несколько изображений. Но метод `read()` из класса `ImageIO` позволяет считывать только одно из них. Для чтения нескольких изображений нужно преобразовать сначала источник входных данных (например, поток ввода или файл) в объект потока ввода изображений типа `ImageInputStream` следующим образом:

```
InputStream in = . . .;
ImageInputStream imageIn = ImageIO.createImageInputStream(in);
```

Затем этот объект следует присоединить к средству чтения, вызвав следующий метод:

```
reader.setInput(imageIn, true);
```

Второй параметр этого метода принимает логическое значение `true`, а это означает, что поток ввода находится в режиме просмотра данных только в прямом направлении. Логическое значение `false` этого параметра допускает произвольный доступ. В этом случае данные, читаемые из потока ввода, буферизуются, или вводятся в режиме произвольного доступа к файлу. Произвольный доступ требуется только для определенных операций. Например, для подсчета количества изображений в файле анимационного формата GIF нужно прочитать весь этот файл. Если затем потребуется извлечь какое-нибудь изображение, то придется снова прочитать все введенные данные.

Это имеет значение только для чтения из потока ввода, если файл содержит несколько изображений, а формат изображения в его заголовке не предоставляет нужных сведений, например, о количестве изображений. А для чтения изображений непосредственно из файла можно воспользоваться приведенным ниже фрагментом кода.

```
File f = . . .;
ImageInputStream imageIn = ImageIO.createImageInputStream(f);
reader.setInput(imageIn);
```

Присоединив средство чтения к потоку ввода изображений, можно приступать к вводу из этого потока, вызвав приведенный ниже метод, где `index` — это индекс изображений, начиная с нуля.

```
BufferedImage image = reader.read(index);
```

Если поток ввода находится в режиме просмотра данных только в прямом направлении, то изображения должны считываться до тех пор, пока метод `read()` не сгенерирует исключение типа `IndexOutOgBoundsException`. В противном случае можно вызвать метод `getNumImages()` следующим образом:

```
int n = reader.getNumImages(true);
```

Параметр этого метода принимает логическое значение `true`, а это означает, что при вводе разрешается поиск и подсчет количества изображений. Следует иметь в виду, что метод `getNumImages()` генерирует исключение типа `IllegalStateException`, если поток ввода находится в режиме просмотра данных только в прямом направлении. Данный метод возвращает значение **-1**, если он не в состоянии определить количество изображений без поиска. В этом случае изображения придется считывать до тех пор, пока не возникнет исключение типа `IndexOutOfBoundsException`.

Некоторые файлы могут содержать миниатюрные виды изображений для их предварительного просмотра. Количество миниатюрных видов изображений можно выяснить, сделав следующий вызов:

```
int count = reader.getNumThumbnails(index);
```

А извлечь конкретный миниатюрный вид по его индексу можно следующим образом:

```
BufferedImage thumbnail = reader.getThumbnail(index, thumbnailIndex);
```

Иногда размеры изображения требуется получить еще до его извлечения из файла. Это особенно важно, когда речь идет о крупном изображении или его передаче по низкоскоростному сетевому соединению. Для получения размеров изображения по указанному индексу вызываются следующие методы:

```
int width = reader.getWidth(index);
int height = reader.getHeight(index);
```

Для записи нескольких изображений в файл нужно создать сначала объект типа `ImageWriter`, а затем с помощью класса `IOImage` перечислить все средства, способные записывать изображения в конкретном формате:

```
String format = . . .;
ImageWriter writer = null;
Iterator<ImageWriter> iter =
    ImageIO.getImageWritersByFormatName( format );
if (iter.hasNext()) writer = iter.next();
```

Затем поток ввода следует преобразовать в объект типа `ImageOutputStream` и присоединить его к средству записи следующим образом:

```
File f = . . .;
ImageOutputStream imageOut = ImageIO.createImageOutputStream(f);
writer.setOutput(imageOut);
```

Каждое изображение нужно заключить в оболочку объекта типа `IIOImage`, как показано ниже. Дополнительно можно также предоставить список миниатюрных видов и метаданные (например, алгоритм сжатия данных и цветовую информацию). В рассматриваемом здесь примере вместо списка миниатюрных видов и метаданных указываются пустые значения `null`. За дополнительной справкой по данному вопросу обращайтесь к документации на соответствующий прикладной интерфейс API.

```
IIOImage iioImage = new IIOImage(images[i], null, null);
```

Для записи *первого* изображения вызывается метод `write()`:

```
writer.write(new IIOImage(images[0], null, null));
```

А для записи всех последующих изображений служит приведенный ниже код. В качестве третьего параметра метода `write()` может быть указан объект типа `ImageWriteParam`, предоставляющий такие подробности, как мозаичное расположение и алгоритм сжатия данных. В данном примере в качестве третьего параметра указано пустое значение `null`.

```
if (writer.canInsertImage(i))
    writer.writeInsert(i, iioImage, null);
```

Не все форматы допускают сохранение нескольких изображений в файлах. В этом случае метод `canInsertImage()` возвращает логическое значение `false` по условию **`i > 0`**, и тогда в файле сохраняется только одно изображение. В примере программы из листинга 7.7 изображения загружаются и сохраняются в файлах тех форматов, для которых в библиотеке Java предусмотрены средства чтения и записи. Эта программа позволяет отображать сразу несколько изображений, но не их миниатюрные виды, как показано на рис. 7.27.

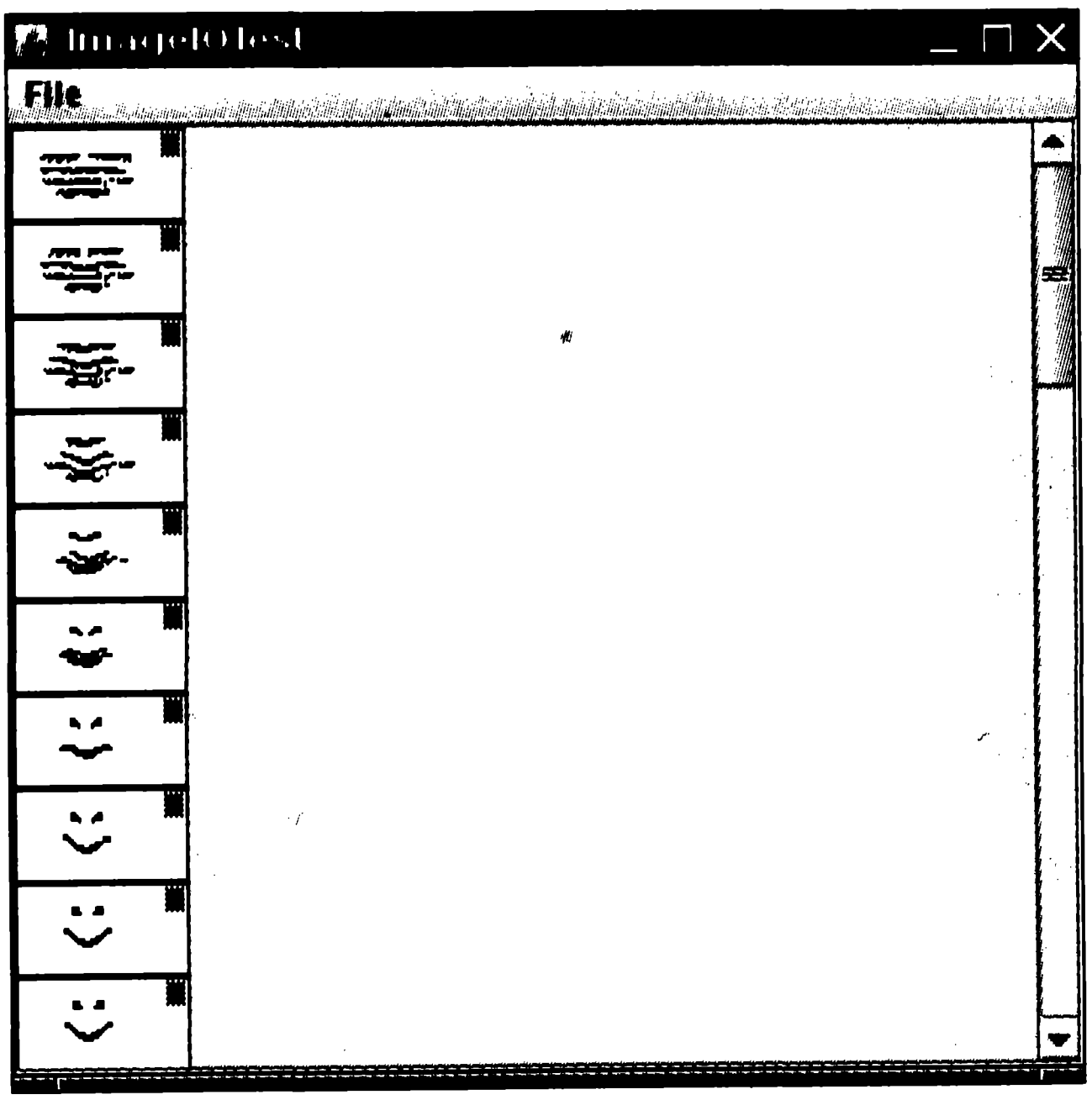


Рис. 7.27. Анимационная последовательность изображений в формате GIF

Листинг 7.7. Исходный код из файла `imageIO/ImageIOFrame.java`

```
1   package imageIO;
2
3   import java.awt.event.*;
4   import java.awt.image.*;
5   import java.io.*;
6   import java.util.*;
7   import javax.imageio.*;
8   import javax.imageio.stream.*;
9   import javax.swing.*;
10  import javax.swing.filechooser.*;
11
12  /**
13   * В этом фрейме отображаются загружаемые изображения. Для загрузки и
14   * сохранения изображений в файл предоставляются отдельные пункты меню
15  */
16  public class ImageIOFrame extends JFrame
17  {
18    private static final int DEFAULT_WIDTH = 400;
19    private static final int DEFAULT_HEIGHT = 400;
20
21    private static Set<String> writerFormats = getWriterFormats();
22
23    private BufferedImage[] images;
24
25    public ImageIOFrame()
26    {
27      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
28
29      JMenu fileMenu = new JMenu("File");
30      JMenuItem openItem = new JMenuItem("Open");
31      openItem.addActionListener(new ActionListener()
32        {
33          public void actionPerformed(ActionEvent event)
34          {
35            openFile();
36          }
37        });
38      fileMenu.add(openItem);
39
40      JMenu saveMenu = new JMenu("Save");
41      fileMenu.add(saveMenu);
42      Iterator<String> iter = writerFormats.iterator();
43      while (iter.hasNext())
44      {
45        final String formatName = iter.next();
46        JMenuItem formatItem = new JMenuItem(formatName);
47        saveMenu.add(formatItem);
48        formatItem.addActionListener(new ActionListener()
49          {
50            public void actionPerformed(ActionEvent event)
51            {
52              saveFile(formatName);
53            }
54          });
55      }
56
57      JMenuItem exitItem = new JMenuItem("Exit");
58      exitItem.addActionListener(new ActionListener()
```

```
59          {
60            public void actionPerformed(ActionEvent event)
61            {
62              System.exit(0);
63            }
64          });
65        fileMenu.add(exitItem);
66
67        JMenuBar menuBar = new JMenuBar();
68        menuBar.add(fileMenu);
69        setJMenuBar(menuBar);
70      }
71
72      /**
73       * Открыть файл и загрузить изображения
74      */
75      public void openFile()
76      {
77        JFileChooser chooser = new JFileChooser();
78        chooser.setCurrentDirectory(new File("."));
79        String[] extensions = ImageIO.getReaderFileSuffixes();
80        chooser.setFileFilter(new FileNameExtensionFilter(
81           "Image files", extensions));
82        int r = chooser.showOpenDialog(this);
83        if (r != JFileChooser.APPROVE_OPTION) return;
84        File f = chooser.getSelectedFile();
85        Box box = Box.createVerticalBox();
86        try
87        {
88          String name = f.getName();
89          String suffix = name.substring(name.lastIndexOf('.') + 1);
90          Iterator<ImageReader> iter =
91            ImageIO.getImageReadersBySuffix(suffix);
93          ImageReader reader = iter.next();
94          ImageInputStream imageIn = ImageIO.createImageInputStream(f);
95          reader.setInput(imageIn);
96          int count = reader.getNumImages(true);
97          images = new BufferedImage[count];
98          for (int i = 0; i < count; i++)
99          {
100           images[i] = reader.read(i);
101           box.add(new JLabel(new ImageIcon(images[i])));
102         }
103       }
104       catch (IOException e)
105       {
106         JOptionPane.showMessageDialog(this, e);
107       }
108       setContentPane(new JScrollPane(box));
109       validate();
110     }
111
112     /**
113      * Сохранить текущее изображение в файле
114      * @param formatName Формат файла
115     */
116     public void saveFile(final String formatName)
117     {
118       if (images == null) return;
119       Iterator<ImageWriter> iter =
```

```
120        ImageIO.getImageWritersByFormatName(formatName);
121     ImageWriter writer = iter.next();
122     JFileChooser chooser = new JFileChooser();
124     chooser.setCurrentDirectory(new File("."));
125     String[] extensions =
126        writer.getOriginatingProvider().getFileSuffixes();
127     chooser.setFileFilter(new FileNameExtensionFilter(
128        "Image files", extensions));
129
130     int r = chooser.showSaveDialog(this);
131     if (r != JFileChooser.APPROVE_OPTION) return;
132     File f = chooser.getSelectedFile();
133     try
134     {
135       ImageOutputStream imageOut =
136          ImageIO.createImageOutputStream(f);
137       writer.setOutput(imageOut);
138
139       writer.write(new IIOImage(images[0], null, null));
140       for (int i = 1; i < images.length; i++)
141       {
142         IIOImage iioImage = new IIOImage(images[i], null, null);
143         if (writer.canInsertImage(i))
144            writer.writeInsert(i, iioImage, null);
145       }
146     }
147     catch (IOException e)
148     {
149       JOptionPane.showMessageDialog(this, e);
150     }
151   }
152
153   /**
154    * Получает ряд предпочтительных названий форматов из всех средств
155    * записи. Предпочтительным считается первое название формата,
156    * указываемое средством записи
157    * @return Возвращает ряд названий форматов
158   */
159   public static Set<String> getWriterFormats()
160   {
161     Set<String> writerFormats = new TreeSet<>();
162     Set<String> formatNames = new TreeSet<>(Arrays.asList(ImageIO
163        .getWriterFormatNames()));
164     while (formatNames.size() > 0)
165     {
166       String name = formatNames.iterator().next();
167       Iterator<ImageWriter> iter =
168          ImageIO.getImageWritersByFormatName(name);
169       ImageWriter writer = iter.next();
170       String[] names =
171          writer.getOriginatingProvider().getFormatNames();
172       String format = names[0];
173       if (format.equals(format.toLowerCase())) format =
174          format.toUpperCase();
175       writerFormats.add(format);
176       formatNames.removeAll(Arrays.asList(names));
177     }
178     return writerFormats;
179   }
180 }
```

javax.imageio.ImageIO 1.4

- `static BufferedImage read(File` *input*`)`
- `static BufferedImage read(InputStream` *input*`)`
- `static BufferedImage read(URL` *input*`)`

  Считывают изображение из указанного источника ввода данных.
- `static boolean write(RenderedImage` *image*`, String` *formatName*`, File` *output*`)`
- `static boolean write(RenderedImage` *image*`, String` *formatName*`, OutputStream` *output*`)`

  Записывают изображение в указанное место назначения вывода данных. Возвращают логическое значение `false`, если соответствующее средство записи не удалось найти.
- `static Iterator<ImageReader> getImageReadersByFormatName(String` *formatName*`)`
- `static Iterator<ImageReader> getImageReadersBySuffix(String` *fileSuffix*`)`
- `static Iterator<ImageReader> getImageReadersByMIMEType(String` *mimeType*`)`
- `static Iterator<ImageWriter> getImageWritersByFormatName(String` *formatName*`)`
- `static Iterator<ImageWriter> getImageWritersBySuffix(String` *fileSuffix*`)`
- `static Iterator<ImageWriter> getImageWritersByMIMEType(String` *mimeType*`)`

  Получают все средства чтения или записи, поддерживающие указанный формат (например, JPG), расширение файла (например, `jpg`) или тип MIME (например, `mage/jpeg`).
- `static String[] getReaderFormatNames()`
- `static String[] getReaderMIMETypes()`
- `static String[] getWriterFormatNames()`
- `static String[] getWriterMIMETypes()`
- `static String[] getReaderFileSuffixes()` **6**
- `static String[] getWriterFileSuffixes()` **6**

  Получают названия всех форматов, расширений файлов и типов MIME, которые поддерживаются средствами чтения и записи.
- `ImageInputStream createImageInputStream(Object` *input*`)`
- `ImageOutputStream createImageOutputStream(Object` *output*`)`

  Создают поток ввода или вывода изображений, исходя из указанного объекта. В качестве такого объекта может быть указан файл, поток ввода-вывода, экземпляр класса `RandomAccessFile` или другой объект, для которого существует соответствующий поставщик услуг. Если для обращения с указанным объектом не зарегистрирован требующийся поставщик услуг, то возвращается пустое значение `null`.

javax.imageio.ImageReader 1.4

- `void setInput(Object input)`
- `void setInput(Object input, boolean seekForwardOnly)`

Устанавливают источник ввода данных для средства чтения.

Параметры:	*input*	Объект типа `ImageInputStream` или другой объект, приемлемый для данного средства чтения
	ImageInputStream	Принимает логическое значение `true` для чтения только в прямом направлении. По умолчанию средство чтения использует произвольный доступ и, если требуется, буферизирует данные изображения

- `BufferedImage read(int index)`

Считывает изображение по указанному индексу, начиная с нуля. Если такое изображение отсутствует, то генерируется исключение типа `IndexOutOfBoundsException`.

- `int getNumImages(boolean allowSearch)`

Получает количество изображений для данного средства чтения. Если параметр *allowSearch* принимает логическое значение `false` и количество изображений не может быть определено без предварительного просмотра, то возвращается значение **-1**. Если же параметр `allowSearch` принимает логическое значение `true` и средство чтения находится в режиме просмотра данных только в прямом направлении, то генерируется исключение типа `IllegalStateException`.

- `int getNumThumbnails(int index)`

Получает количество миниатюрных видов изображения по указанному индексу.

- `BufferedImage readThumbnail(int index, int thumbnailIndex)`

Получает миниатюрный вид по индексу *thumbnailIndex* изображения по указанному индексу `index`.

- `int getWidth(int index)`
- `int getHeight(int index)`

Получают ширину и высоту изображения. Если изображение не найдено, то генерируется исключение типа `IndexOutOfBoundsException`.

- `ImageReaderSpi getOriginatingProvider()`

Получает поставщика услуг, создавшего данное средство чтения.

javax.imageio.spi.IIOServiceProvider 1.4

- `String getVendorName()`
- `String getVersion()`

  Получают имя и версию данного поставщика услуг.

javax.imageio.spi.ImageReaderWriterSpi 1.4

- `String[] getFormatNames()`
- `String[] getFileSuffixes()`
- `String[] getMIMETypes()`

  Получают названия форматов, расширения файлов и типа MIME, которые поддерживаются средствами чтения или записи, созданными данным поставщиком услуг.

javax.imageio.ImageWriter 1.4

- `void setOutput(Object output)`

  Устанавливает место назначения вывода для данного средства записи.

  Параметры: *output* Объект типа `ImageOutputStream` или другой объект, приемлемый для данного средства записи

- `void write(IIOImage image)`
- `void write(RenderedImage image)`

  Записывают одиночное изображение по месту вывода.

- `void writeInsert(int index, IIOImage image, ImageWriteParam param)`

  Записывает изображение в файл с несколькими изображениями.

- `boolean canInsertImage(int index)`

  Возвращает логическое значение `true`, если в файл можно ввести изображение по указанному индексу.

- `ImageWriterSpi getOriginatingProvider()`

  Получает поставщика услуг, создавшего данное средство записи.

javax.imageio.IIOImage 1.4

- `IIOImage(RenderedImage image, List thumbnails, IIOMetadata metadata)`
- Создает объект типа `IIOImage`, исходя из указанного изображения, необязательных миниатюрных видов и метаданных.

Манипулирование изображениями

Допустим, требуется улучшить внешний вид какого-нибудь изображения. Для этого нужно получить доступ к отдельным пикселям изображения и заменить их другими. Не исключено также, что придется сформировать новые пиксели сызнова, например, для отображения результатов физических измерений или математических расчетов. Класс `BufferedImage` дает возможность манипулировать пикселями изображения, а классы, реализующие интерфейс `BufferedImageOp`, — преобразовывать изображения.

НА ЗАМЕТКУ! В версии JDK 1.0 предоставлялась совершенно другая и намного более сложная среда для обработки изображений, которая была оптимизирована для *пошагового воспроизведения* изображений, построчно загружавшихся из Интернета. Но манипулировать изображениями в такой среде было совсем не просто. Поэтому в данной книге эта среда не рассматривается.

Формирование растровых изображений

Большинство обрабатываемых изображений считываются из файла и формируются аппаратными средствами (например, цифровой камерой или сканером) или программными (например, графическими приложениями). В этом разделе рассматривается совершенно другая методика формирования изображений, предполагающая построение в каждый отдельный момент времени только одного пикселя.

Для того чтобы сформировать изображение, следует построить сначала объект типа `BufferedImage` обычным способом:

```
image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
```

Затем нужно вызвать метод `getRaster()`, как показано ниже, чтобы получить объект типа `WritableRaster`, который служит для доступа к пикселям изображения с целью внести в них изменения.

```
WritableRaster raster = image.getRaster();
```

Метод `setPixel()` позволяет задавать значение цвета отдельного пикселя. Но при его применении возникает следующее осложнение: пикселю нельзя непосредственно присвоить значение цвета типа `Color`. Нужно знать, каким образом цвета определяются в буферизованном изображении, что зависит от его *типа*. Так, если изображение относится к типу `TYPE_INT_ARGB`, то каждый пиксель описывается четырьмя значениями (в пределах от **0** до **255**): для красной, зеленой и синей составляющих, а также для альфа-канала. Эти значения нужно предоставить в виде массива из четырех элементов целого типа, как показано ниже. В терминах Java 2D API эти значения называются *выборочными значениями* цвета пикселя.

```
int[] black = { 0, 0, 0, 255 };
raster.setPixel(i, j, black);
```

ВНИМАНИЕ! Метод `setPixel()` может принимать в качестве параметра массив типа `float[]` или `double[]`. Следует, однако, иметь в виду, что значения цвета, которыми заполняется этот массив, *не* должны быть нормализованными в пределах от **0** до **1**,**0**, как показано ниже.

```
float[] red = { 1.0F, 0.0F, 0.0F, 1.0F };
raster.setPixel(i, j, red); // ОШИБКА!
```

Независимо от типа массив все равно должен быть заполнен значениями цвета в пределах от **0** до **255**.

Для установки значений прямоугольной группы пикселей можно указать в качестве параметров метода `setPixels()` исходное положение, ширину и высоту прямоугольника, а также массив с выборочными значениями цвета этой группы пикселей. Так, если изображение относится к типу `TYPE_INT_ARGB`, сначала следует указать величины красной, зеленой и синей составляющих и значение прозрачности в альфа-канале для первого пикселя, а затем те же самые данные для второго пикселя и т.д. В приведенном ниже фрагменте кода показано, как это делается.

```
int[] pixels = new int[4 * width * height];
pixels[0] = . . . // значение красной составляющей цвета первого пикселя
pixels[1] = . . . // значение зеленой составляющей цвета первого пикселя
pixels[2] = . . . // значение синей составляющей цвета первого пикселя
pixels[3] = . . . // значение прозрачности в альфа-канале первого пикселя
. . .
raster.setPixels(x, y, width, height, pixels);
```

А для считывания значения цвета пикселя служит приведенный ниже метод `getPixel()`, который возвращает массив из четырех целых чисел.

```
int[] sample = new int[4];
raster.getPixel(x, y, sample);
Color c = new Color(sample[0], sample[1], sample[2], sample[3]);
```

С помощью метода `getPixels()` можно прочитать значения цвета группы пикселей следующим образом:

```
raster.getPixels(x, y, width, height, samples);
```

Методы `getPixel()` и `setPixel()` можно использовать и для другого типа изображения, если известен способ представления в нем значений цвета пикселей. Иными словами, чтобы применять эти методы, нужно знать способ кодирования выборочных значений цвета пикселей в конкретном типе изображения.

Манипулировать изображением немного сложнее, если его тип заранее неизвестен. У каждого типа изображений имеется своя *цветовая модель*, в соответствии с которой выборочные значения цвета пикселей можно привести к стандартной цветовой модели RGB.

НА ЗАМЕТКУ! На самом деле цветовая модель RGB не является стандартом. Точное представление цветов зависит от характеристик используемого устройства отображения. Цифровые камеры, сканеры, мониторы и жидкокристаллические дисплеи имеют совершенно разные характеристики. В результате одно и то же значение цвета RGB выглядит по-разному на разных устройствах отображения. Международный консорциум по цвету (International Color Consortium — ICC; `http://www.color.org`) рекомендует сопровождать все цветовые данные цветовым *профилем ICC*, который определяет степень соответствия цветов стандартной форме их представления по спецификации 1931 CIE XYZ. Эта спецификация разработана Международной комиссией по освещению (Commission Internationale de l'Eclairage — CIE; `http://www.cie.co.at/cie`), обеспечивающей техническую поддержку всех видов освещения и цвета. Она регламентирует стандартный метод представления всех цветов, которые человеческий глаз может воспринять на основе трех гипотетических координат X, Y, Z. (Подробнее эта спецификация описана в главе 13 упоминавшейся ранее книги *Computer Graphics: Principles and Practice, Second Edititon in C* Джеймса Фоли, Андриса ван Дама, Стивена Фэйнера и др.)

Но цветовые профили ICC имеют очень сложную структуру. Поэтому наряду с ними используется более простой стандарт sRGB (`http://www.w3.org/Graphics/Color/sRGB.html`), который устанавливает точное соответствие значений цветов RGB спецификации 1931 CIE XYZ. Он разработан специально для стандартных цветных мониторов и используется в прикладном интерфейсе Java 2D API для преобразования цветов RGB в значения цветов других цветовых моделей и пространств.

Для получения цветовой модели вызывается метод `getColorModel()`:

```
ColorModel model = image.getColorModel();
```

Для определения цвета пикселя следует вызвать приведенный ниже метод `getDataElements()` из класса `Raster`, который возвращает объект типа `Object` с описанием цвета в соответствии с конкретной цветовой моделью.

```
Object data = raster.getDataElements(x, y, null);
```

НА ЗАМЕТКУ! Объект, который возвращается методом `getDataElements()`, в действительности является массивом выборочных значений цвета пикселя. Это объясняет, почему метод `getDataElements()` называется именно так, а не иначе. Хотя для обработки данного объекта знать об этом совсем не обязательно.

С помощью цветовой модели можно преобразовать полученный объект в стандартные значения цвета ARGB. Метод `getRGB()` возвращает значение типа `int`, состоящее из значений прозрачности в альфа-канале, красной, зеленой и синей составляющих цвета, упакованных в четыре блока по 8 бит в каждом. Из этого целочисленного составного значения цвета с помощью конструктора `Color(int argb, boolean hasAlpha)` можно создать объект типа `Color`:

```
int argb = model.getRGB(data);
Color color = new Color(argb, true);
```

Если же требуется установить определенный цвет пикселя, то описанные выше действия следует выполнить в обратном порядке. Метод `getRGB()` из класса `Color` возвращает целочисленное значение, состоящее из значений прозрачности в альфа-канале, красной, зеленой и синей составляющих цвета. Именно его следует указать в качестве параметра при вызове метода `getDataElements()` из класса `ColorModel`, который, в свою очередь, возвратит объект с описанием цвета пикселя в используемой цветовой модели. Далее этот объект нужно передать методу `setDataElements()` из класса `WritableRaster`, как показано ниже.

```
int argb = color.getRGB();
Object data = model.getDataElements(argb, null);
raster.setDataElements(x, y, data);
```

Для того чтобы проиллюстрировать, каким образом изображение формируется из отдельных пикселей, рассмотрим пример рисования фрактального множества Мандельброта, отдавая дань традиции. Множество Мандельброта образуется путем связывания каждой точки плоскости с некоторой последовательностью цифр. Если последовательность ограничена, точка отмечается цветом. Неокрашенные точки плоскости связаны с неограниченными последовательностями цифр и остаются прозрачными (рис. 7.28).

Ниже показано, как построить простейшее множество Мандельброта. Сначала нужно отыскать для каждой точки с координатами (a, b) последовательность, которая начинается с $(x, y) = (0, 0)$, и применить к ней следующие формулы:

$$x_{new} = x^2 - y^2 + a$$
$$y_{new} = 2 \cdot x \cdot y + b$$

Если значение x или y оказывается больше 2, последовательность будет превращаться в бесконечную и окрашиваться цветом будут только те пиксели, которые соответствуют точкам с координатами (a, b), ведущим к ограниченной последовательности.

(Формулы для вычисления числовых последовательностей взяты из области математики комплексных чисел в их исходном виде. А дополнительные сведения о математике фракталов можно получить, например, по адресу `http://classes.yale.edu/fractals/`.)

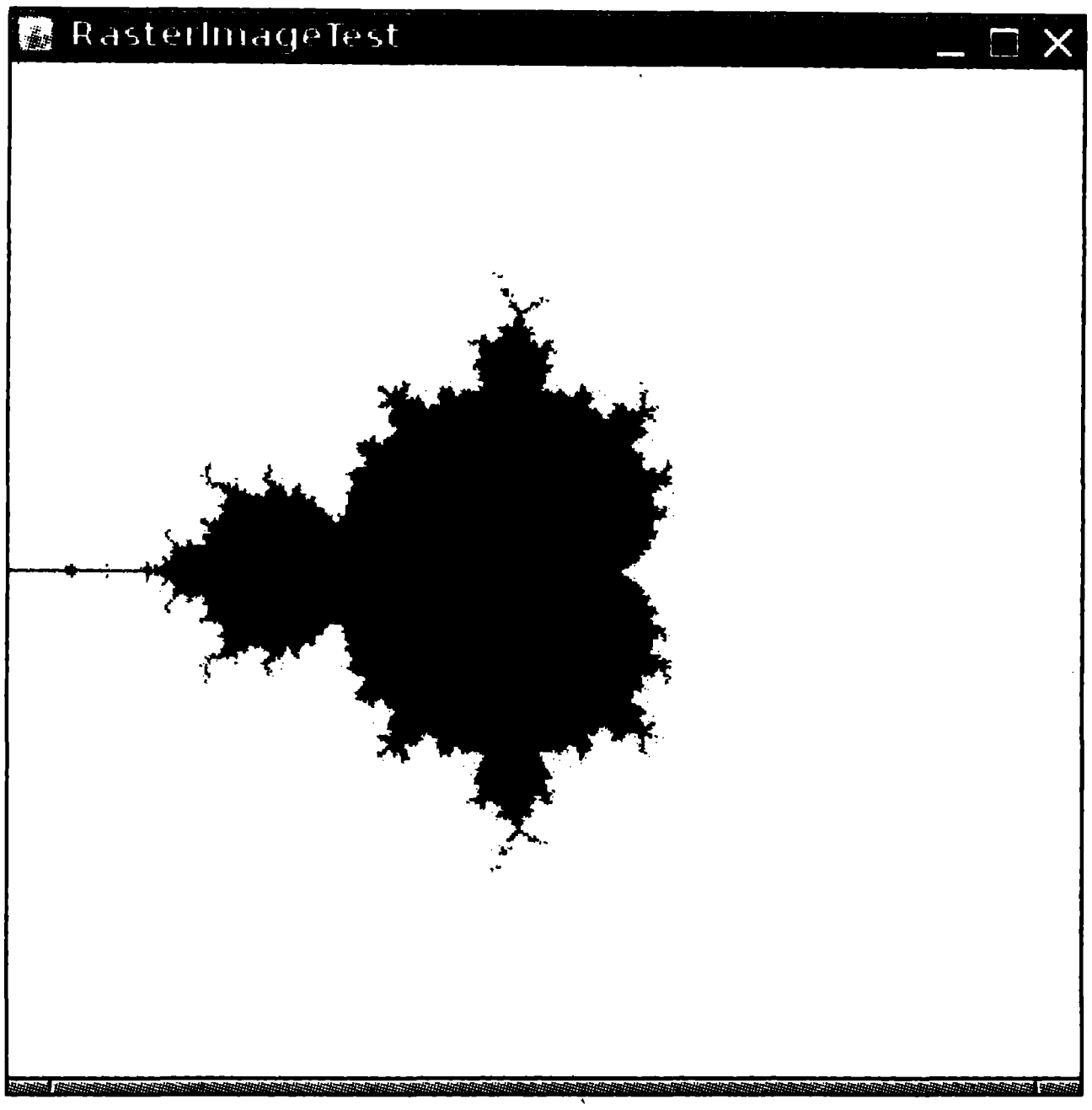


Рис. 7.28. Множество Мандельброта

В примере программы, исходный код которой приведен в листинге 7.8, демонстрируется применение класса `ColorModel` для преобразования значений цвета типа `Color` в выборочные данные цвета пикселей. Этот процесс не зависит от типа изображения. Ради интереса попробуйте поэкспериментировать, изменив тип цвета для буферизованного изображения на `TYPE_BYTE_GRAY`. В исходный код данной программы не нужно вносить больше никаких других изменений, так как цветовая модель изображения автоматически преобразует цвета пикселей в выборочные значения.

Листинг 7.8. Исходный код из файла `rasterImage/RasterImageFrame.java`

```
1  package rasterImage;
2
3  import java.awt.*;
4  import java.awt.image.*;
5  import javax.swing.*;
6
7  /**
8   * В этом фрейме показывается изображение множества Мандельброта
9  */
10 public class RasterImageFrame extends JFrame
11 {
12   private static final double XMIN = -2;
13   private static final double XMAX = 2;
```

```
14   private static final double YMIN = -2;
15   private static final double YMAX = 2;
16   private static final int MAX_ITERATIONS = 16;
17   private static final int IMAGE_WIDTH = 400;
18   private static final int IMAGE_HEIGHT = 400;
19
20   public RasterImageFrame()
21   {
22     BufferedImage image = makeMandelbrot(IMAGE_WIDTH, IMAGE_HEIGHT);
23     add(new JLabel(new ImageIcon(image)));
24     pack();
25   }
26
27   /**
28    * Формирует изображение множества Мандельброта
29    * @param width Ширина изображения
30    * @parah height Высота изображения
31    * @return Возвращает изображение
32   */
33   public BufferedImage makeMandelbrot(int width, int height)
34   {
35     BufferedImage image = new BufferedImage(width, height,
36        BufferedImage.TYPE_INT_ARGB);
37     WritableRaster raster = image.getRaster();
38     ColorModel model = image.getColorModel();
39
40     Color fractalColor = Color.red;
41     int argb = fractalColor.getRGB();
42     Object colorData = model.getDataElements(argb, null);
43
44     for (int i = 0; i < width; i++)
45       for (int j = 0; j < height; j++)
46       {
47         double a = XMIN + i * (XMAX - XMIN) / width;
48         double b = YMIN + j * (YMAX - YMIN) / height;
49         if (!escapesToInfinity(a, b))
50            raster.setDataElements(i, j, colorData);
51       }
52     return image;
53   }
54
55   private boolean escapesToInfinity(double a, double b)
56   {
57     double x = 0.0;
58     double y = 0.0;
59     int iterations = 0;
60     while (x <= 2 && y <= 2 && iterations < MAX_ITERATIONS)
61     {
62       double xnew = x * x - y * y + a;
63       double ynew = 2 * x * y + b;
64       x = xnew;
65       y = ynew;
66       iterations++;
67     }
68     return x > 2 || y > 2;
69   }
70 }
```

java.awt.image.BufferedImage 1.2

- `BufferedImage(int width, int height, int imageType)`
- Создает объект буферизованного изображения.

  Параметры:

width, *height*	Размеры изображения
imageType	Тип изображения. К числу наиболее распространенных относятся следующие типы изображений: TYPE_INT_RGB, TYPE_INT_ARGB, TYPE_BYTE_GRAY и TYPE_BYTE_INDEXED

- `ColorModel getColorModel()`

  Возвращает цветовую модель данного буферизованного изображения.

- `WritableRaster getRaster()`

  Возвращает растр для доступа к пикселям данного буферизованного изображения с целью их изменения.

java.awt.image.Raster 1.2

- `Object getDataElements(int x, int y, Object data)`

  Возвращает для точки растра выборочные данные в виде массива, тип элементов и длина которого зависят от цветовой модели. Если параметр `data` не принимает пустое значение `null`, то предполагается, что это значение является массивом для хранения выборочных данных, причем массив заполнен. А если параметр `data` принимает пустое значение `null`, то создается новый массив. Тип его элементов и длина зависят от цветовой модели.

- `int[] getPixel(int x, int y, int[] sampleValues)`
- `float[] getPixel(int x, int y, float[] sampleValues)`
- `double[] getPixel(int x, int y, double[] sampleValues)`
- `int[] getPixels(int x, int y, int width, int height, int[] sampleValues)`
- `float[] getPixels(int x, int y, int width, int height, float[] sampleValues)`
- `double[] getPixels(int x, int y, int width, int height, double[] sampleValues)`

  Возвращают выборочные значения для точки растра или группы точек, составляющих прямоугольную область. Эти значения помещаются в массив, длина которого зависит от конкретной цветовой модели. Если параметр *sampleValues* принимает пустое значение `null`, то создается новый массив. Применяются только в том случае, если заранее известно, каким образом выборочные значения определяются для цветовой модели.

java.awt.image.WritableRaster 1.2

- `void setDataElements(int x, int y, Object data)`

  Задает выборочные данные для точки растра. Параметр *data* обозначает массив выборочных значений цвета пикселя. Тип элементов и длина этого массива зависят от конкретной цветовой модели.

- `void setPixel(int x, int y, int[] sampleValues)`
- `void setPixel(int x, int y, float[] sampleValues)`
- `void setPixel(int x, int y, double[] sampleValues)`
- `void setPixels(int x, int y, int width, int height, int[] sampleValues)`
- `void setPixels(int x, int y, int width, int height, float[] sampleValues)`
- `void setPixels(int x, int y, int width, int height, double[] sampleValues)`

  Устанавливают выборочные значения для точки растра или группы точек, составляющих прямоугольную область. Применяются только в том случае, если заранее известен порядок кодировки выборочных значений для цветовой модели.

java.awt.image.ColorModel 1.2

- `int getRGB(Object data)`

  Возвращает значение цвета ARGB, которое соответствует выборочным данным, переданным в массиве *data*, тип элементов и длина которого зависят от конкретной цветовой модели.

- `Object getDataElements(int argb, Object data);`

  Возвращает выборочные данные для указанного значения цвета. Если параметр `data` не принимает пустое значение `null`, то предполагается, что это массив, имеющий подходящую длину для хранения данных и заполненный ими. А если параметр *data* принимает пустое значение `null`, то создается новый массив. Этот массив заполняется выборочными данными цвета пикселя. Тип его элементов и длина массива зависят от цветовой модели.

java.awt.Color 1.0

- `Color(int argb, boolean hasAlpha)` **1.2**

  Формирует цвет по указанному составному значению ARGB, если параметр *hasAlpha* принимает логическое значение `true`, или же по указанному стандартному значению RGB, если параметр `hasAlpha` принимает логическое значение `false`.

- `int getRGB()`

  Возвращает значение ARGB, соответствующее данному цвету.

Фильтрация изображений

В предыдущем разделе рассматривался процесс формирования изображения сызнова. Но нередко требуется обработать уже имеющееся изображение. Безусловно, данные изображения можно сначала прочитать, используя методы `getPixel()`/`getDataElements()`, представленные в предыдущем разделе, затем преобразовать их и записать обратно. Но, к счастью, в прикладном интерфейсе Java 2D API поддерживается целый ряд *фильтров*, автоматически выполняющих многие рутинные операции обработки изображений.

Все классы, выполняющие операции манипулирования изображениями, реализуют интерфейс `BufferedImageOp`. Для преобразования одного изображения в другое после создания объекта требующейся операции достаточно вызвать метод `filter()` следующим образом:

```
BufferedImageOp op = . . .;
BufferedImage filteredImage =
   new BufferedImage(image.getWidth(), image.getHeight(), image.getType());
op.filter(image, filteredImage);
```

Некоторые методы, например `op.filter(image, image)`, могут непосредственно выполнять преобразования изображений, но большинство остальных методов не способны на это. Ниже перечислены пять классов, реализующих интерфейс `BufferedImageOp`.

```
AffineTransformOp
RescaleOp
LookupOp
ColorConvertOp
ConvolveOp
```

В частности, класс `AffineTransformOp` выполняет аффинное преобразование пикселей. Например, в приведенном ниже фрагменте кода показано, как повернуть изображение вокруг его центра.

```
AffineTransform transform = AffineTransform.getRotateInstance(
   Math.toRadians(angle), image.getWidth() / 2, image.getHeight() / 2);
AffineTransformOp op = new AffineTransformOp(transform, interpolation);
op.filter(image, filteredImage);
```

При вызове конструктора класса `AffineTransformOp` в качестве параметров следует указать аффинное преобразование и алгоритм *интерполяции*. Интерполяция требуется для определения местоположения пикселей в целевом изображении, если исходные пиксели оказываются после преобразования где-то между целевыми пикселями.

Например, после вращения исходные пиксели точно совпадают с целевыми пикселями. Для интерполяции используются два алгоритма: билинейный (`AffineTransformOp.TYPE_BILINEAR`) и ближайшего соседа (`AffineTransformOp.TYPE_NEAREST_NEIGHBOR`). Билинейная интерполяция выполняется дольше, но позволяет добиться более качественного внешнего вида преобразованного изображения.

В примере программы из листинга 7.9 изображение поворачивается на 5° (рис. 7.29). В частности, класс `RescaleOp` выполняет для каждой составляющей цвета изображения следующую операцию по изменению масштаба:

$$x_{new} = a \cdot x + b$$

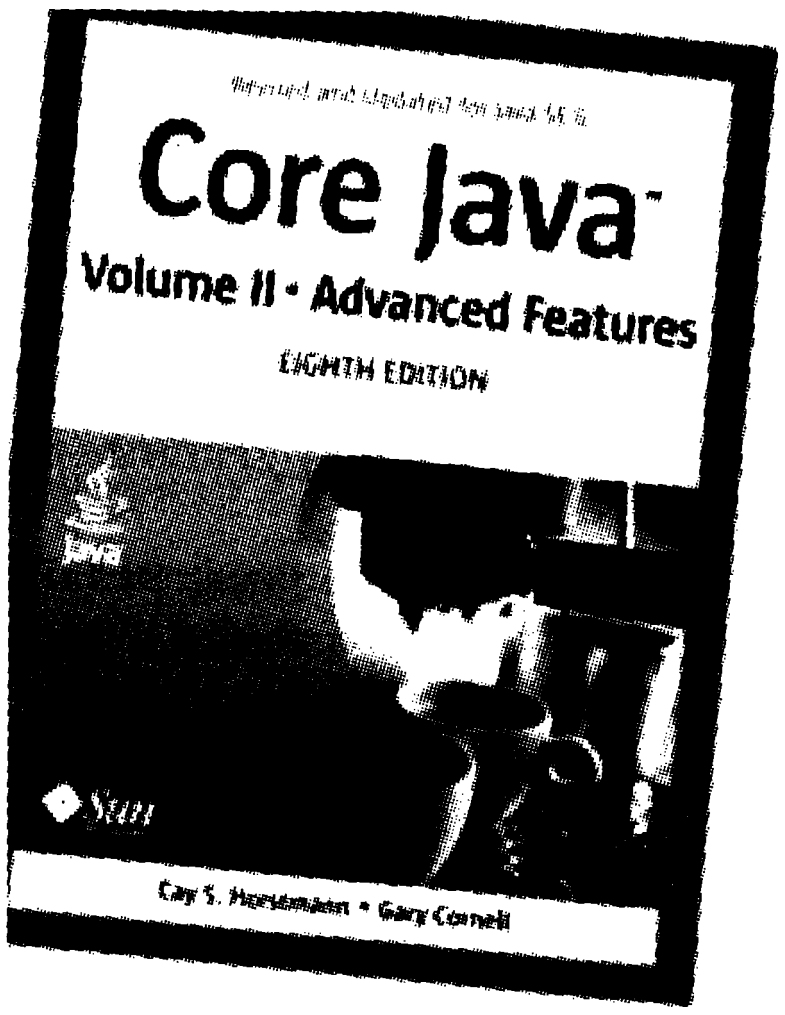


Рис. 7.29. Вращение изображения

(На составляющие прозрачности из альфа-канала эта операция не распространяется). В результате изменения масштаба на величину `a > 1` увеличивается яркость изображения. Объект типа `RescaleOp` создается по указанным параметрам масштабирования и дополнительным, но не обязательным параметрам рисования. В примере программы из листинга 7.9 для этой цели используется приведенный ниже фрагмент кода. Кроме того, отдельные величины изменения масштаба можно предоставить для каждой составляющей цвета, как поясняется далее при описании прикладного интерфейса API.

```
float a = 1.1f;
float 20.0f;
RescaleOp op = new RescaleOp(a, b, null);
```

Конструктор класса `LookupOp` позволяет указать произвольное отображение выборочных значений цвета. Для этого достаточно предоставить таблицу, в которой указаны правила отображения каждого выборочного значения цвета. В рассматриваемом здесь примере формируется *негатив* исходного изображения. Для этого каждый цвет `c` заменяется цветом `255 - c`.

В качестве параметров конструктору класса `LookupOp` требуется указать объект типа `LookupTable` и таблицу с необязательными указаниями по воспроизведению. Класс `LookupTable` является абстрактным и имеет два конкретных подкласса: `ByteLookupTable` и `ShortLookupTable`. На первый взгляд, для целей преобразования было бы достаточно и класса `ByteLookupTable`, поскольку значения цвета RGB представлены в виде байтов. Но из-за программной ошибки, подробно описанной в документации по адресу `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6183251`, вместо него в данном примере используется класс `ShortLookupTable`. В приведенном ниже фрагменте кода показано, каким образом формируется операция табличного поиска для преобразования цветного изображения в негатив.

```
short negative[] = new short[256];
for (int i = 0; i < 256; i++) negative[i] = (short) (255 - i);
ShortLookupTable table = new ShortLookupTable(0, negative);
LookupOp op = new LookupOp(table, null);
```

Данная операция выполняется над каждой составляющей цвета в отдельности, но не над составляющими прозрачности в альфа-канале. Для преобразования каждой составляющей цвета можно предоставить и другие таблицы поиска (подробнее об этом см. далее в описании прикладного интерфейса API).

НА ЗАМЕТКУ! Класс `LookupOp` нельзя применять к изображениям с индексированной цифровой моделью, потому что в таких изображениях каждое выборочное значение цвета пикселей задается в виде смещения в палитре цветов.

Для преобразования цветового пространства служит класс `ColorConvertOp`, который здесь не рассматривается. А для наиболее глубоких преобразований

предназначен класс `ConvolveOp`, который выполняет математическую операцию *свертки*. Мы не будем углубляться в математические подробности выполнения операции свертки, но понять основную идею несложно. В качестве примера рассмотрим фильтр размытия, действие которого показано на рис. 7.30.

Размытие достигается путем замены каждого пикселя средним значением этого пикселя и соседних с ним восьми пикселей. Интуитивно понятно, почему эта операция делает изображение размытым, а с математической точки зрения получение среднего значения может быть представлено как операция свертки со следующим ядром:

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Ядро свертки представляет собой матрицу с весами соседних значений. Например, приведенное выше ядро приводит к размытию изображения. А представленное ниже ядро выполняет операцию *определения краев*, т.е. определяет участки, на которых происходит изменение цвета. Алгоритм определения краев играет важную роль в обработке фотоизображений (рис. 7.31).

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

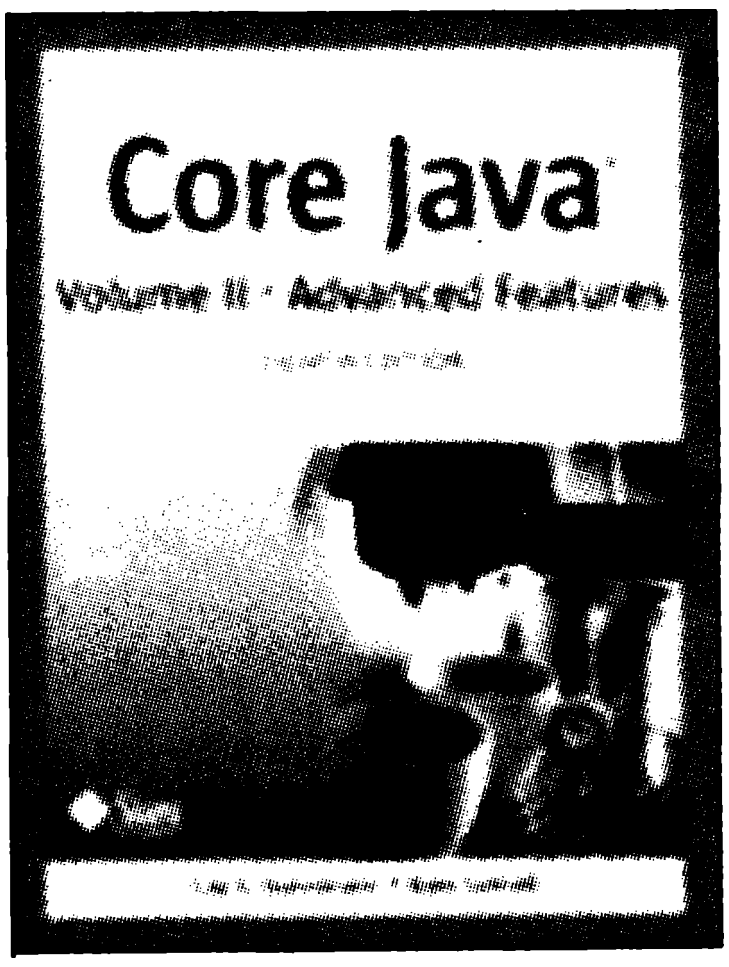


Рис. 7.30. Размытие изображения

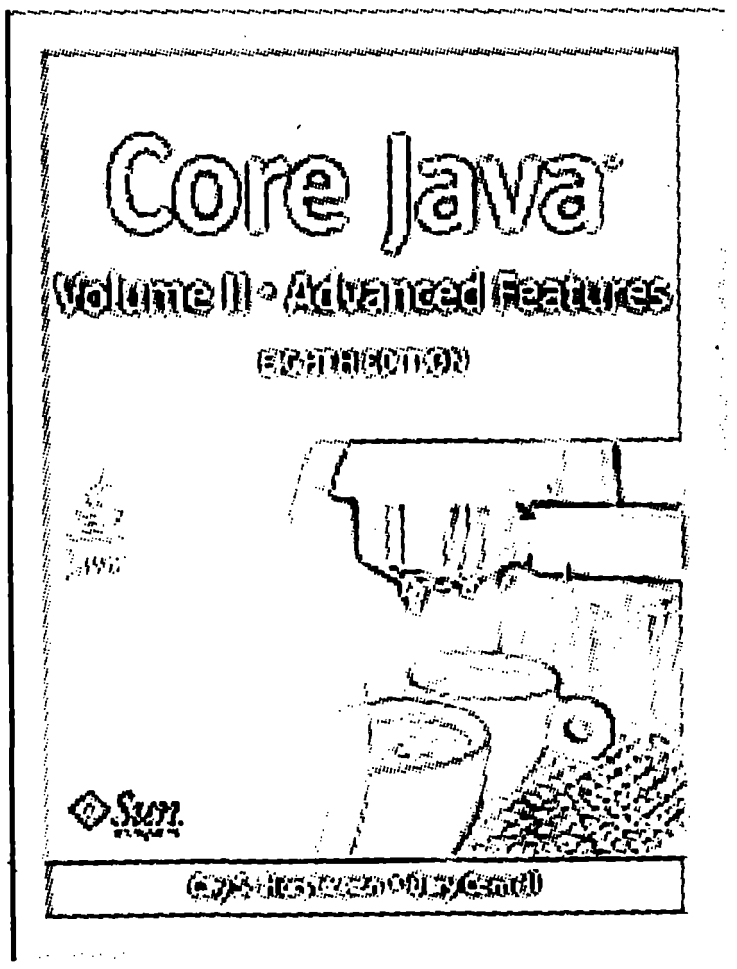


Рис. 7.31. Определение краев и инверсия

Для того чтобы сформировать операцию свертки, нужно сначала составить массив значений для ядра свертки и создать объект типа `Kernel`. Затем на основе этого ядра необходимо создать объект типа `ConvolveOp`, чтобы использовать его для фильтрации изображения следующим образом:

```
float[] elements =
   {
     0.0f, -1.0f, 0.0f,
```

```
      -1.0f, 4.f, -1.0f,
      0.0f, -1.0f, 0.0f
   };
Kernel kernel = new Kernel(3, 3, elements);
ConvolveOp op = new ConvolveOp(kernel);
op.filter(image, filteredImage);
```

В примере программы из листинга 7.9 пользователю предоставляется возможность загружать файлы формата GIF и JPEG и выполнять над ними описанные выше операции. Простота кода этой программы объясняется тем, что в прикладном интерфейсе Java 2D API предусмотрены весьма эффективные средства для обработки изображений.

Листинг 7.9. Исходный код из файла `ImageProcessing/ImageProcessingFrame.java`

```
1   package imageProcessing;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.awt.geom.*;
6   import java.awt.image.*;
7   import java.io.*;
8   import javax.imageio.*;
9   import javax.swing.*;
10  import javax.swing.filechooser.*;
11
12  /**
13   * Этот фрейм содержит меню для загрузки изображения и выбора различных
14   * его преобразований, а также компонент для показа итогового изображения
15  */
16  public class ImageProcessingFrame extends JFrame
17  {
18    private static final int DEFAULT_WIDTH = 400;
19    private static final int DEFAULT_HEIGHT = 400;
20
21    private BufferedImage image;
22
23    public ImageProcessingFrame()
24    {
25      setTitle("ImageProcessingTest");
26      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
27
28      add(new JComponent()
29        {
30          public void paintComponent(Graphics g)
31          {
32            if (image != null) g.drawImage(image, 0, 0, null);
33          }
34        });
35
36      JMenu fileMenu = new JMenu("File");
37      JMenuItem openItem = new JMenuItem("Open");
38      openItem.addActionListener(new ActionListener()
39        {
40          public void actionPerformed(ActionEvent event)
41          {
42            openFile();
43          }
```

```
44          });
45        fileMenu.add(openItem);
46
47        JMenuItem exitItem = new JMenuItem("Exit");
48        exitItem.addActionListener(new ActionListener()
49          {
50            public void actionPerformed(ActionEvent event)
51            {
52              System.exit(0);
53            }
54          });
55        fileMenu.add(exitItem);
56
57        JMenu editMenu = new JMenu("Edit");
58        JMenuItem blurItem = new JMenuItem("Blur");
59        blurItem.addActionListener(new ActionListener()
60          {
61            public void actionPerformed(ActionEvent event)
62            {
63              float weight = 1.0f / 9.0f;
64              float[] elements = new float[9];
65              for (int i = 0; i < 9; i++)
66              elements[i] = weight;
67              convolve(elements);
68            }
69          });
70        editMenu.add(blurItem);
71
72        JMenuItem sharpenItem = new JMenuItem("Sharpen");
73        sharpenItem.addActionListener(new ActionListener()
74          {
75            public void actionPerformed(ActionEvent event)
76            {
77              float[] elements =
78                 { 0.0f, -1.0f, 0.0f, -1.0f, 5.f, -1.0f,
79                   0.0f, -1.0f, 0.0f };
80              convolve(elements);
81            }
82          });
83        editMenu.add(sharpenItem);
84
85        JMenuItem brightenItem = new JMenuItem("Brighten");
86        brightenItem.addActionListener(new ActionListener()
87          {
88            public void actionPerformed(ActionEvent event)
89            {
90              float a = 1.1f;
91              // float b = 20.0f;
92              float b = 0;
93              RescaleOp op = new RescaleOp(a, b, null);
94              filter(op);
95            }
96          });
97        editMenu.add(brightenItem);
98
99        JMenuItem edgeDetectItem = new JMenuItem("Edge detect");
100       edgeDetectItem.addActionListener(new ActionListener()
101         {
102           public void actionPerformed(ActionEvent event)
```

```
103         {
104           float[] elements =
105              { 0.0f, -1.0f, 0.0f, -1.0f, 4.f,
106                -1.0f, 0.0f, -1.0f, 0.0f };
107           convolve(elements);
108         }
109       });
110     editMenu.add(edgeDetectItem);
111
112     JMenuItem negativeItem = new JMenuItem("Negative");
113     negativeItem.addActionListener(new ActionListener()
114       {
115         public void actionPerformed(ActionEvent event)
116         {
117           short[] negative = new short[256 * 1];
118           for (int i = 0; i < 256; i++)
119             negative[i] = (short) (255 - i);
120           ShortLookupTable table = new ShortLookupTable(0, negative);
121           LookupOp op = new LookupOp(table, null);
122           filter(op);
123         }
124       });
125     editMenu.add(negativeItem);
126
127     JMenuItem rotateItem = new JMenuItem("Rotate");
128     rotateItem.addActionListener(new ActionListener()
129       {
130         public void actionPerformed(ActionEvent event)
131         {
132           if (image == null) return;
133             AffineTransform transform =
134                AffineTransform.getRotateInstance(Math.toRadians(5),
135                image.getWidth() / 2, image.getHeight() / 2);
136             AffineTransformOp op = new AffineTransformOp(transform,
137                AffineTransformOp.TYPE_BICUBIC);
138             filter(op);
139         }
140       });
141     editMenu.add(rotateItem);
142
143     JMenuBar menuBar = new JMenuBar();
144     menuBar.add(fileMenu);
145     menuBar.add(editMenu);
146     setJMenuBar(menuBar);
147   }
148
149   /**
150    * Открыть файл и загрузить изображение
151   */
152   public void openFile()
153   {
154     JFileChooser chooser = new JFileChooser(".");
155     chooser.setCurrentDirectory(
156        new File(getClass().getPackage().getName()));
157     String[] extensions = ImageIO.getReaderFileSuffixes();
158     chooser.setFileFilter(new FileNameExtensionFilter(
159        "Image files", extensions));
160     int r = chooser.showOpenDialog(this);
161     if (r != JFileChooser.APPROVE_OPTION) return;
```

```
162     try
163     {
164       Image img = ImageIO.read(chooser.getSelectedFile());
165       image = new BufferedImage(img.getWidth(null), img.getHeight(null),
166       BufferedImage.TYPE_INT_RGB);
167       image.getGraphics().drawImage(img, 0, 0, null);
168     }
169     catch (IOException e)
170     {
171       JOptionPane.showMessageDialog(this, e);
172     }
173     repaint();
174   }
175
176   /**
177    * Применить фильтр и перерисовать
178    * @param op Выполняемая операция преобразования
179   */
180   private void filter(BufferedImageOp op)
181   {
182     if (image == null) return;
183     image = op.filter(image, null);
184     repaint();
185   }
186
187   /**
188    * Выполнить свертку и перерисовать
189    * @param elements Ядро свертки (массив из 9 элементов матрицы)
190   */
191   private void convolve(float[] elements)
192   {
193     Kernel kernel = new Kernel(3, 3, elements);
194     ConvolveOp op = new ConvolveOp(kernel);
195     filter(op);
196   }
197 }
```

java.awt.image.BufferedImageOp 1.2

- `BufferedImage filter(BufferedImage` *`source`*`, BufferedImage` *`dest`*`)`

  Выполняет операцию над исходным изображением *`source`* и сохраняет (а также возвращает) результат в виде изображения *`dest`*. Если в качестве параметра *`dest`* указано пустое значение `null`, то создается новое целевое изображение, которое затем возвращается.

java.awt.image.AffineTransformOp 1.2

- `AffineTransformOp(AffineTransform` *`t`*`, int` *`interpolationType`*`)`

  Создает объект для операции аффинного преобразования. В качестве алгоритма интерполяции может быть указано значение одной из следующих констант: `TYPE_BILINEAR`, `TYPE_BICUBIC` или `TYPE_NEAREST_NEIGHBOR`.

java.awt.image.RescaleOp 1.2

- `RescaleOp(float a, float b, RenderingHints hints)`
- `RescaleOp(float[] as, float[] bs, RenderingHints hints)`

Создают объект для следующей операции изменения масштаба: $x_{new} = a \cdot x + b$. При использовании первого конструктора все составляющие цвета (кроме составляющей прозрачности из альфа-канала) масштабируются с одинаковыми коэффициентами. А при использовании второго конструктора значения предоставляются для каждой составляющей цвета в отдельности, но не затрагивая составляющую прозрачности из альфа-канала, или же для составляющих как цвета, так и прозрачности из альфа-канала.

java.awt.image.LookupOp 1.2

- `LookupOp(LookupTable table, RenderingHints hints)`

Создает объект операции поиска для указанной таблицы поиска.

java.awt.image.ByteLookupTable 1.2

- `ByteLookupTable(int offset, byte[] data)`
- `ByteLookupTable(int offset, byte[][] data)`

Создают таблицу поиска для преобразования значений типа `byte`. Значение смещения вычитается из входных данных до преобразования. Значения в первом конструкторе применяются ко всем составляющим цвета, но не к составляющей прозрачности из альфа-канала. При использовании второго конструктора значения предоставляются для каждой составляющей цвета в отдельности, но не затрагивая составляющую прозрачности из альфа-канала, или же для составляющих как цвета, так и прозрачности из альфа-канала.

java.awt.image.ShortLookupTable 1.2

- `ShortLookupTable(int offset, short[] data)`
- `ShortLookupTable(int offset, short[][] data)`

Создают таблицу поиска для преобразования значений типа `short`. Значение смещения вычитается из входных данных до преобразования. Значения в первом конструкторе применяются ко всем составляющим цвета, но не к составляющей прозрачности из альфа-канала. При использовании второго конструктора значения предоставляются для каждой составляющей цвета в отдельности, но не затрагивая составляющую прозрачности из альфа-канала, или же для составляющих как цвета, так и прозрачности из альфа-канала.

java.awt.image.ConvolveOp 1.2

- `ConvolveOp(Kernel kernel)`
- `ConvolveOp(Kernel kernel, int edgeCondition, RenderingHints hints)`

  Создают объект для операции свертки. В качестве параметра *edgeCondition* может быть указано значение одной из следующих констант: `EDGE_NO_OP` или `EDGE_ZERO_FILL`. С этими значениями следует обращаться особенно внимательно, поскольку у них нет достаточного количество соседних значений для вычисления свертки. По умолчанию устанавливается значение константы `EDGE_ZERO_FILL`.

java.awt.image.Kernel 1.2

- `Kernel(int width, int height, float[] matrixElements)`

  Создает ядро свертки для указанной матрицы.

Вывод изображений на печать

В исходной версии JDK вообще не поддерживался вывод на печать. Например, нельзя было печатать из аплетов, а для печати из приложений необходимо было пользоваться дополнительными библиотеками независимых поставщиков. Впервые средства для организации вывода на печать появились в версии JDK 1.1, но с их помощью можно было выводить очень простые документы довольно низкого качества. Разработчикам веб-браузеров была предоставлена специальная модель печати, позволявшая выводить окно аплета в таком виде, в каком оно отображалось на веб-странице (чем они так и не воспользовались).

В версии Java SE 1.2 были заложены основы надежной модели печати, полностью интегрированной со средствами двухмерной графики. А в версии Java SE 1.4 эта модель была существенно усовершенствована и дополнена возможностями управления параметрами печатающих устройств и организации заданий потоковой печати на стороне сервера.

В этом разделе показано, каким образом изображения печатаются на одной странице бумаги, как организовать многостраничную печать, а также демонстрируются возможности модели воспроизведения изображений в прикладном интерфейсе Java 2D API и средства для создания окна предварительного просмотра печати.

Вывод двухмерной графики на печать

В данном разделе рассматривается одна из наиболее распространенных задач, а именно печать двухмерной графики. Разумеется, изображение может содержать фрагменты текста, отформатированного различными шрифтами, или даже полностью состоять из текста. Для вывода двухмерной графики на печать необходимо выполнить следующие действия:

- Предоставить объект, класс которого реализует интерфейс `Printable`.
- Запустить задание на печать.

В интерфейсе `Printable` объявлен единственный метод `print()`, как показано ниже.

```
int print(Graphics g, PageFormat format, int page)
```

Этот метод вызывается всякий раз, когда механизм печати форматирует страницу. Прикладной код рисует текст и изображение, которые должны быть напечатаны в заданном графическом контексте `g`. Формат страницы (параметр `format`) определяет формат бумаги и поля для печати, а номер страницы (параметр `page`) служит для выбора воспроизводимой страницы.

Для запуска задания на печать служит класс `PrinterJob`. Сначала для получения объекта задания на печать вызывается статический метод `getPrinterJob()`, а затем с помощью метода `setPrintable()` указывается печатаемый объект типа `Printable`:

```
Printable canvas = . . .;
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(canvas);
```


ВНИМАНИЕ! Не путайте класс `PrinterJob` с устаревшим классом `PrintJob`, который управлял процессом печати в версии JDK 1.1.

Прежде чем запускать задание на печать, следует вызвать метод `printDialog()`, чтобы открыть диалоговое окно, приведенное на рис. 7.32. В этом окне пользователю предоставляется возможность выбрать устройство для вывода на печать (если доступно несколько печатающих устройств), указать диапазон печатаемых страниц и сделать прочие настройки печати.

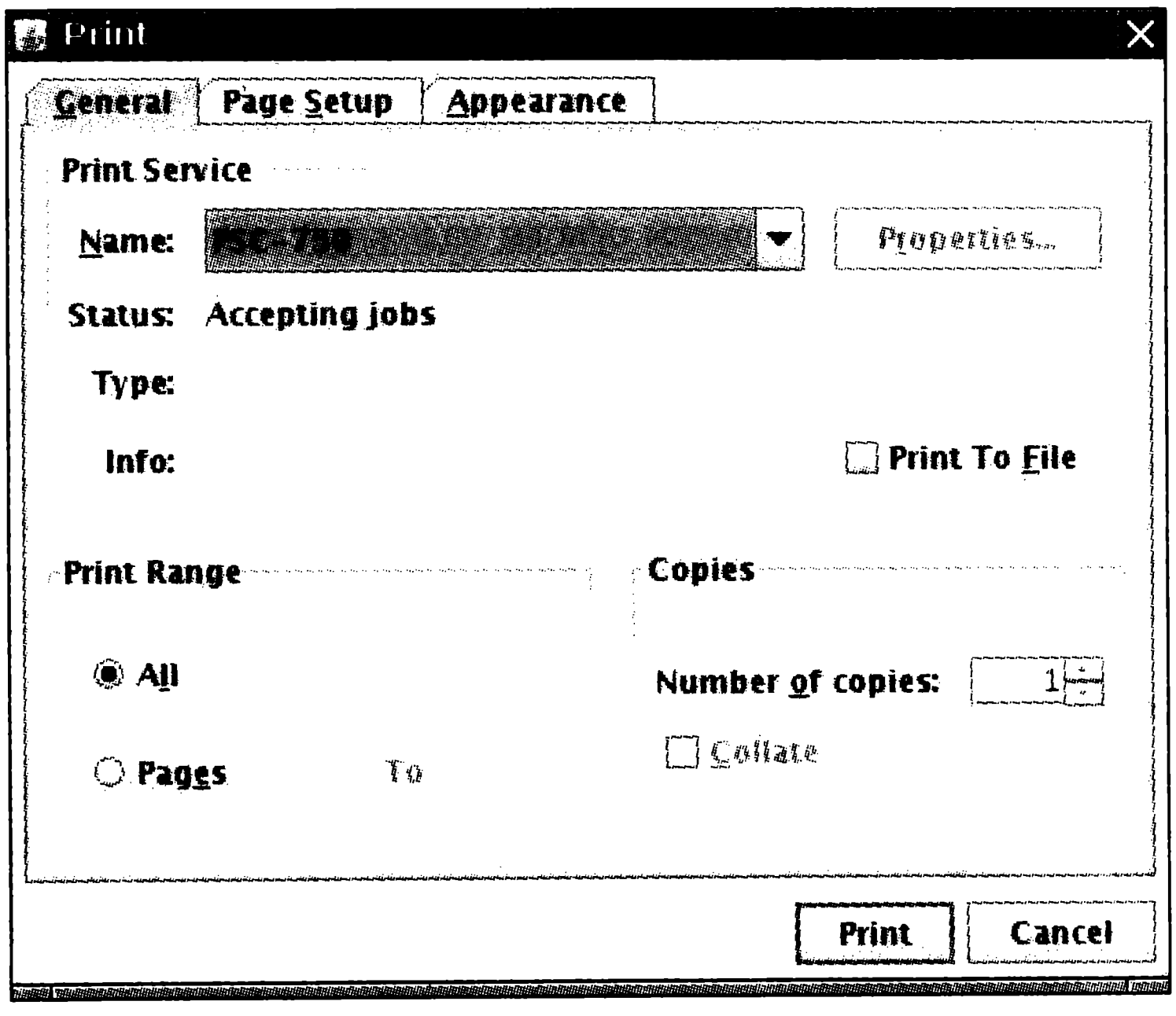


Рис. 7.32. Вид диалогового окна для настройки печати независимо от используемой платформы

Все параметры печатающего устройства обычно собираются в экземпляре класса, реализующего интерфейс `PrintRequestAttributeSet`, например класса `HashPrintRequestAttributeSet`:

```
HashPrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
```

Атрибуты печати должны быть переданы методу `printDialog()` в виде объекта `attributes`. Метод `printDialog()` возвращает логическое значение `true`, если пользователь подтверждает выбор параметров печати, или логическое значение `false`, если пользователь отменяет внесенные изменения. В первом случае вызывается метод `print()` из класса `PrinterJob` для запуска задания на печать. Он может генерировать исключение типа `PrinterException`. В приведенном ниже фрагменте кода показано, каким образом вывод на печать организуется в общих чертах.

```
if (job.printDialog(attributes))
{
  try
  {
    job.print(attributes);
  }
  catch (PrinterException exception)
  {
    . . .
  }
}
```

НА ЗАМЕТКУ! До выпуска версии JDK 1.4 в механизме печати Java использовались диалоговые окна настройки печати, ориентированные именно на ту платформу, на которой работал пользователь. Для отображения платформенно-ориентированного диалогового окна настройки печати следует вызвать метод `printDialog()` без указания параметров. (В этом случае пользовательские установки нельзя объединить в набор атрибутов печати.)

Во время печати метод `print()` из класса `PrinterJob` неоднократно вызывает метод `print()` для объекта типа `Printable`, связанного с текущим заданием на печать. А поскольку в задании на печать неизвестно количество выводимых страниц, то метод `print()` вызывается повторно, если он возвращает значение константы `Printable.PAGE_EXISTS`. Печать завершится лишь тогда, когда метод `print()` возвратит значение константы `Printable.NO_SUCH_PAGE`.

ВНИМАНИЕ! Отсчет номеров страниц, передаваемых методу `print()`, начинается с нуля.

Таким образом, задание на печать не может определить количество страниц до тех пор, пока печать не будет завершена. Поэтому в диалоговом окне настройки печати не может быть представлен фактический диапазон номеров страниц, а вместо него выводится сообщение `"Pages 1 to 1"` (Страниц от 1 до 1). В следующем разделе будет показано, как этот недостаток устраняется с помощью объекта типа `Book`, предоставляемого для задания на печать.

Итак, метод `print()` может повторно вызываться для объекта `Printable` в задании на печать, чтобы напечатать *одну и ту же* страницу. Поэтому вместо подсчета количества страниц в методе `print()` лучше полагаться на номер страницы. Возможность

многократного обращения к одной и той же странице предназначена для обслуживания некоторых печатающих устройств, особенно матричных и струйных, которые используют *полосовой* способ печати. Они печатают сначала одну полосу, продвигают бумагу и затем печатают следующую полосу. Полосовой способ печати может использоваться даже на лазерных принтерах, которые обычно печатают сразу всю страницу. Это дает возможность управлять размером файла буферизации печати.

Если для вывода полосы в задании на печать требуется объект типа `Printable`, то следует указать область отсечения графического контекста печатаемой полосой и вызвать метод `print()`. Таким образом, все операции рисования будут выполняться только на данной полосе, и только в пределах ее прямоугольной области будут воспроизводиться рисуемые графические элементы. Методу `print()` совсем не обязательно знать об этих действиях, при условии, что он *не* затрагивает область отсечения.

ВНИМАНИЕ! Объект типа `Graphics`, возвращаемый методом `print()`, отсекается также по полям печатной страницы. Хотя рисовать на полях страницы можно, если переместить область отсечения. Но в графическом контексте печатающего устройства нужно строго соблюдать заданную область отсечения. Для дальнейшего ограничения области отсечения следует вызывать метод `clip()`, а не `setClip()`. Если же требуется временно удалить область отсечения, то в начале разрабатываемого метода `print()` придется вызвать метод `getClip()`, чтобы получить эту область для ее последующего восстановления.

Параметр типа `PageFormat` метода `print()` содержит сведения о печатаемой странице. Методы `getWidth()` и `getHeight()` возвращают формат бумаги, измеряемый в *пунктах*. Один пункт равен 1/72 дюйма, а один дюйм составляет 25,4 миллиметра.

Пункты являются общепринятой единицей измерения в полиграфической промышленности США. Эта же единица применяется по умолчанию для всех графических контекстов печати. В этом можно убедиться на примере программы, исходный код которой приведен в конце данного раздела. Эта программа печатает две строки текста, которые находятся на расстоянии 72 пунктов друг от друга. Запустите эту программу, измерьте расстояние между напечатанными строками и убедитесь в том, что оно равно точно 1 дюйму, или 25,4 мм.

Методы `getWidth()` и `getHeight()` из класса `PageFormat` возвращают полную ширину и высоту страницы. Но печатать можно не на всей странице. Обычно пользователи самостоятельно задают размеры полей печатаемой страницы, но даже если они этого не сделают явно, то все равно поля будут определены. Дело в том, что печатающему устройству нужно каким-то образом удерживать листы бумаги, на которых производится печать, и поэтому по краям остаются небольшие непечатаемые участки.

Методы `getImageableWidth()` и `getImageableHeight()` возвращают ширину и высоту области, доступной для печати. Но поля не обязательно должны быть симметричными, поэтому нужно также знать координаты верхнего левого угла области, доступной для печати, которая условно показана на рис. 7.33. Эти координаты можно получить с помощью методов `getImageableX()` и `getImageableY()`.

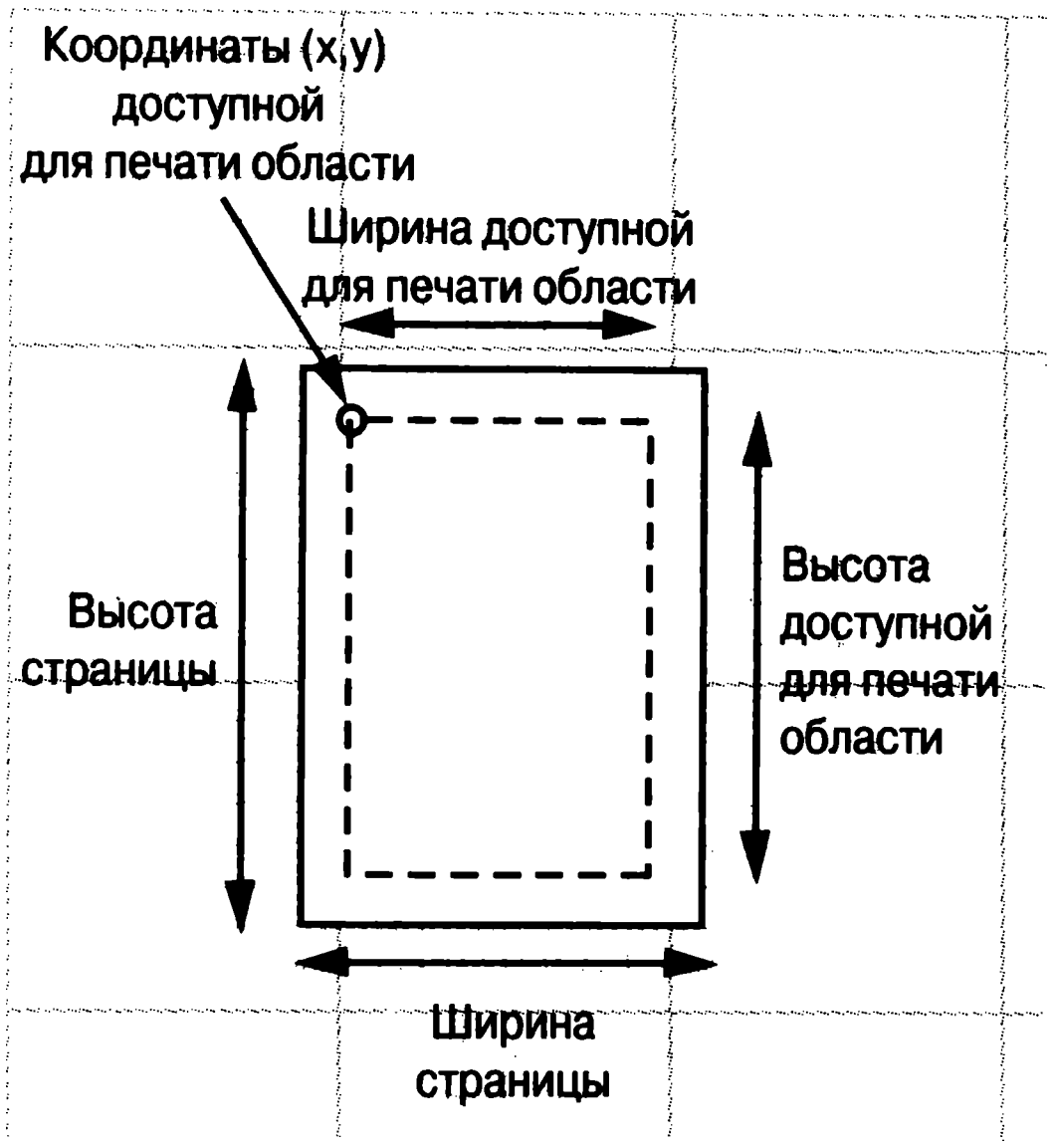


Рис. 7.33. Размеры страницы и области, доступной для печати

СОВЕТ. Графический контекст, который передается методу `print()`, обрезается по полям печатаемой страницы, тем не менее начало системы координат остается по-прежнему в верхнем левом углу страницы. Поэтому начало координат имеет смысл перенести в левый верхний угол области, доступной для печати. Для этого разрабатываемый метод `print()` нужно начать со следующей строки кода:

```
g.translate(pageFormat.getImageableX(), pageFormat.getImageableY());
```

Если же требуется предоставить пользователю возможность определять границы страницы и задавать книжную или альбомную ориентацию, не устанавливая других атрибутов печати, то для этого можно вызвать метод `pageDialog()` из класса `PrinterJob` следующим образом:

```
PageFormat format = job.pageDialog(attributes);
```

НА ЗАМЕТКУ! На одной из вкладок диалогового окна настройки печати содержатся параметры печатаемой страницы (рис. 7.34), которые пользователь может просмотреть и поправить непосредственно перед печатью. Это особенно полезно, если в программе реализован принцип WYSIWIG, когда на отпечатке получается именно то, что отображается на экране. Метод `pageDialog()` возвращает объект типа `PageFormat` со значениями параметров печати, задаваемых пользователем.

В примере программы, исходный код которой приведен в листингах 7.10 и 7.11, показано, каким образом один и тот же ряд фигур выводится на экран и на печать. Класс `PrintPanel` расширяет класс `JPanel` и реализует интерфейс `Printable`, а его методы `paintComponent()` и `print()` вызывают один и тот же метод непосредственно для рисования изображения, как выделено ниже полужирным.

```
class PrintPanel extends JPanel implements Printable
{
  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    drawPage(g2);
  }

  public int print(Graphics g, PageFormat pf, int page)
    throws PrinterException
  {
    if (page >= 1) return Printable.NO_SUCH_PAGE;
    Graphics2D g2 = (Graphics2D) g;
    g2.translate(pf.getImageableX(), pf.getImageableY());
    drawPage(g2);
    return Printable.PAGE_EXISTS;
  }
  public void drawPage(Graphics2D g2)
  {
    // здесь следует общий код для рисования
    . . .
  }
  . . .
}
```

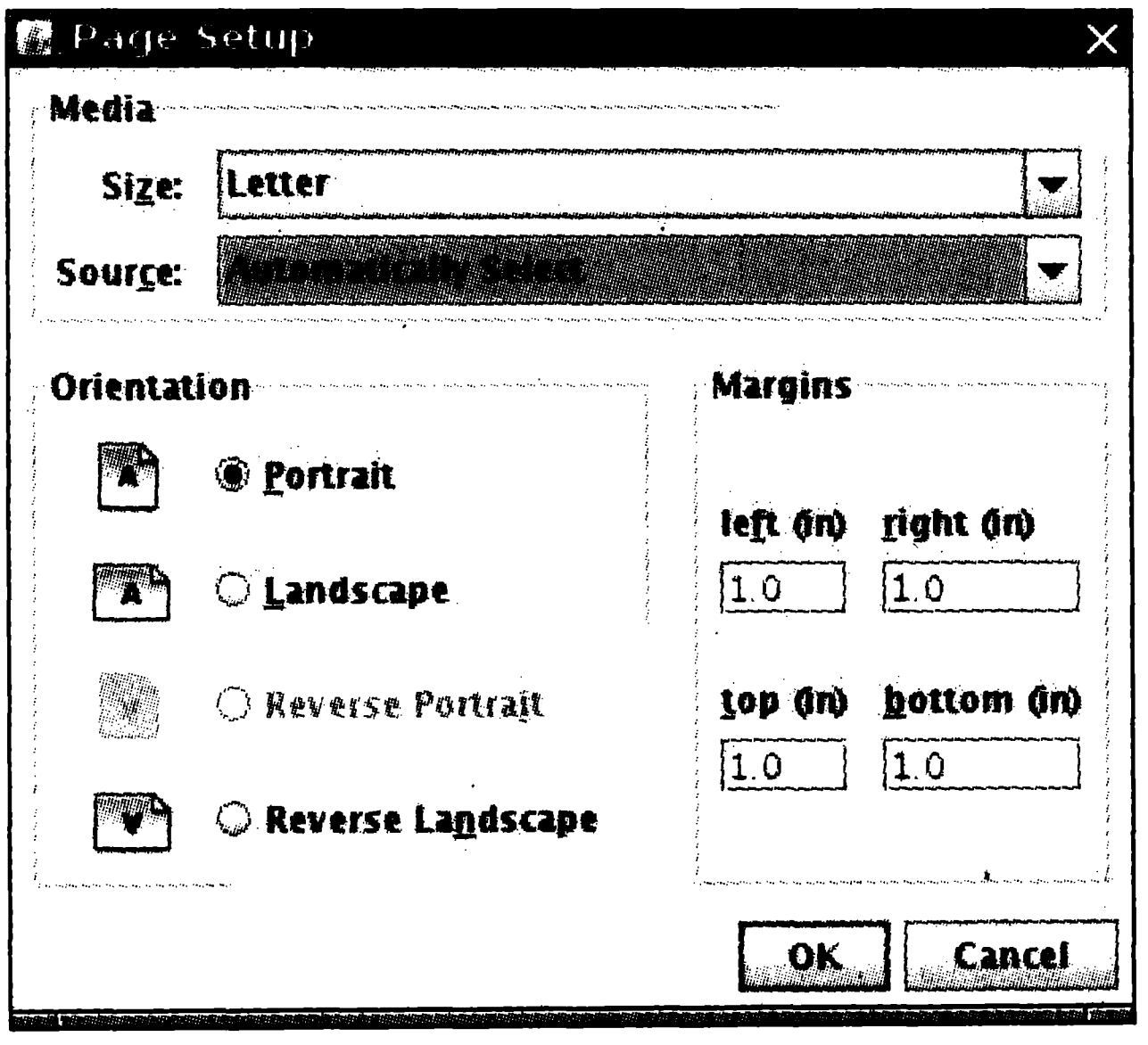


Рис. 7.34. Вид диалогового окна для настройки печатаемой страницы независимо от используемой платформы

В рассматриваемом здесь примере для демонстрации вывода данных на печать используется то же самое изображение, что и на рис. 7.20. В этом изображении контуры символов, составляющих строку `"Hello, World"`, используются в качестве области отсечения для линий штрихового рисунка.

Для запуска задания на печать достаточно щелкнуть на кнопке Print (Печать), а для открытия диалогового окна настройки печати — на кнопке Page setup (Параметры страницы). Соответствующий исходный код приведен в листинге 7.10.

НА ЗАМЕТКУ! Для того чтобы отобразить платформенно-ориентированное диалоговое окно параметров печатаемой страницы, следует передать методу `pageDialog()` объект типа `PageFormat`. Этот метод создает копию объекта, изменяет ее в соответствии с пользовательскими установками и возвращает обратно, как показано ниже.

```
PageFormat defaultFormat = printJob.defaultPage();
PageFormat selectedFormat = printJob.pageDialog(defaultFormat);
```

Листинг 7.10. Исходный код из файла `print/PrintTestFrame.java`

```
1  package print;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.awt.print.*;
6  import javax.print.attribute.*;
7  import javax.swing.*;
8
9  /**
10  * В этом фрейме отображается панель с двухмерной графикой и
11  * кнопками для печати графики и установки формата страницы
12 */
13 public class PrintTestFrame extends JFrame
14 {
15   private PrintComponent canvas;
16   private PrintRequestAttributeSet attributes;
17
18   public PrintTestFrame()
19   {
20     canvas = new PrintComponent();
21     add(canvas, BorderLayout.CENTER);
22
23     attributes = new HashPrintRequestAttributeSet();
24
25     JPanel buttonPanel = new JPanel();
26     JButton printButton = new JButton("Print");
27     buttonPanel.add(printButton);
28     printButton.addActionListener(new ActionListener()
29       {
30         public void actionPerformed(ActionEvent event)
31         {
32           try
33           {
34             PrinterJob job = PrinterJob.getPrinterJob();
35             job.setPrintable(canvas);
36             if (job.printDialog(attributes)) job.print(attributes);
37           }
38           catch (PrinterException e)
39           {
40             JOptionPane.showMessageDialog(PrintTestFrame.this, e);
41           }
42         }
```

```
43        });
44
45     JButton pageSetupButton = new JButton("Page setup");
46     buttonPanel.add(pageSetupButton);
47     pageSetupButton.addActionListener(new ActionListener()
48        {
49          public void actionPerformed(ActionEvent event)
50          {
51             PrinterJob job = PrinterJob.getPrinterJob();
52             job.pageDialog(attributes);
53          }
54        });
55
56     add(buttonPanel, BorderLayout.NORTH);
57     pack();
58   }
59 }
```

Листинг 7.11. Исходный код из файла `print/PrintComponent.java`

```
1  package print;
2
3  import java.awt.*;
4  import java.awt.font.*;
5  import java.awt.geom.*;
6  import java.awt.print.*;
7  import javax.swing.*;
8  /**
9   * Этот компонент формирует двухмерную графику для
10  * вывода на экран и на печать
11 */
12 public class PrintComponent extends JComponent implements Printable
13 {
14   private static final int DEFAULT_WIDTH = 300;
15   private static final int DEFAULT_HEIGHT = 300;
16
17   public void paintComponent(Graphics g)
18   {
19     Graphics2D g2 = (Graphics2D) g;
20     drawPage(g2);
21   }
22
23   public int print(Graphics g, PageFormat pf, int page)
24      throws PrinterException
25   {
26     if (page >= 1) return Printable.NO_SUCH_PAGE;
27     Graphics2D g2 = (Graphics2D) g;
28     g2.translate(pf.getImageableX(), pf.getImageableY());
29     g2.draw(new Rectangle2D.Double(0, 0, pf.getImageableWidth(),
30        pf.getImageableHeight()));
31
32     drawPage(g2);
33     return Printable.PAGE_EXISTS;
34   }
35   /**
36    * Этот метод рисует страницу в графическом контексте экрана и принтера
37    * @param g2 Графический контекст
38   */
```

```
39   public void drawPage(Graphics2D g2)
40   {
41     FontRenderContext context = g2.getFontRenderContext();
42     Font f = new Font("Serif", Font.PLAIN, 72);
43     GeneralPath clipShape = new GeneralPath();
44
45     TextLayout layout = new TextLayout("Hello", f, context);
46     AffineTransform transform =
47        AffineTransform.getTranslateInstance(0, 72);
48     Shape outline = layout.getOutline(transform);
49     clipShape.append(outline, false);
50
51     layout = new TextLayout("World", f, context);
52     transform = AffineTransform.getTranslateInstance(0, 144);
53     outline = layout.getOutline(transform);
54     clipShape.append(outline, false);
55
56     g2.draw(clipShape);
57     g2.clip(clipShape);
58
59     final int NLINES = 50;
60     Point2D p = new Point2D.Double(0, 0);
61     for (int i = 0; i < NLINES; i++)
62     {
63       double x = (2 * getWidth() * i) / NLINES;
64       double y = (2 * getHeight() * (NLINES - 1 - i)) / NLINES;
65       Point2D q = new Point2D.Double(x, y);
66       g2.draw(new Line2D.Double(p, q));
67     }
68   }
69
70   public Dimension getPreferredSize()
71      { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
72 }
```

java.awt.print.Printable 1.2

- `int print(Graphics g, PageFormat format, int pageNumber)`

  Воспроизводит страницу и возвращает значение `PAGE_EXISTS` или `NO_SUCH_PAGE`.

Параметры:	*g*	Графический контекст, в котором воспроизводится страница
	format	Формат воспроизводимой страницы
	pageNumber	Номер требуемой страницы

java.awt.print.PrinterJob 1.2

- `static PrinterJob getPrinterJob()`

  Возвращает объект задания на печать.

- `PageFormat defaultPage()`

  Возвращает установленный по умолчанию формат страницы для данного печатающего устройства.

- `boolean printDialog(PrintRequestAttributeSet` *`attributes`*`)`
- `boolean printDialog()`

  Открывают диалоговое окно, позволяющее пользователю выбирать страницы для печати и настраивать ее параметры. Первый метод отображает платформенно-независимое диалоговое окно, а второй — платформенно-ориентированное окно. Оба метода изменяют объект *`attributes`* с учетом пользовательских настроек и возвращают логическое значение `true`, если пользователь подтвердит установленные параметры настройки печати.

- `PageFormat pageDialog(PrintRequestAttributeSet` *`attributes`*`)`
- `PageFormat pageDialog(PageFormat` *`defaults`*`)`

  Выводят диалоговое окно настройки страницы. Первый из указанных методов отображает платформенно-независимое диалоговое окно, а второй — платформенно-ориентированное окно. Оба метода возвращают объект типа `PageFormat`, определяющий формат, указанный пользователем в диалоговом окне. Первый метод видоизменяет объект *`attributes`* таким образом, чтобы тот отражал пользовательские установки. А второй метод не видоизменяет объект *`defaults`*.

- `void setPrintable(Printable` *`p`*`)`
- `void setPrintable(Printable` *`p`*`, PageFormat` *`format`*`)`

  Устанавливают объект типа `Printable` и необязательный формат страницы для данного задания на печать.

- `void print()`
- `void print(PrintRequestAttributeSet` *`attributes`*`)`

  Выводят на печать текущий объект типа `Printable`, повторно вызывая его метод `print()` и отправляя воспроизводимые страницы на печатающее устройство до тех пор, пока не будут напечатаны все страницы.

java.awt.print.PageFormat 1.2

- `double getWidth()`
- `double getHeight()`

  Возвращают ширину и высоту страницы.

- `double getImageableWidth()`
- `double getImageableHeight()`

  Возвращают ширину и высоту доступной для печати области страницы.

- `double getImageableX()`
- `double getImageableY()`

  Возвращают координаты верхнего левого угла доступной для печати области страницы.

- `int getOrientation()`

  Возвращают значение одной из следующих констант, определяющих ориентацию страницы: `PORTRAIT`, `LANDSCAPE` или `REVERSE_LANDSCAPE`. Ориентация "прозрачна" для программирования, поскольку она автоматически учитывается в установках формата страниц и графического контекста.

Многостраничная печать

На практике объекты типа `Printable` нежелательно передавать заданию на печать без предварительной обработки. Сначала следует получить экземпляр класса, который реализует интерфейс `Pageable`. На платформе Java для этой цели предоставляется класс `Book`, реализующий книгу, состоящую из разделов, представленных объектами типа `Printable`. Для составления книги сначала вводятся объекты типа `Printable` в ее разделы и организуется нумерация страниц, как показано ниже.

```
Book book = new Book();
Printable coverPage = . . .;
Printable bodyPages = . . .;
book.append(coverPage, pageFormat); // присоединить одну страницу
book.append(bodyPages, pageFormat, pageCount);
```

Затем вызывается метод `setPageable()`, чтобы передать книгу в виде объекта типа `Book` заданию на печать:

```
printJob.setPageable(book);
```

Теперь заданию на печать точно известно количество печатаемых страниц, а в диалоговом окне настройки печати отображаются их номера, чтобы пользователь мог выбрать весь диапазон печатаемых страниц или только его часть.

ВНИМАНИЕ! Когда из задания на печать вызывается метод `print()` для объектов разделов типа `Printable`, ему передается текущий номер страницы, отсчитываемый в пределах *книги*, а *не* текущего раздела. Это очень неудобно, поскольку в каждом разделе должны храниться данные о количестве страниц в предыдущих разделах.

Для программирования наибольшую сложность при использовании класса `Book` представляет подсчет и сохранение данных о количестве страниц в каждом разделе во время печати. В объекте типа `Printable` нужно создать *алгоритм обработки компоновки* печатаемой страницы. Обработку компоновки страницы нужно выполнить перед началом ее печати, чтобы определить количество печатаемых страниц и их разрывы. Сведения о компоновке страницы можно сохранить для дальнейшего применения во время печати. Необходимо также учитывать, что пользователь может изменить формат страницы. В таком случае придется повторно рассчитать компоновку страницы, даже если печатаемые данные не изменились.

В примере программы из листинга 7.13 показано, каким образом организуется многостраничная печать. Эта программа выводит крупный заголовок (так называемый *баннер*), состоящий из строки символов очень большого размера, и поэтому они располагаются на нескольких страницах (рис. 7.35). После распечатки всех страниц поля обрезаются, а страницы склеиваются вместе.

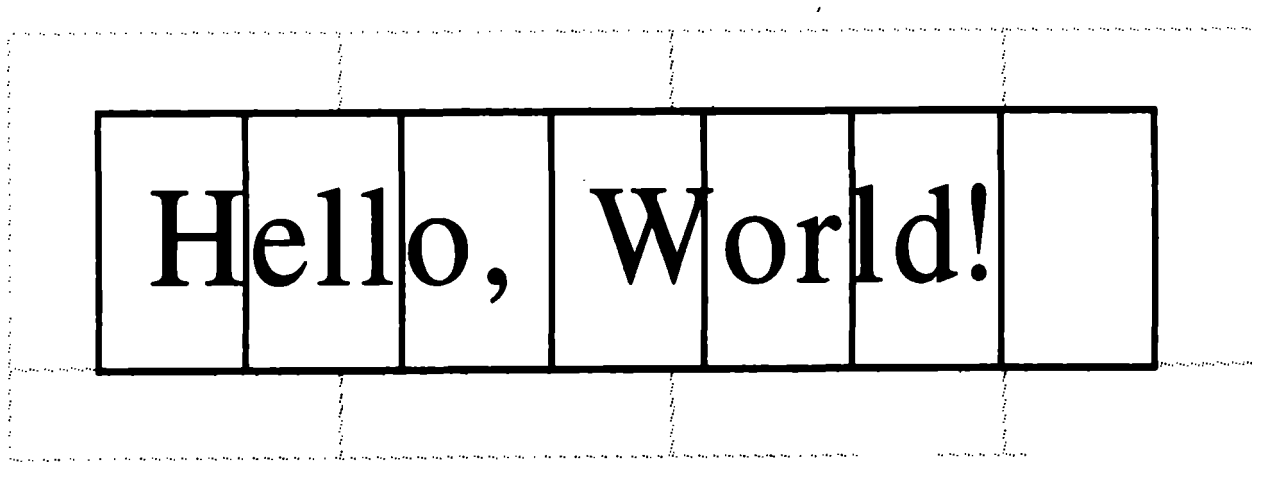


Рис. 7.35. Баннер

Метод `layoutPages()` из класса `Banner` рассчитывает компоновку страницы. Сначала компонуется строка сообщения, отформатированная шрифтом **72** пункта. Затем высота результирующей строки вычисляется и сравнивается с высотой печатаемой области страницы. На основании этих двух измерений определяется масштабный коэффициент. При выводе на печать строка увеличивается на этот масштабный коэффициент.

ВНИМАНИЕ! Для точного размещения на странице печатаемой информации требуется доступ к графическому контексту печатающего устройства. К сожалению, получить доступ к графическому контексту нельзя до тех пор, пока не начнется сама печать. В программе, рассматриваемой здесь в качестве примера, все требующиеся действия выполняются в экранном графическом контексте в надежде, что типографские параметры шрифтов на экране и на печатной странице совпадут.

В методе `getPageCount()` из класса `Banner` сначала вызывается метод определения компоновки страницы. Затем ширина символьной строки увеличивается в масштабе и делится на ширину доступной для печати области отдельной страницы. Частное от этого деления, округленное до следующего целого значения, определяет искомое количество страниц.

Из всего сказанного выше можно сделать вывод, что напечатать баннер не так-то просто, поскольку отдельные части символа могут располагаться на разных страницах. Но благодаря эффективным средствам Java 2D API эта задача решается очень просто. Для получения определенной страницы вызывается метод `translate()` из класса `Graphics2D`, который смещает верхний левый угол печатаемой символьной строки на заданное расстояние влево. Затем устанавливается прямоугольная область отсечения с размерами текущей страницы (рис. 7.36). И наконец, масштаб графического контекста изменяется по масштабному коэффициенту, вычисленному методом компоновки страницы.

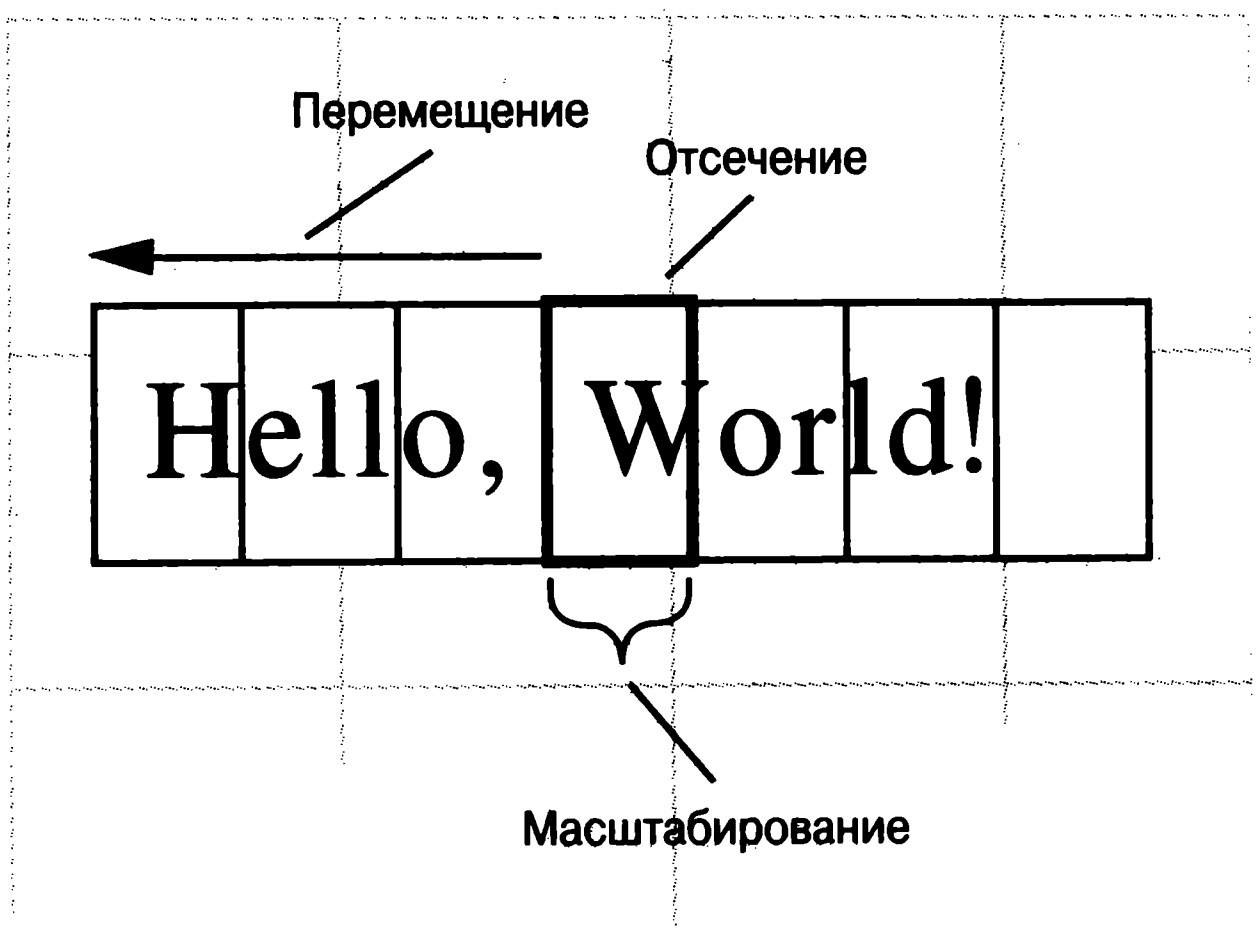


Рис. 7.36. Алгоритм постраничной печати баннера

Рассматриваемый здесь пример демонстрирует возможности функций преобразования, благодаря которым код для рисования остается простым. Как видите, преобразования выполняют всю работу по размещению изображения в соответствующем

месте и отсечению той его части, которая выходит за пределы доступной для печати области страницы. В следующем разделе рассматривается еще один наглядный пример применения преобразований для организации предварительного просмотра печатаемой страницы.

Предварительный просмотр печати

В большинстве прикладных программ профессионального уровня предусмотрен механизм предварительного просмотра, который позволяет оценить внешний вид страницы на экране компьютера перед запуском задания на печать, чтобы не тратить зря бумагу на распечатку неверно скомпонованных страниц. В классах платформы Java, предназначенных для вывода данных на печать, не предусмотрено стандартного диалогового окна предварительного просмотра печати. Тем не менее окно, аналогичное приведенному на рис. 7.37, можно без особого труда создать самостоятельно. В этом разделе рассматривается один из способов создания такого окна. В частности, класс `PrintPreviewDialog` из листинга 7.14 можно рассматривать как полностью обобщенный, поскольку он обеспечивает предварительный просмотр внешнего вида любого вида печати.

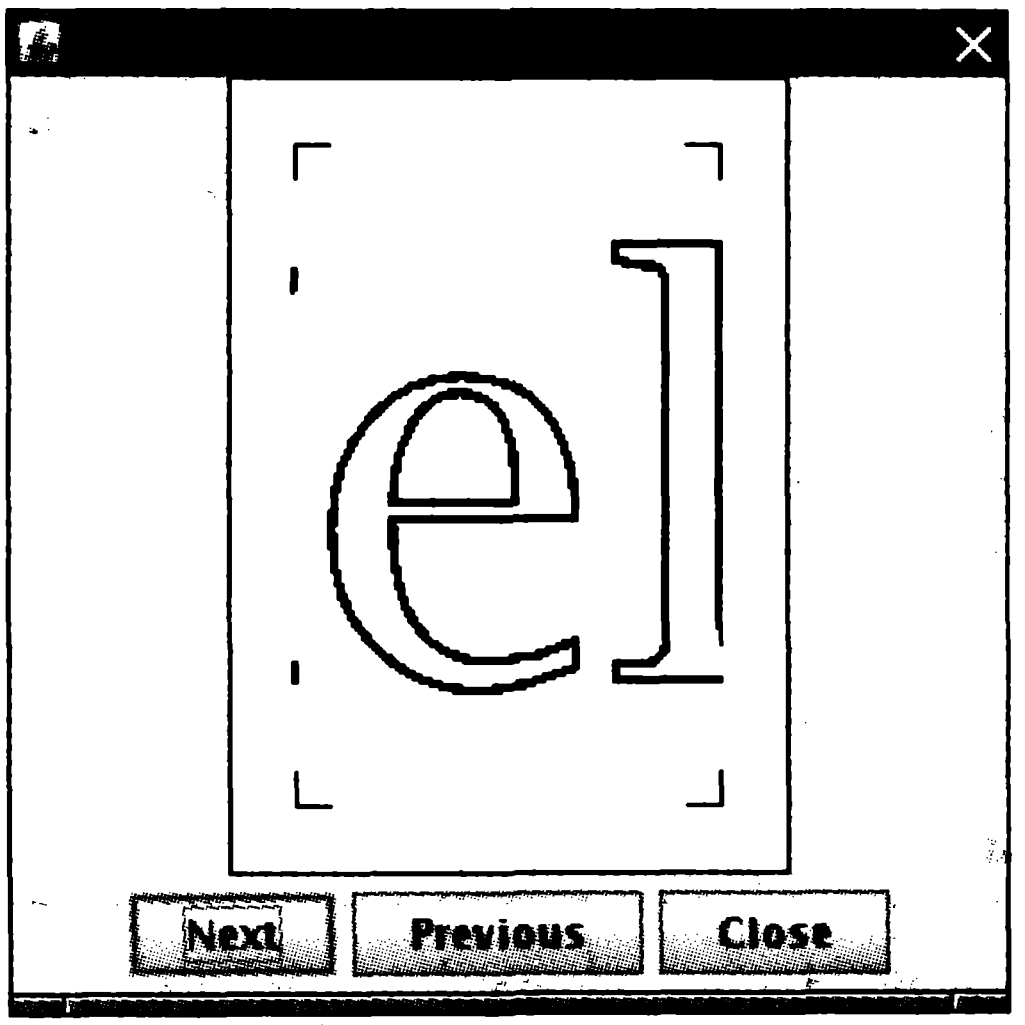


Рис. 7.37. Диалоговое окно предварительного просмотра печатаемой страницы с баннером

Для создания диалогового окна предварительного просмотра печати конструируется объект типа `PrintPreviewDialog` из объектов типа `Printable` или `Book`, а также объекта типа `PageFormat`. А для оформления внешнего вида диалогового окна служит класс `PrintPreviewCanvas` из листинга 7.15. При активизации кнопок Next и Previous, предназначенных для перехода к следующей и предыдущей просматриваемым страницам, из метода `paintComponent()` вызывается метод `print()` для объекта типа `Printable` запрашиваемой страницы.

Обычно метод `print()` отображает содержимое печатаемой страницы в графическом контексте печатающего устройства. Но в данном случае используется графический контекст экрана, масштабированный таким образом, чтобы печатаемая

страница полностью вмещалась в небольшой прямоугольной области экрана, как показано ниже.

```
float xoff = . . .; // левая сторона страницы
float yoff = . . .; // верхняя сторона страницы
float scale = . . .; // масштабный коэффициент для подгонки печатаемой
                     // страницы по размерам прямоугольной области экрана
g2.translate(xoff, yoff);
g2.scale(scale, scale);
Printable printable = book.getPrintable(currentPage);
printable.print(g2, pageFormat, currentPage);
```

В данном случае метод `print()` выполняет те же самые функции, что и при выводе страницы на печать, просто рисуя в заданном графическом контексте страницу, которая отображается на экране в уменьшенном виде для предварительного просмотра. Данный пример наглядно демонстрирует возможности модели воспроизведения изображений Java 2D.

В листинге 7.12 представлен исходный код примера программы для вывода баннера на печать с диалоговым окном предварительного просмотра. Эта программа позволяет ввести строку `"Hello, World!"` в текстовом поле, оценить внешний вид печатаемой страницы, а затем напечатать введенную строку в виде баннера.

Листинг 7.12. Исходный код из файла `Book/BookTestFrame.java`

```
1  package book;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.awt.print.*;
6  import javax.print.attribute.*;
7  import javax.swing.*;
8
9  /**
10  * Этот фрейм содержит текстовое поле для ввода баннера, а также кнопки
11  * для печати, настройки, предварительного просмотра печатаемой страницы
12 */
13 public class BookTestFrame extends JFrame
14 {
15   private JTextField text;
16   private PageFormat pageFormat;
17   private PrintRequestAttributeSet attributes;
18
19   public BookTestFrame()
20   {
21     text = new JTextField();
22     add(text, BorderLayout.NORTH);
23
24     attributes = new HashPrintRequestAttributeSet();
25
26     JPanel buttonPanel = new JPanel();
27
28     JButton printButton = new JButton("Print");
29     buttonPanel.add(printButton);
30     printButton.addActionListener(new ActionListener()
31     {
32       public void actionPerformed(ActionEvent event)
```

```
33          {
34            try
35            {
36              PrinterJob job = PrinterJob.getPrinterJob();
37              job.setPageable(makeBook());
38              if (job.printDialog(attributes))
39              {
40                job.print(attributes);
41              }
42            }
43            catch (PrinterException e)
44            {
45              JOptionPane.showMessageDialog(BookTestFrame.this, e);
46            }
47          }
48        });
49
50        JButton pageSetupButton = new JButton("Page setup");
51        buttonPanel.add(pageSetupButton);
52        pageSetupButton.addActionListener(new ActionListener()
53          {
54            public void actionPerformed(ActionEvent event)
55            {
56              PrinterJob job = PrinterJob.getPrinterJob();
57              pageFormat = job.pageDialog(attributes);
58            }
59          });
60
61        JButton printPreviewButton = new JButton("Print preview");
62        buttonPanel.add(printPreviewButton);
63        printPreviewButton.addActionListener(new ActionListener()
64          {
65            public void actionPerformed(ActionEvent event)
66            {
67              PrintPreviewDialog dialog = new PrintPreviewDialog(makeBook());
68              dialog.setVisible(true);
69            }
70          });
71
72        add(buttonPanel, BorderLayout.SOUTH);
73        pack();
74      }
75
76      /**
77       * Составляет книгу из страницы обложки и страниц баннера
78      */
79      public Book makeBook()
80      {
81        if (pageFormat == null)
82        {
83          PrinterJob job = PrinterJob.getPrinterJob();
84          pageFormat = job.defaultPage();
85        }
86        Book book = new Book();
87        String message = text.getText();
88        Banner banner = new Banner(message);
89        int pageCount =
90           banner.getPageCount((Graphics2D) getGraphics(), pageFormat);
```

```
91      book.append(new CoverPage(
92         message + " (" + pageCount + " pages)"), pageFormat);
93      book.append(banner, pageFormat, pageCount);
94      return book;
95    }
96 }
```

Листинг 7.13. Исходный код из файла book/Banner.java

```
1   package book;
2
3   import java.awt.*;
4   import java.awt.font.*;
5   import java.awt.geom.*;
6   import java.awt.print.*;
7
8   /**
9    * Баннер для печати текстовой строки на нескольких страницах
10  */
11  public class Banner implements Printable
12  {
13    private String message;
14    private double scale;
15
16    /**
17     * Конструирует баннер
18     * @param m Строка сообщения
19    */
20    public Banner(String m)
21    {
22      message = m;
23    }
24
25    /**
26     * Получает количество страниц в данном разделе
27     * @param g2 Графический контекст
28     * @param pf Формат страницы
29     * @return Возвращает количество требующихся страниц
30    */
31    public int getPageCount(Graphics2D g2, PageFormat pf)
32    {
33      if (message.equals("")) return 0;
34      FontRenderContext context = g2.getFontRenderContext();
35      Font f = new Font("Serif", Font.PLAIN, 72);
36      Rectangle2D bounds = f.getStringBounds(message, context);
37      scale = pf.getImageableHeight() / bounds.getHeight();
38      double width = scale * bounds.getWidth();
39      int pages = (int) Math.ceil(width / pf.getImageableWidth());
40      return pages;
41    }
42
43    public int print(Graphics g, PageFormat pf, int page)
44       throws PrinterException
45    {
46      Graphics2D g2 = (Graphics2D) g;
47      if (page > getPageCount(g2, pf)) return Printable.NO_SUCH_PAGE;
48      g2.translate(pf.getImageableX(), pf.getImageableY());
```

```
49
50        drawPage(g2, pf, page);
51        return Printable.PAGE_EXISTS;
52      }
53
54      public void drawPage(Graphics2D g2, PageFormat pf, int page)
55      {
56        if (message.equals("")) return;
57        page--; // учитывать страницу обложки
58
59        drawCropMarks(g2, pf);
60        g2.clip(new Rectangle2D.Double(0, 0, pf.getImageableWidth(),
61           pf.getImageableHeight()));
62        g2.translate(-page * pf.getImageableWidth(), 0);
63        g2.scale(scale, scale);
64        FontRenderContext context = g2.getFontRenderContext();
65        Font f = new Font("Serif", Font.PLAIN, 72);
66        TextLayout layout = new TextLayout(message, f, context);
67        AffineTransform transform =
68           AffineTransform.getTranslateInstance(0, layout.getAscent());
69        Shape outline = layout.getOutline(transform);
70        g2.draw(outline);
71      }
72
73      /**
74       * Рисует полудюймовые метки обрезки в углах страницы
75       * @param g2 Графический контекст
76       * @param pf Формат страницы
77      */
78      public void drawCropMarks(Graphics2D g2, PageFormat pf)
79      {
80        final double C = 36; // длина метки обрезки = 1/2 дюйма
81        double w = pf.getImageableWidth();
82        double h = pf.getImageableHeight();
83        g2.draw(new Line2D.Double(0, 0, 0, C));
84        g2.draw(new Line2D.Double(0, 0, C, 0));
85        g2.draw(new Line2D.Double(w, 0, w, C));
86        g2.draw(new Line2D.Double(w, 0, w - C, 0));
87        g2.draw(new Line2D.Double(0, h, 0, h - C));
88        g2.draw(new Line2D.Double(0, h, C, h));
89        g2.draw(new Line2D.Double(w, h, w, h - C));
90        g2.draw(new Line2D.Double(w, h, w - C, h));
91      }
92    }
93
94    /**
95     * Этот класс печатает страницу обложки с заглавием
96    */
97    class CoverPage implements Printable
98    {
99      private String title;
100
101     /**
102      * Конструирует страницу обложки
103      * @param t Заглавие
104     */
105     public CoverPage(String t)
106     {
```

```
107     title = t;
108   }
109
110   public int print(Graphics g, PageFormat pf, int page)
111      throws PrinterException
112   {
113     if (page >= 1) return Printable.NO_SUCH_PAGE;
114     Graphics2D g2 = (Graphics2D) g;
115     g2.setPaint(Color.black);
116     g2.translate(pf.getImageableX(), pf.getImageableY());
117     FontRenderContext context = g2.getFontRenderContext();
118     Font f = g2.getFont();
119     TextLayout layout = new TextLayout(title, f, context);
120     float ascent = layout.getAscent();
121     g2.drawString(title, 0, ascent);
122     return Printable.PAGE_EXISTS;
123   }
124 }
```

Листинг 7.14. Исходный код из файла book/PrintPreviewDialog.java

```
1  package book;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.awt.font.*;
6  import java.awt.print.*;
7  import javax.swing.*;
8
9  /**
10  * Этот класс реализует типичное окно предварительного просмотра печати
11 */
12 class PrintPreviewDialog extends JDialog
13 {
14   private static final int DEFAULT_WIDTH = 300;
15   private static final int DEFAULT_HEIGHT = 300;
16
17   private PrintPreviewCanvas canvas;
18   /**
19    * Конструирует диалоговое окно предварительного просмотра печати
20    * @param p Печатаемый объект типа Printable
21    * @param pf Формат страницы
22    * @param pages Количество страниц в печатаемом объекте p
23   */
24   public PrintPreviewDialog(Printable p, PageFormat pf, int pages)
25   {
26     Book book = new Book();
27     book.append(p, pf, pages);
28     layoutUI(book);
29   }
30
31   /**
32    * Создает диалоговое окно предварительного просмотра печати
33    * @param b Объект книги типа Book
34   */
35   public PrintPreviewDialog(Book b)
36   {
```

```
37      layoutUI(b);
38    }
39
40    /**
41     * Компонует ГПИ в диалоговом окне
42     * @param book Предварительно просматриваемая книга
43    */
44    public void layoutUI(Book book)
45    {
46      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
47
48      canvas = new PrintPreviewCanvas(book);
49      add(canvas, BorderLayout.CENTER);
50
51      JPanel buttonPanel = new JPanel();
52
53      JButton nextButton = new JButton("Next");
54      buttonPanel.add(nextButton);
55      nextButton.addActionListener(new ActionListener()
56        {
57          public void actionPerformed(ActionEvent event)
58          {
59            canvas.flipPage(1);
60          }
61        });
62
63      JButton previousButton = new JButton("Previous");
64      buttonPanel.add(previousButton);
65      previousButton.addActionListener(new ActionListener()
66        {
67          public void actionPerformed(ActionEvent event)
68          {
69            canvas.flipPage(-1);
70          }
71        });
72
73      JButton closeButton = new JButton("Close");
74      buttonPanel.add(closeButton);
75      closeButton.addActionListener(new ActionListener()
76        {
77          public void actionPerformed(ActionEvent event)
78          {
79            setVisible(false);
80          }
81        });
82
83      add(buttonPanel, BorderLayout.SOUTH);
84    }
85  }
```

Листинг 7.15. Исходный код из файла book/PrintPreviewCanvas.java

```
1  package book;
2
3  import java.awt.*;
4  import java.awt.geom.*;
5  import java.awt.print.*;
```

```
6  import javax.swing.*;
7
8  /**
9   * Полотно для отображения предварительного просмотра печати
10 */
11 class PrintPreviewCanvas extends JComponent
12 {
13   private Book book;
14   private int currentPage;
15
16   /**
17    * Конструирует полотно для предварительного просмотра печати
18    * @param b Предварительно просматриваемая книга
19   */
20   public PrintPreviewCanvas(Book b)
21   {
22     book = b;
23     currentPage = 0;
24   }
25   public void paintComponent(Graphics g)
26   {
27     Graphics2D g2 = (Graphics2D) g;
28     PageFormat pageFormat = book.getPageFormat(currentPage);
29
30     double xoff; // координата x смещения начала страницы в окне
31     double yoff; // координата y смещения начала страницы в окне
32     double scale; // масштабный коэффициент для подгонки
33                   // просматриваемой страницы по размерам окна
34     double px = pageFormat.getWidth();
35     double py = pageFormat.getHeight();
36     double sx = getWidth() - 1;
37     double sy = getHeight() - 1;
38     if (px / py < sx / sy) // отцентровать по горизонтали
39     {
40       scale = sy / py;
41       xoff = 0.5 * (sx - scale * px);
42       yoff = 0;
43     }
44     else
45     // отцентровать по вертикали
46     {
47       scale = sx / px;
48       xoff = 0;
49       yoff = 0.5 * (sy - scale * py);
50     }
51     g2.translate((float) xoff, (float) yoff);
52     g2.scale((float) scale, (float) scale);
53
54     // нарисовать контуры страницы, игнорируя поля
55     Rectangle2D page = new Rectangle2D.Double(0, 0, px, py);
56     g2.setPaint(Color.white);
57     g2.fill(page);
58     g2.setPaint(Color.black);
59     g2.draw(page);
60
61     Printable printable = book.getPrintable(currentPage);
62     try
63     {
64       printable.print(g2, pageFormat, currentPage);
```

```
65      }
66      catch (PrinterException e)
67      {
68        g2.draw(new Line2D.Double(0, 0, px, py));
69        g2.draw(new Line2D.Double(px, 0, 0, py));
70      }
71    }
72    /**
73     * Листать книгу на заданное число страниц
74     * @param by Число листаемых страниц. Отрицательные значения
75     *           данного параметра обозначают листание книги назад
76    */
77    public void flipPage(int by)
78    {
79      int newPage = currentPage + by;
80      if (0 <= newPage && newPage < book.getNumberOfPages())
81      {
82        currentPage = newPage;
83        repaint();
84      }
85    }
86  }
```

java.awt.print.PrinterJob 1.2

- `void setPageable(Pageable p)`

  Устанавливает печатаемый объект типа `Pageable`, например, объект книги типа `Book`.

java.awt.print.Book 1.2

- `void append(Printable p, PageFormat format)`
- `void append(Printable p, PageFormat format, int pageCount)`

  Добавляют новый раздел в данную книгу. Если количество страниц не указано, то добавляется первая страница.

- `Printable getPrintable(int page)`

  Получает печатаемый объект типа `Printable` для указанной страницы.

Службы печати

В предыдущих разделах было показано, как выводить двухмерную графику на печать. Но прикладной интерфейс печати API, внедренный в версии Java SE 1.4, обладает более широкими возможностями. В этом прикладном интерфейсе определен целый ряд типов данных и поддерживается поиск служб печати для их вывода. Ниже перечислены некоторые из поддерживаемых типов данных.

- Изображения в формате GIF, JPEG или PNG.
- Текстовые документы в формате HTML, PostScript или PDF.
- Неформатированные данные для печатающего устройства.

- Экземпляры класса, реализующего интерфейсы `Printable`, `Pageable` или `RenderableImage`.

Сами данные могут храниться в любом источнике байтов или символов, например, в потоке ввода, веб-ресурсе, доступном по указанному URL, или массиве. Сочетание типа данных и источника данных называется *разновидностью документа*. Так, в классе `DocFlavor` определен целый ряд внутренних классов для различных источников данных. Каждый из этих внутренних классов содержит константы для указания разновидности документа. Например, приведенная ниже константа описывает изображение формата GIF, считываемое из потока ввода. Все разновидности документов, т.е. допустимые сочетания типов источников и данных, включая и тип MIME, перечислены в табл. 7.3.

```
DocFlavor.INPUT_STREAM.GI
```

Таблица 7.3. Разновидности документов для служб печати

Источник данных	Тип данных	Тип MIME
`INPUT_STREAM`	`GIF`	`image/gif`
`URL`	`JPEG`	`image/jpeg`
`BYTE_ARRAY`	`PNG`	`image/png`
	`POSTSCRIPT`	`application/postscript`
	`PDF`	`application/pdf`
	`TEXT_HTML_HOST`	`text/html` (в кодировке, характерной для конкретного узла)
	`TEXT_HTML_US_ASCII`	`text/html; charset=us-ascii`
	`TEXT_HTML_UTF_8`	`text/html; charset=utf-8`
	`TEXT_HTML_UTF_16`	`text/html; charset=utf-16`
	`TEXT_HTML_UTF_16LE`	`text/html; charset=utf-16le` (прямой порядок следования байтов)
	`TEXT_HTML_UTF_16BE`	`text/html; charset=utf-16be` (обратный порядок следования байтов)
	`TEXT_PLAIN_HOST`	`text/plain` (в кодировке, характерной для конкретного узла)
	`TEXT_PLAIN_US_ASCII`	`text/plain; charset=us-ascii`
	`TEXT_PLAIN_UTF_8`	`text/plain; charset=utf-8`
	`TEXT_PLAIN_UTF_16`	`text/plain; charset=utf-16`
	`TEXT_PLAIN_UTF_16LE`	`text/plain; charset=utf-16le` (прямой порядок следования байтов)
	`TEXT_PLAIN_UTF_16BE`	`text/plain; charset=utf-16be` (обратный порядок следования байтов)
	`PCL`	`application/vnd.hp-PCL` (Hewlett Packard Printer Control Language)
	`AUTOSENSE`	`application/octet-stream` (неформатированные данные для печатающего устройства)
`READER`	`TEXT_HTML`	`text/html; charset=utf-16`
`STRING`	`TEXT_PLAIN`	`text/plain; charset=utf-16`

Окончание табл. 7.3.

Источник данных	Тип данных	Тип MIME
CHAR_ARRAY		
SERVICE_FORMATTED	**PRINTABLE**	Отсутствует
	PAGEABLE	Отсутствует
	RENDERABLE_IMAGE	Отсутствует

Допустим, требуется напечатать изображение из файла формата GIF. Для этого нужно сначала выяснить, имеется ли соответствующая *служба печати*, способная справиться с подобной задачей. Статический метод `lookupPrintServices()` из класса `PrintServiceLookup` возвращает массив объектов типа `PrintService`, способных напечатать указанную разновидность документа, как показано ниже.

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
PrintService[] services =
    PrintServiceLookup.lookupPrintServices(flavor, null);
```

Второй параметр метода `lookupPrintServices()` принимает пустое значение `null`, которое означает, что поиск подходящей службы печати не ограничивается какими-то определенными атрибутами печати. Более подробно атрибуты печати рассматриваются в следующем разделе.

Если в результате поиска служб печати получен массив **`PrintService[]`** с несколькими элементами, то среди перечисленных в нем служб печати следует выбирать какую-нибудь одну. Для того чтобы извлечь список имен служб печати (например, имен печатающих устройств) и предоставить пользователю возможность выбора, достаточно вызвать метод **`getName()`** из класса **`PrintService`**.

Далее из выбранной службы нужно получить задание на печать следующим образом:

```
DocPrintJob job = services[i].createPrintJob();
```

Для печати документа определенной разновидности нужно создать объект, класс которого реализует интерфейс **`Doc`**. Для этой цели в библиотеке Java предусмотрен класс `SimpleDoc`. Конструктору класса `SimpleDoc` следует передать объект, представляющий источник данных, разновидность документа и дополнительный, но необязательный набор атрибутов печати, как показано в приведенном ниже примере кода.

```
InputStream in = new FileInputStream(fileName);
Doc doc = new SimpleDoc(in, flavor, null);
```

И наконец, когда все готово для печати, можно приступить к выводу документа на печать, вызвав приведенный ниже метод. Как и прежде, пустое значение `null` второго параметра этого метода можно заменить набором атрибутов печати.

```
job.print(doc, null);
```

Следует, однако, иметь в виду, что описанный выше процесс печати совершенно не похож на рассматривавшийся в предыдущем разделе, потому что в данном случае отсутствует всякое взаимодействие с пользователем посредством диалоговых окон. Применяя подобный механизм, можно организовать печать на сервере, где пользователи должны передавать задания на печать из специальной веб-формы. В примере программы, исходный код которой приведен в листинге 7.16, демонстрируется применение службы печати для распечатки файла изображения.

Листинг 7.16. Исходный код из файла `printService/PrintServiceTest.java`

```
1  package printService;
2
3  import java.io.*;
4  import java.nio.file.*;
5  import javax.print.*;
6
7  /**
8   * В этой программе демонстрируется применение служб печати. Она
9   * позволяет напечатать изображение формата GIF в любой службе
10  * печати, поддерживающей разновидность документов формата GIF
11  * @version 1.10 2007-08-16
12  * @author Cay Horstmann
13 */
14 public class PrintServiceTest
15 {
16   public static void main(String[] args)
17   {
18     DocFlavor flavor = DocFlavor.URL.GIF;
19     PrintService[] services =
20        PrintServiceLookup.lookupPrintServices(flavor, null);
21     if (args.length == 0)
22     {
23       if (services.length == 0) System.out.println(
24          "No printer for flavor " + flavor);
25       else
26       {
27         System.out.println("Specify a file of flavor " + flavor
28            + "\nand optionally the number of the desired printer.");
29         for (int i = 0; i < services.length; i++)
30            System.out.println((i + 1) + ": " + services[i].getName());
31       }
32       System.exit(0);
33     }
34     String fileName = args[0];
35     int p = 1;
36     if (args.length > 1) p = Integer.parseInt(args[1]);
37     if (fileName == null) return;
38     try (InputStream in = Files.newInputStream(Paths.get(fileName)))
39     {
40       Doc doc = new SimpleDoc(in, flavor, null);
41       DocPrintJob job = services[p - 1].createPrintJob();
42       job.print(doc, null);
43     }
44     catch (Exception ex)
45     {
46       ex.printStackTrace();
47     }
48   }
49 }
```

javax.print.PrintServiceLookup 1.4

- `PrintService[] lookupPrintServices(DocFlavor` *flavor*`, AttributeSet` *attributes*`)`

  Ищет службы печати, способные справиться с указанной разновидностью документа и атрибутами печати.

Параметры:	*flavor*	Разновидность документа
	attributes	Требуемые атрибуты печати или пустое значение `null`, если эти атрибуты не принимаются во внимание

javax.print.PrintService 1.4

- `DocPrintJob createPrintJob()`

  Составляет задание на печать для вывода экземпляра класса, реализующего интерфейс `Doc` (например, `SimpleDoc`).

javax.print.DocPrintJob 1.4

- `void print(Doc` *doc*`, PrintRequestAttributeSet` *attributes*`)`

  Выводит на печать заданный документ с указанными атрибутами печати.

Параметры:	*doc*	Печатаемый документ
	attributes	Требуемые атрибуты печати или пустое значение `null`, если эти атрибуты не принимаются во внимание

javax.print.SimpleDoc 1.4

- `SimpleDoc(Object` *data*`, DocFlavor` *flavor*`, DocAttributeSet` *attributes*`)`

  Создает объект типа `SimpleDoc`, который может быть выведен на печать в задании типа `DocPrintJob`.

Параметры:	*data*	Объект с данными для печати, например, поток ввода или объект типа `Printable`
	flavor	Разновидность документа с данными для печати
	attributes	Атрибуты печати документа или пустое значение `null`, если атрибуты печати не требуются

Потоковые службы печати

Обычные службы печати направляют данные на печатающее устройство. А потоковые службы печати формируют данные аналогичным образом, но направляют их в поток вывода. Такая потребность возникает в тех случаях, когда необходимо задержать печать или интерпретировать формат печати данных в других программах. Если, например, данные печатаются в формате PostScript, их целесообразно сохранить в файле, поскольку многие программы способны обращаться с файлами формата PostScript. На платформе Java предусмотрена потоковая служба печати, способная распечатывать изображения и двухмерную графику в формате PostScript. Эти службы можно использовать во всех системах — даже в тех, где нет локальных печатающих устройств.

Перечисление потоковых служб печати организовано немного сложнее, чем локальных служб печати. Для этого нужно получить сначала разновидность типа `DocFlavor` печатаемого объекта и тип MIME потока вывода, а затем массив фабричных объектов, представляющих потоковые службы печати, как показано ниже.

```
DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
String mimeType = "application/postscript";
StreamPrintServiceFactory[] factories =
    StreamPrintServiceFactory.lookupStreamPrintServiceFactories(
    flavor, mimeType);
```

В классе `StreamPrintServiceFactory` нет ни одного метода, с помощью которого можно было бы отличить один фабричный объект потоковой службы от другого, поэтому выбирается самый первый объект, находящийся в элементе массива `factories[0]`. Для получения объекта потоковой службы печати типа `StreamPrintService` следует вызвать приведенный ниже метод `getPrintService()`, указав поток вывода в качестве его параметра. Класс `StreamPrintService` является производным от класса `PrintService`, поэтому для вывода на печать остается лишь выполнить действия, описанные в предыдущем разделе.

```
OutputStream out = new FileOutputStream(fileName);
StreamPrintService service = factories[0].getPrintService(out);
```

javax.print.StreamPrintServiceFactory 1.4

- `StreamPrintServiceFactory[] lookupStreamPrintServiceFactories(DocFlavor` *`flavor`*`, String` *`mimeType`*`)`

  Ищет потоковые службы печати, способные напечатать документ указанной разновидности и создать поток вывода заданного типа MIME.

- `StreamPrintService getPrintService(OutputStream` *`out`*`)`

  Получает службу печати, направляющую данные в указанный поток вывода.

Атрибуты печати

Прикладной интерфейс API службы печати содержит сложный набор интерфейсов и классов для указания различных типов атрибутов печати, которые делятся на четыре основные группы. Первые две группы атрибутов печати определяют запросы, направляемые на печатающее устройство.

- *Атрибуты запроса печати.* Запрашивают отдельные характеристики всех документов в задании на печать. В качестве примеров таких характеристик можно привести двухстороннюю печать или формат бумаги.
- *Атрибуты документа.* Используются только для какого-нибудь одного объекта-документа.

Оставшиеся две группы атрибутов содержат сведения о печатающем устройстве и состоянии задания на печать.

- *Атрибуты службы печати.* Содержат сведения о службе печати (например, о производителе и модели печатающего устройства, а также о том, принимает ли печатающее устройство задание на печать в настоящий момент).
- *Атрибуты задания на печать* содержат сведения о состоянии определенного задания на печать (например, завершено ли задание или еще не выполнено).

Для описания различных атрибутов печати предусмотрен интерфейс `Attribute` и перечисленные ниже подчиненные интерфейсы.

```
PrintRequestAttribute
DocAttribute
PrintServiceAttribute
PrintJobAttribute
SupportedValuesAttribute
```

Отдельные классы атрибутов реализуют один или несколько этих интерфейсов. Например, экземпляры класса `Copies` используются для описания количества копий печатаемого документа. Этот класс реализует интерфейсы `PrintRequestAttribute` и `PrintJobAttribute`. Задание на печать может содержать запрос на печать нескольких копий, тогда как атрибутом задания на печать может быть количество фактически напечатанных копий. Фактическое количество копий может оказаться меньше требуемого из-за ограничений, накладываемых печатающим устройством, или исчерпания бумаги.

Интерфейс `SupportedValuesAttribute` указывает, что значения атрибута не отражают фактический запрос или текущее состояние, а демонстрируют функциональные возможности службы печати. Например, объект класса `CopiesSupported` реализует интерфейс `SupportedValuesAttribute` и сообщает, что печатающее устройство поддерживает печать от 1 до 99 копий. На рис. 7.38 схематически показана иерархия наследования классов, реализующих атрибуты.

Помимо интерфейсов и классов для отдельных атрибутов, в прикладном интерфейсе API служб печати предусмотрены интерфейсы и классы для наборов атрибутов. Так, у интерфейса `AttribureSet` имеются следующие подчиненные интерфейсы:

```
PrintRequestAttributeSet
DocAttributeSet
PrintServiceAttributeSet
PrintJobAttributeSet
```

Для каждого из них предусмотрен класс, реализующий свой интерфейс. Всего насчитывается пять таких классов, как показано ниже. На рис. 7.39 схематически показана иерархия наследования классов, реализующих наборы атрибутов.

```
HashAttributeSet
HashPrintRequestAttributeSet
HashDocAttributeSet
HashPrintServiceAttributeSet
HashPrintJobAttributeSet
```

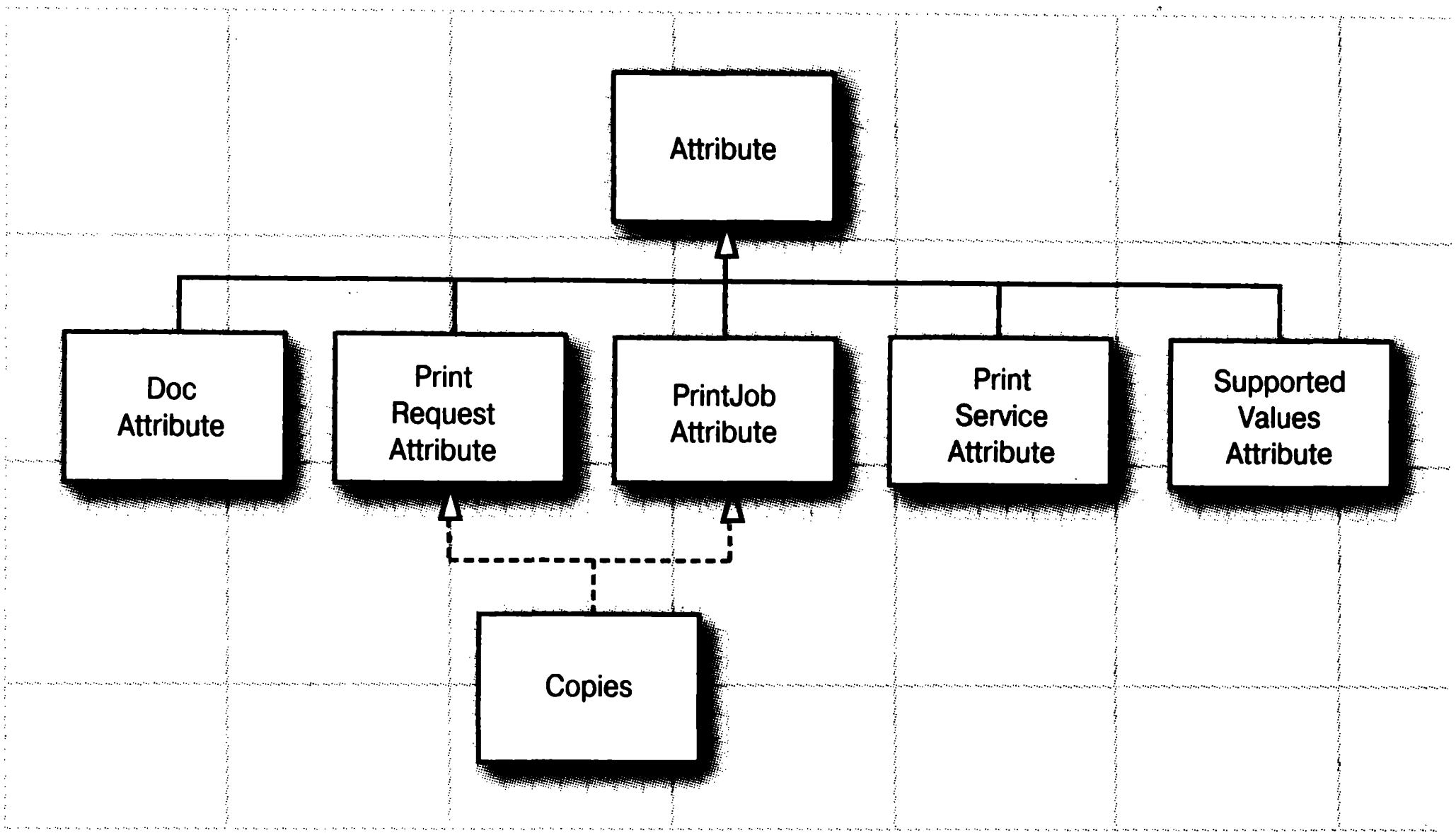


Рис. 7.38. Иерархия наследования классов, реализующих атрибуты

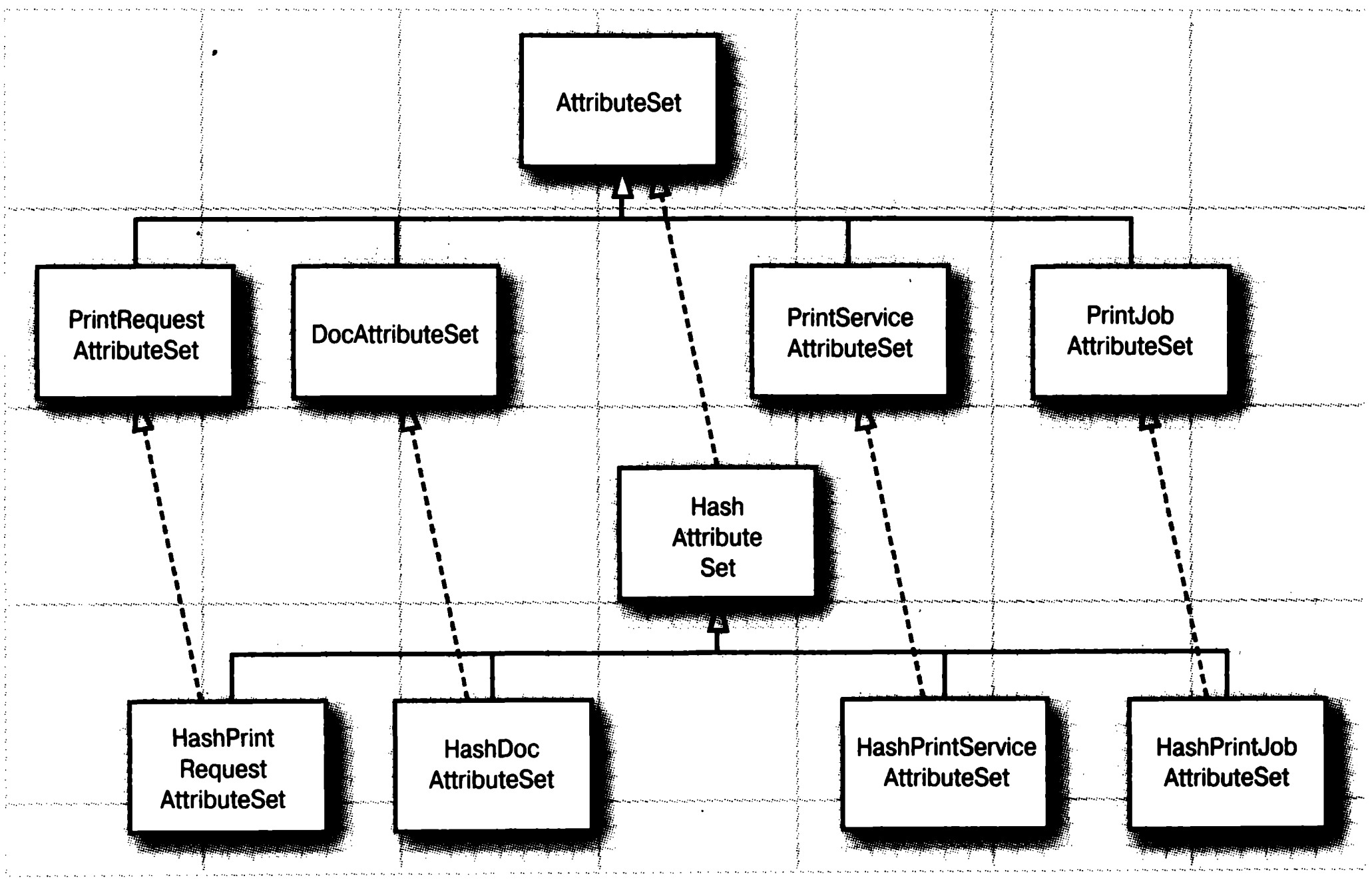


Рис. 7.39. Иерархия наследования классов, реализующих наборы атрибутов

Например, создать набор атрибутов для запроса на печать можно следующим образом:

```
PrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
```

После создания набора атрибутов потребность в префиксе Hash отпадает. А зачем вообще нужны все эти интерфейсы? Они дают возможность проверить правильность использования атрибутов. Например, интерфейс DocAttributeSet принимает только те объекты, классы которых реализуют интерфейс DocAttribute, а любая попытка добавить какой-нибудь другой атрибут приведет к ошибке.

Набор атрибутов представляет собой особый вид отображения, где ключи являются объектами класса Class, а значения — объектами класса, реализующего интерфейс Attribute. Так, если создать и вставить объект в набор атрибутов с помощью оператора new Copies(10), то его ключом будет объект Copies.class типа Class. Этот ключ называется *категорией* атрибутов. В интерфейсе Attribute объявляется приведенный ниже метод, возвращающий категорию атрибута. Класс Copies содержит определение этого метода для возврата объекта Copies.class, но совсем не обязательно, чтобы категория была объектом того же класса, что и класс атрибута.

```
Class getCategory()
```

При вводе атрибута в набор атрибутов категория извлекается автоматически, поэтому достаточно ввести только значение атрибута, как показано ниже. Если же впоследствии ввести другой атрибут из той же самой категории, то он будет записан вместо первого.

```
attributes.add(new Copies(10));
```

Категория используется в качестве ключа для извлечения атрибута, как показано ниже.

```
AttributeSet attributes = job.getAttributes();
Copies copies = (Copies) attribute.get(Copies.class);
```

Атрибуты группируются по типам значений. Например, атрибут типа Copies может иметь любое целочисленное значение, поскольку класс Copies является расширением класса IntegerSyntax, предназначенного для обработки всех целочисленных атрибутов. Метод getValue() возвращает целочисленное значение атрибута:

```
int n = copies.getValue();
```

Перечисленные ниже классы инкапсулируют соответственно символьную строку, дату/время или URI.

```
TextSyntax
DateTimeSyntax
URISyntax
```

И наконец, многие атрибуты могут принимать ограниченное количество значений. Например, у атрибута типа PrintQuality имеются всего три допустимых значения для указания экономного, нормального и высокого качества печати, представленных следующими константами:

```
PrintQuality.DRAFT
PrintQuality.NORMAL
PrintQuality.HIGH
```

Классы атрибутов с ограниченным количеством значений расширяют класс EnumSyntax, предоставляющий служебные методы для установки перечислимых значений с учетом типовой безопасности. Наличие таких классов избавляет от необходимости беспокоиться о конкретном механизме реализации при использовании атрибутов. Достаточно ввести в набор атрибутов именованные значения следующим образом:

```
attributes.add(PrintQuality.HIGH);
```

Ниже показано, как проверить значение атрибута.

```
if (attributes.get(PrintQuality.class) == PrintQuality.HIGH)
. . .
```

Атрибуты печати перечислены в табл. 7.4. Во втором столбце приведен суперкласс каждого класса атрибута (например, класс `IntegerSyntax` для атрибута типа `Copies`) или набор перечислимых значений. А в последних четырех столбцах указаны интерфейсы, которые реализует данный класс атрибута: `DocAttribute` (DA), `PrintJobAttribute` (PJA), `PrintRequestAttribute` (PRA) и `PrintServiceAttribute` (PSA).

Таблица 7.4. Атрибуты печати

Атрибут	Суперкласс или набор констант	DA	PJA	PRA	PSA
Chromaticity	MONOCHROME, COLOR	✓	✓	✓	
ColorSupported	SUPPORTED, NOT_SUPPORTED				✓
Compression	COMPRESS, DEFLATE, GZIP, NONE	✓			
Copies	IntegerSyntax		✓	✓	
DateTimeAtCompleted	DateTimeSyntax		✓		
DateTimeAtCreation	DateTimeSyntax		✓		
DateTimeAtProcessing	DateTimeSyntax		✓		
Destination	URISyntax		✓	✓	
DocumentName	TextSyntax	✓			
Fidelity	FIDELITY_TRUE, FIDELITY_FALSE		✓	✓	
Finishings	NONE, STAPLE, EDGE_STITCH, BIND, SADDLE_STITCH, COVER, . . .	✓	✓	✓	
JobHoldUntil	DateTimeSyntax		✓	✓	
JobImpressions	IntegerSyntax		✓	✓	
JobImpressionsCompleted	IntegerSyntax		✓		
JobKOctets	IntegerSyntax		✓	✓	
JobKOctetsProcessed	IntegerSyntax		✓		
JobMediaSheets	IntegerSyntax		✓	✓	
JobMediaSheetsCompleted	IntegerSyntax		✓		
JobMessageFromOperator	TextSyntax		✓		
JobName	TextSyntax		✓	✓	
JobOriginatingUserName	TextSyntax		✓		
JobPriority	IntegerSyntax		✓	✓	
JobSheets	STANDARD, NONE		✓	✓	
JobState	ABORTED, CANCELED, COMPLETED, PENDING, PENDING_HELD, PROCESSING, PROCESSING_STOPPED		✓		

Продолжение табл. 7.4

Атрибут	Суперкласс или набор констант	DA	PJA	PRA	PSA
JobStateReason	ABORTED_BY_SYSTEM, DOCUMENT_FORMAT_ERROR и проч.				
JobStateReasons	HashSet				
MediaName	ISO_A4_WHITE, ISO_A4_TRANSPARENT, NA_LETTER_WHITE, NA_LETTER_TRANSPARENT	✓	✓	✓	
MediaSize	ISO.A0-ISO.A10, ISO.B0-ISO.B10, ISO.C0-ISO.C10, NA.LETTER, NA.LEGAL и другие форматы страниц и бумаги				
MediaSizeName	ISO.A0-ISO.A10, ISO.B0-ISO.B10, ISO.C0-ISO.C10, NA.LETTER, NA.LEGAL и другие форматы страниц и бумаги	✓	✓	✓	
MediaTray	TOP, MIDDLE, BOTTOM, SIDE, ENVELOPE, LARGE_CAPACITY, MAIN, MANUAL	✓	✓	✓	
MultipleDocumentHandling	SINGLE_DOCUMENT, SINGLE_DOCUMENT_NEW_SHEET, SEPARATE_DOCUMENTS_COLLATED_COPIES, SEPARATE_DOCUMENTS_UNCOLLATED_COPIES		✓	✓	
NumberOfDocuments	IntegerSyntax		✓		
NumberOfInterveningJobs	IntegerSyntax		✓		
NumberUp	IntegerSyntax	✓	✓	✓	
OrientationRequested	PORTRAIT, LANDSCAPE, REVERSE_PORTRAIT, REVERSE_LANDSCAPE	✓	✓	✓	
OutputDeviceAssigned	TextSyntax		✓		
PageRanges	SetOfInteger	✓	✓	✓	
PagesPerMinute	IntegerSyntax				✓
PagesPerMinuteColor	IntegerSyntax				✓
PDLOverrideSupported	ATTEMPTED, NOT_ATTEMPTED				✓
PresentationDirection	TORIGHT_TOBOTTOM, TORIGHT_TOTOP, TOBOTTOM_TORIGHT, TOBOTTOM_TOLEFT, TOLEFT_TOBOTTOM, TOLEFT_TOTOP, TOTOP_TORIGHT, TOTOP_TOLEFT	✓	✓		
PrinterInfo	TextSyntax				✓
PrinterIsAcceptingJobs	ACCEPTING_JOBS, NOT_ACCEPTING_JOBS				✓
PrinterLocation	TextSyntax				✓
PrinterMakeAndModel	TextSyntax				✓

Окончание табл. 7.4

Атрибут	Суперкласс или набор констант	DA	PJA	PRA	PSA
PrinterMessageFromOperator	TextSyntax				✓
PrinterMoreInfo	URISyntax				✓
PrinterMoreInfoManufacturer	URISyntax				✓
PrinterName	TextSyntax				✓
PrinterResolution	ResolutionSyntax	✓	✓	✓	
PrinterState	PROCESSING, IDLE, STOPPED, UNKNOWN				✓
PrinterStateReason	COVER_OPEN, FUSER_OVER_TEMP, MEDIA_JAM и проч.				
PrinterStateReasons	HashMap				
PrinterURI	URISyntax				✓
PrintQuality	DRAFT, NORMAL, HIGH	✓	✓	✓	
QueuedJobCount	IntegerSyntax				✓
ReferenceUriSchemesSupported	FILE, FTP, GOPHER, HTTP, HTTPS, NEWS, NNTP, WAIS				
RequestingUserName	TextSyntax			✓	
Severity	ERROR, REPORT, WARNING				
SheetCollate	COLLATED, UNCOLLATED	✓	✓	✓	
Sides	ONE_SIDED, DUPLEX (или TWO_SIDED_LONG_EDGE), TUMBLE (или TWO_SIDED_SHORT_EDGE)	✓	✓	✓	

НА ЗАМЕТКУ! Как видите, многие из перечисленных в табл. 7.4 атрибутов являются узкоспециализированными и появились вследствие применения протокола Internet Printing Protocol 1.1 (по стандарту RFC 2911).

НА ЗАМЕТКУ! В более ранней версии прикладного интерфейса API печати были внедрены классы `JobAttributes` и `PageAttributes`, которые служили той же цели, что и описываемые в этом разделе атрибуты печати. Но теперь эти классы уже считаются устаревшими.

javax.print.attribute.Attribute 1.4

- `Class getCategory()`

  Получает категорию данного атрибута.

- `String getName()`

  Получает имя данного атрибута.

javax.print.attribute.AttributeSet 1.4

- `boolean add(Attribute attr)`

  Вводит атрибут в набор атрибутов. Если в наборе имеется другой атрибут из той же категории, то новый атрибут замещает его. Возвращает логическое значение `true`, если в результате выполнения данной операции набор атрибутов изменился.

- `Attribute get(Class category)`

  Извлекает атрибут с ключом данной категории или пустое значение `null`, если такого атрибута не существует.

- `boolean remove(Attribute attr)`
- `boolean remove(Class category)`

  Удаляют указанный атрибут или атрибут указанной категории из набора атрибутов. Возвращают логическое значение `true`, если в результате выполнения данной операции набор атрибутов изменился.

- `Attribute[] toArray()`

  Возвращает массив со всеми атрибутами из данного набора.

`javax.print.PrintService 1.4`

- `PrintServiceAttributeSet getAttributes()`

  Получает атрибуты данной службы печати.

javax.print.DocPrintJob 1.4

- `PrintJobAttributeSet getAttributes()`

  Получает атрибуты данного задания на печать.

На этом обсуждение средств печати изображений завершается. Теперь вы знаете, как выводить двухмерную графику и другие типы документов на печать, как перечислять печатающие устройства и потоковые службы печати и как устанавливать и извлекать атрибуты. Перейдем далее к рассмотрению двух других средств, очень важных для создания пользовательских интерфейсов: буфера обмена и механизма перетаскивания объектов.

Буфер обмена

Одним из самых полезных и удобных в средах с ГПИ (вроде Windows и X Window System) является механизм *вырезания и вставки*. Он позволяет выбрать фрагмент данных в одной программе, вырезать или скопировать их в буфер обмена, а затем перейти в другую программу и вставить в ней содержимое буфера. Используя буфер обмена, можно переносить текст, рисунки или другие данные из одного документа

в другой или из одной части документа в другую. Механизм вырезания и вставки настолько естествен, что большинство пользователей применяют его совершенно интуитивно.

Несмотря на то что действия, требующиеся для работы с буфером обмена, очень просты, реализация буфера намного сложнее, чем это может показаться на первый взгляд. Допустим, какой-нибудь фрагмент текста скопирован из текстового редактора в буфер обмена. При вставке этого текста в другой текстовый редактор было бы желательно, чтобы шрифты и прочие атрибуты форматирования данного фрагмента остались прежними. Это означает, что в буфере обмена должны также храниться сведения о форматировании. Но при вставке фрагмента текста в поле обычно предполагается получить только символы вставляемого текста без дополнительных кодов форматирования. Для обеспечения такой гибкости данных поставщик должен предоставить их в различных форматах, а потребитель — выбрать среди них подходящий формат.

Системный буфер обмена в Windows и Mac OS реализован практически одинаково. Эти реализации отличаются лишь в деталях. А в механизме работы с буфером обмена в X Window System на вырезание и вставку элементов с более сложным форматом, чем простой текст, накладываются некоторые ограничения. Эти ограничения следует учитывать, прорабатывая примеры программ, рассматриваемые в данном разделе.

НА ЗАМЕТКУ! Для проверки типов объектов, которые могут передаваться между прикладными программами Java и системным буфером обмена, следует обратиться к файлу `jre/lib/flavormap.properties` на используемой платформе.

Нередко в прикладных программах требуется организовать копирование и вставку таких типов данных, которые не поддерживаются системным буфером обмена. В прикладном интерфейсе API для обмена данными поддерживаются ссылки на произвольные локальные объекты в рамках одной виртуальной машины. Между различными виртуальными машинами можно передавать сериализированные объекты и ссылки на удаленные объекты. В табл. 7.5 перечислены возможности механизма буфера обмена передавать данные.

Таблица 7.5. Возможности механизма буфера обмена передавать данные

Способ передачи данных	Формат данных
Между программой на Java и платформенно-ориентированной программой	Текст, рисунки, списки и т.п. (в зависимости от платформы)
Между двумя программами на Java	Сериализированные и удаленные объекты
В пределах одной программы на Java	Любые объекты

Классы и интерфейсы для передачи данных

Передача данных в Java реализована в пакете `java.awt.datatransfer`. Ниже перечислены некоторые особенности классов и интерфейсов из этого пакета.

- Объекты, передаваемые через буфер обмена, должны реализовывать интерфейс `Transferable`.

- Класс `Clipboard` описывает буфер обмена. Системный буфер обмена является конкретным примером реализации класса `Clipboard`.
- Класс `DataFlavor` описывает разновидности данных, которые могут быть размещены в буфере обмена.
- Класс `StringSelection` реализует интерфейс `Transferable` и используется для передачи текстовых строк.

  Для того чтобы получить уведомление о перезаписи содержимого буфера обмена, класс должен реализовать интерфейс `ClipboardOwner`. Право владеть буфером обмена позволяет выполнять "отложенное форматирование" сложных данных. Если программа передает простые данные (например, символьную строку), то она просто обновляет содержимое буфера обмена и переходит к выполнению следующих действий. Если же программа собирается вставить в буфер обмена сложные данные, которые могут быть по-разному отформатированы, то вряд ли стоит подготавливать все эти виды форматирования, так как многие из них могут вообще не понадобиться. Тем не менее данные нужно сохранить в буфере таким образом, чтобы впоследствии можно было обрабатывать типы документов по мере надобности. Поэтому владелец буфера обмена с помощью метода `lostOwnership()` должен быть своевременно уведомлен об изменении его содержимого. Изменение содержимого буфера обмена означает, что сведения о сложных данных больше не требуются. В приведенных далее примерах механизм, управляющий правами владения буфером обмена, не применяется.

Передача текста

Изучение классов передачи данных лучшего всего начать с простейшей ситуации: передачи текста через системный буфер обмена. Сначала нужно получить ссылку на системный буфер обмена:

```
Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
```

Строки, которые передаются в буфер обмена, необходимо заключить в оболочку объекта типа `StringSelection` с помощью следующего конструктора:

```
String text = . . .
StringSelection selection = new StringSelection(text);
```

Передача данных, по существу, происходит при вызове метода `setContents()`, который принимает объекты типа `StringSelection` и `ClipBoardOwner` в качестве параметров. Если владелец буфера обмена не имеет значения, то второму параметру данного метода следует присвоить пустое значение `null`:

```
clipboard.setContents(selection, null);
```

Ниже показано, каким образом выполняется обратная операция чтения строки из буфера обмена.

```
DataFlavor flavor = DataFlavor.stringFlavor;
if (clipboard.isDataFlavorAvailable(flavor)
    String text = (String) clipboard.getData(flavor);
```

В листинге 7.17 приведен исходный код примера программы, демонстрирующей операции копирования и вставки текста в приложении Java через системный буфер обмена. Если выбрать фрагмент текста в текстовой области и щелкнуть на кнопке

Copy в этой программе, выбранный текст копируется в системный буфер обмена, после чего его можно вставить в окно любого текстового редактора (рис. 7.40). Имеется также возможность выполнить обратную операцию, скопировав текст из окна любого текстового редактора и вставив его в демонстрационную программу, для чего достаточно щелкнуть в ней на кнопке Paste.

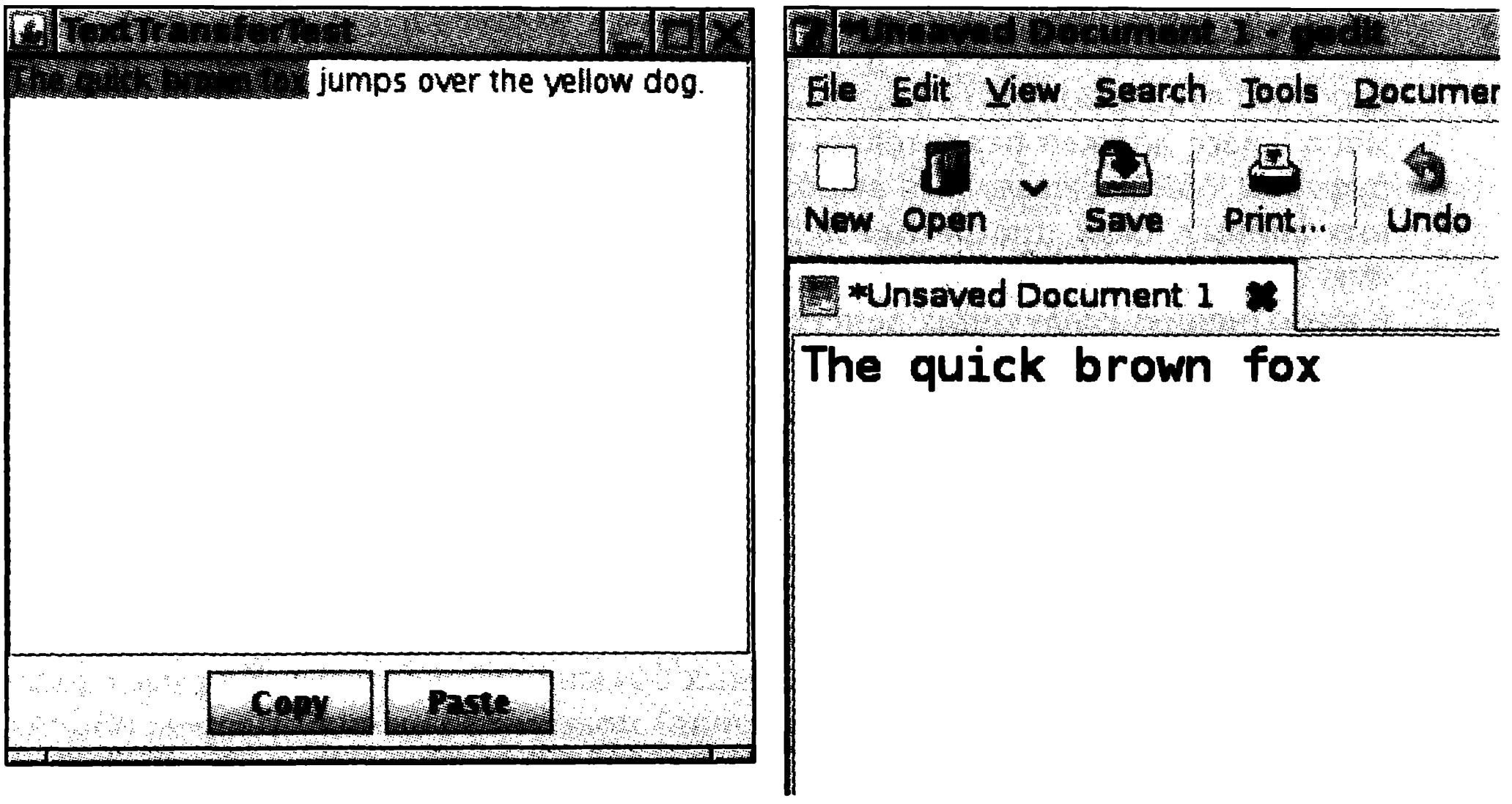


Рис. 7.40. Программа `TextTransferTest`

Листинг 7.17. Исходный код программы из файла `transferText/TextTransferFrame.java`

```
1  package transferText;
2
3  import java.awt.*;
4  import java.awt.datatransfer.*;
5  import java.awt.event.*;
6  import java.io.*;
7  import javax.swing.*;
8
9  /**
10  * Этот фрейм содержит текстовую область и кнопки для
11  * копирования и вставки текста
12 */
13 public class TextTransferFrame extends JFrame
14 {
15   private JTextArea textArea;
16   private static final int TEXT_ROWS = 20;
17   private static final int TEXT_COLUMNS = 60;
18
19   public TextTransferFrame()
20   {
21     textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
22     add(new JScrollPane(textArea), BorderLayout.CENTER);
23     JPanel panel = new JPanel();
24
```

```
25      JButton copyButton = new JButton("Copy");
26      panel.add(copyButton);
27      copyButton.addActionListener(new ActionListener()
28       {
29         public void actionPerformed(ActionEvent event)
30         {
31           copy();
32         }
33       });
34
35      JButton pasteButton = new JButton("Paste");
36      panel.add(pasteButton);
37      pasteButton.addActionListener(new ActionListener()
38        {
39          public void actionPerformed(ActionEvent event)
40          {
41            paste();
42          }
43        });
44
45      add(panel, BorderLayout.SOUTH);
46      pack();
47    }
48
49    /**
50     * Копирует выбранный текст в системный буфер обмена
51    */
52    private void copy()
53    {
54      Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
55      String text = textArea.getSelectedText();
56      if (text == null) text = textArea.getText();
57      StringSelection selection = new StringSelection(text);
58      clipboard.setContents(selection, null);
59    }
60    /**
61     * Вставляет текст из системного буфера обмена в текстовую область
62    */
63    private void paste()
64    {
65      Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
66      DataFlavor flavor = DataFlavor.stringFlavor;
67      if (clipboard.isDataFlavorAvailable(flavor))
68      {
69        try
70        {
71          String text = (String) clipboard.getData(flavor);
72          textArea.replaceSelection(text);
73        }
74        catch (UnsupportedFlavorException e)
75        {
76          JOptionPane.showMessageDialog(this, e);
77        }
78        catch (IOException e)
79        {
80          JOptionPane.showMessageDialog(this, e);
```

```
81          }
82       }
83    }
84 }
```

java.awt.Toolkit 1.0

- `Clipboard getSystemClipboard()` **1.1**

  Получает содержимое системного буфера обмена.

java.awt.datatransfer.Clipboard 1.1

- `Transferable getContents(Object` *requester*`)`

  Получает содержимое буфера обмена.

Параметры:	*requester*	Объект, который запрашивает содержимое буфера обмена. Значение этого параметра на самом деле не используется

- `void setContents(Transferable` *contents*`, ClipboardOwner` *owner*`)`

  Размещает содержимое в буфере обмена.

Параметры:	*contents*	Объект, инкапсулирующий содержимое буфера обмена
	owner	Объект, который должен быть уведомлен об изменении содержимого буфера обмена (методом `lostOwnership()`), а если уведомление не требуется, то указывается пустое значение **null** данного параметра

- `boolean isDataFlavorAvailable(DataFlavor` *flavor*`)` **5.0**

  Возвращает логическое значение `true`, если в буфере обмена содержатся данные указанной разновидности.

- `Object getData(DataFlavor` *flavor*`)` **5.0**

  Получает данные указанной разновидности, а если они недоступны, то генерирует исключение типа `UnsupportedFlavorException`.

java.awt.datatransfer.ClipboardOwner 1.1

- `void lostOwnership(Clipboard` *`clipboard`*`, Transferable` *`contents`*`)`

  Уведомляет объект, что он больше не является владельцем содержимого буфера обмена.

  Параметры:

`clipboard`	Буфер обмена, содержимое которого изменилось
`contents`	Элемент, размещаемый в буфере обмена данным владельцем

java.awt.datatransfer.Transferable 1.1

- `boolean isDataFlavorSupported(DataFlavor` *`flavor`*`)`

  Возвращает логическое значение `true`, если указанная разновидность данных поддерживается, а иначе — логическое значение `false`.

- `Object getTransferData(DataFlavor` *`flavor`*`)`

  Возвращает данные, отформатированные по указанной разновидности. Генерирует исключение типа `UnsupportedFlavorException`, если указанная разновидность данных не поддерживается.

Интерфейс `Transferable` и разновидности данных

Объект типа `DataFlavor`, представляющий разновидность данных, обладает следующими характеристиками.

- Тип MIME (например, `image/gif`).
- Класс представления, предназначенный для доступа к данным (например, `java.awt.Image`).

Кроме того, каждая разновидность данных имеет свое удобочитаемое название (например, `"GIF Image"`). Класс представления может быть указан с помощью параметра `class` в типе MIME следующим образом:

```
image/gif;class=java.awt.Image
```

НА ЗАМЕТКУ! Это всего лишь пример, который иллюстрирует синтаксис указания разновидности данных. Никакой стандартной разновидности для передачи графических данных изображения в формате GIF не существует.

Если параметр `class` не задан, то в качестве класса представления используется поток ввода типа `InputStream`. Для передачи локальных, сериализированных и удаленных объектов в Java предусмотрены следующие три типа MIME:

```
application/x-java-jvm-local-objectref
application/x-java-serialized-object
application/x-java-remote-object
```

НА ЗАМЕТКУ! Префикс **x-** означает экспериментальное имя, которое еще не утверждено IANA (Комитет по цифровым адресам в Интернете) в качестве стандартного типа MIME.

Например, стандартная разновидность данных `stringFlavor` описывается с помощью следующего типа MIME:

```
application/x-java-serialized-object;class=java.lang.String
```

У буфера обмена можно также запрашивать список всех доступных разновидностей данных следующим образом:

```
DataFlavor[] flavors = clipboard.getAvailableDataFlavors()
```

Кроме того, для буфера обмена можно устанавливать приемник событий типа `FlavorListener`, который будет уведомляться об изменении любой разновидности данных в буфере обмена. Подробнее об этом приемнике событий см. ниже в описании соответствующего прикладного интерфейса API.

java.awt.datatransfer.DataFlavor 1.1

- `DataFlavor(String` *mimeType*`, String` *humanPresentableName*`)`

  Создает разновидность данных, описывающую потоковые данные в формате, соответствующем указанному типу MIME.

  Параметры: *mimeType* Символьная строка, представляющая тип MIME

  humanPresentableName Удобочитаемое имя разновидности данных

- `DataFlavor(Class` *class*`, String` *humanPresentableName*`)`

  Создает разновидность данных, описывающую класс платформы Java со следующим типом MIME: `applications/x-java-serialized-object;class=`*имяКласса*.

  Параметры: *class* Класс, извлекаемый с помощью объекта типа `Transferable`

  humanPresentableName Удобочитаемое имя разновидности данных

- `String getMimeType()`

  Возвращает строку, представляющую тип MIME.

- `boolean isMimeTypeEqual(String` *mimeType*`)`

  Проверяет соответствие разновидности данных указанному типу MIME.

- `String getHumanPresentableName()`

  Возвращает удобочитаемое имя формата для разновидности данных

- `Class getRepresentationClass()`

  Возвращает объект типа `Class`, представляющий класс объекта, который возвращается объектом типа `Transferable` для разновидности данных. Это может быть параметр `class` типа MIME или поток ввода типа `InputStream`.

java.awt.datatransfer.Clipboard 1.1

- `DataFlavor[] getAvailableDataFlavors()` **5.0**

  Возвращает массив доступных разновидностей данных.

- `void addFlavorListener(FlavorListener` *`listener`*`)` **5.0**

  Добавляет приемник событий, который уведомляется об изменениях в наборе доступных разновидностей данных.

java.awt.datatransfer.Transferable 1.1

- `DataFlavor[] getTransferDataFlavors()`

  Возвращает массив поддерживаемых разновидностей данных.

java.awt.datatransfer.FlavorListener 5.0

- `void flavorsChanged(FlavorEvent` *`event`*`)`

  Вызывается при изменениях в наборе доступных разновидностей данных.

Передача изображений через буфер обмена

Объекты, которые требуется передать через буфер обмена, должны реализовывать интерфейс `Transferable`. В настоящее время этот интерфейс реализует в стандартной библиотеке Java единственный открытый класс `StringSelection`. В этом разделе показано, каким образом изображения передаются через буфер обмена. В стандартной библиотеке Java не предусмотрен класс для такой передачи, поэтому его придется реализовать самостоятельно.

Создать такой класс совсем не трудно. Он просто сообщает, что единственной доступной разновидностью данных является `DataFlavor.imageFlavor` и что она содержит объект `image` для передачи изображения:

```
class ImageTransferable implements Transferable
{
  private Image theImage;

  public ImageTransferable(Image image)
  {
    theImage = image;
  }

  public DataFlavor[] getTransferDataFlavors()
  {
    return new DataFlavor[] { DataFlavor.imageFlavor };
  }

  public boolean isDataFlavorSupported(DataFlavor flavor)
  {
    return flavor.equals(DataFlavor.imageFlavor);
  }
```

```
  public Object getTransferData(DataFlavor flavor)
    throws UnsupportedFlavorException
  {
    if(flavor.equals(DataFlavor.imageFlavor))
    {
      return theImage;
    }
    else
    {
      throw new UnsupportedFlavorException(flavor);
    }
  }
}
```


НА ЗАМЕТКУ! В Java SE предоставляется константа `DataFlavor.imageFlavor` и автоматически выполняются все необходимые действия по преобразованию изображений, сформированных средствами Java, в изображения собственного формата буфера обмена. Но, как ни странно, в JDK не предусмотрен класс-оболочка, чтобы разместить изображение в буфере обмена.

В примере программы, исходный код которой приведен в листинге 7.18, демонстрируется передача изображений из приложения Java в другое приложение и обратно через системный буфер обмена. После запуска на выполнение эта программа формирует изображение, состоящее из красного круга. Щелкните на кнопке Copy, чтобы скопировать это изображение в буфер обмена, а затем вставьте его в другое приложение (рис. 7.41). Далее скопируйте изображение из другого приложения в системный буфер обмена и щелкните на кнопке Paste, чтобы вставить его обратно в исходную программу, как показано на рис. 7.42.

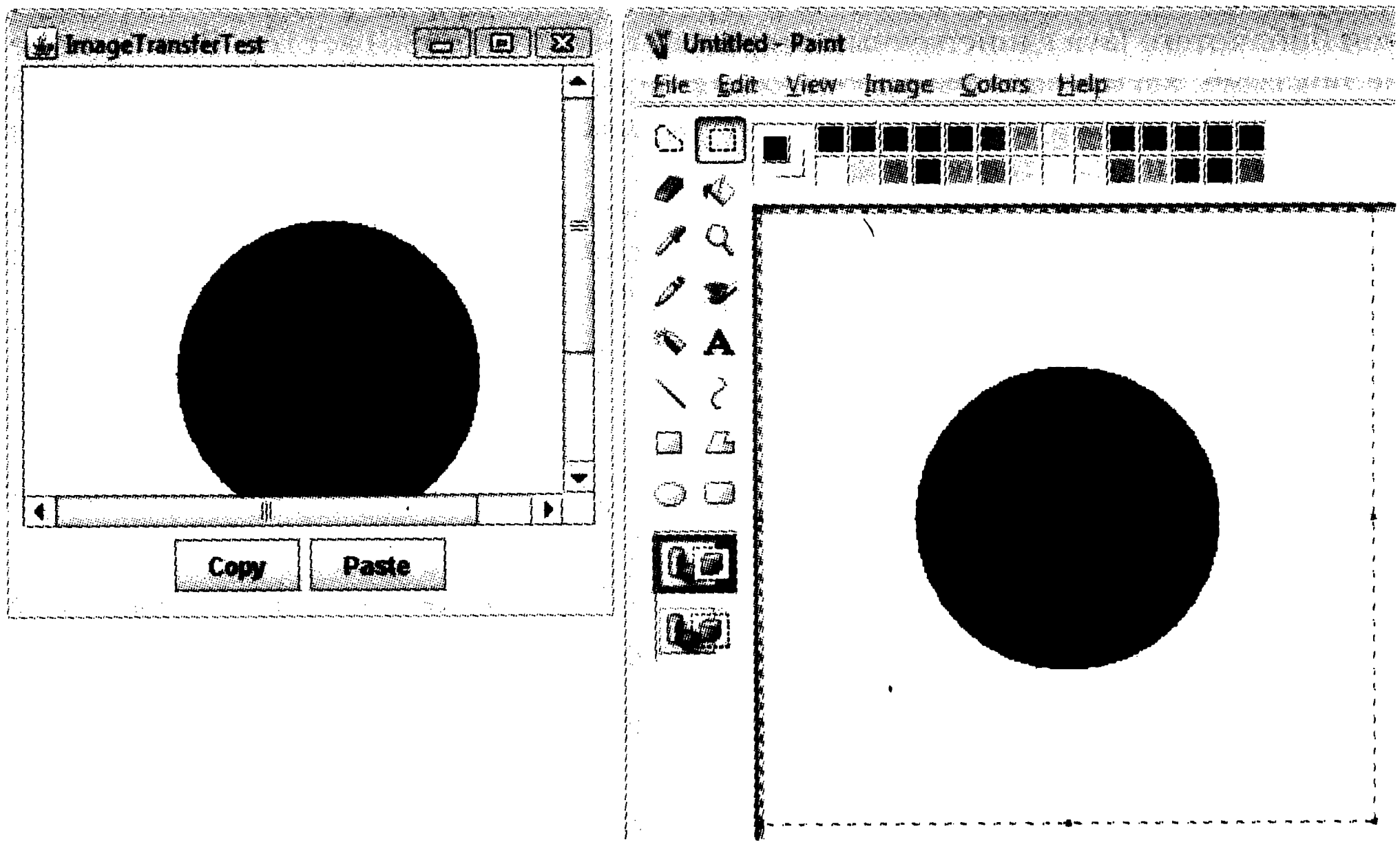


Рис. 7.41. Копирование изображения из прикладной программы на Java в платформенно-ориентированную программу

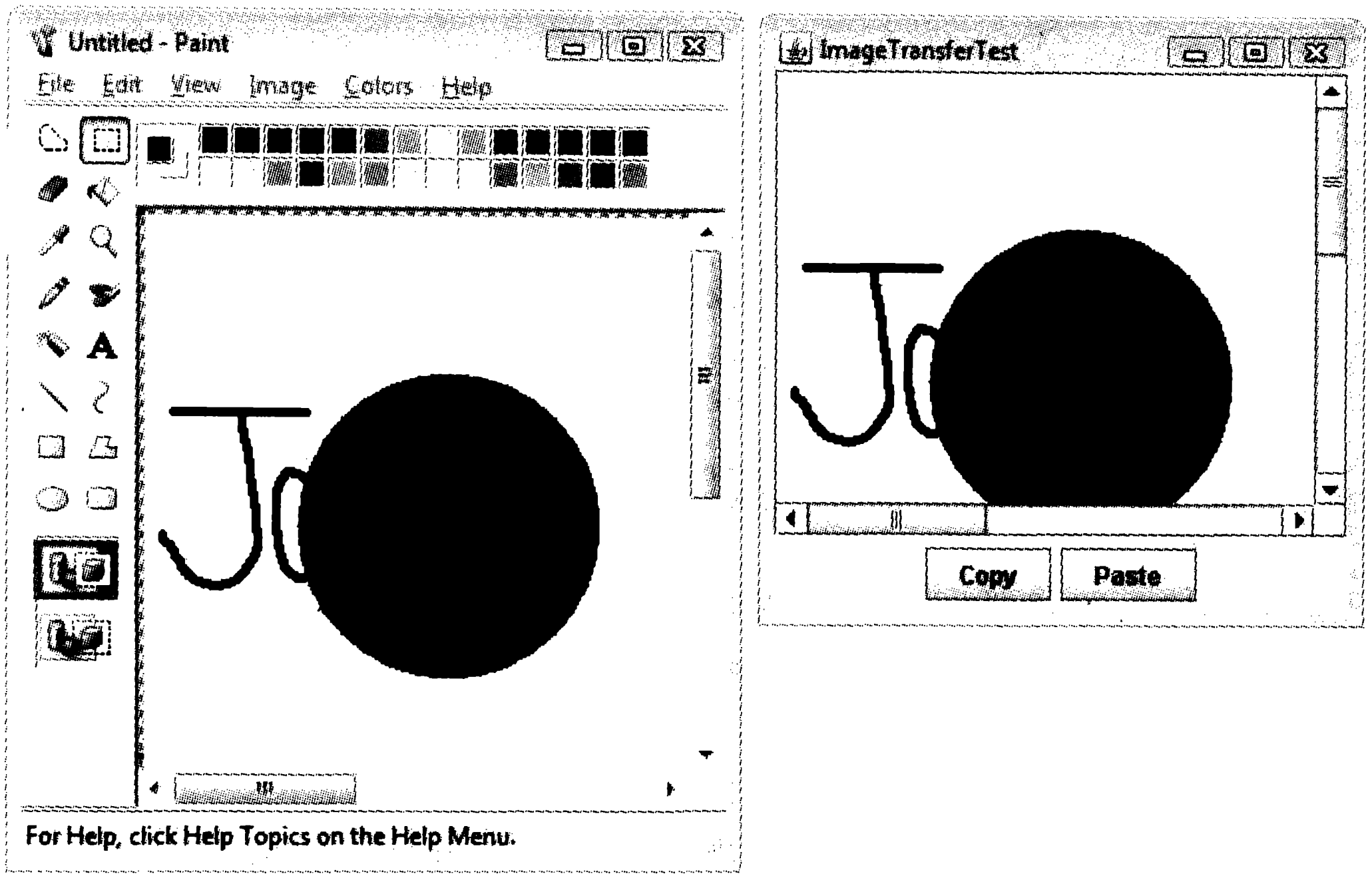


Рис. 7.42. Копирование изображения из платформенно-ориентированной программы в прикладную программу на Java

Данная программа является усовершенствованной версией программы из предыдущего примера, где демонстрировалась передача текста через буфер обмена. В данном же примере используется разновидность данных `DataFlavor.imageFlavor`, а для передачи изображения через системный буфер обмена — класс `ImageTransferable`.

Листинг 7.18. Исходный код из файла `imageTransfer/ImageTransferFrame.java`

```
1  package imageTransfer;
2
3  import java.awt.*;
4  import java.awt.datatransfer.*;
5  import java.awt.event.*;
6  import java.awt.image.*;
7  import java.io.*;
8  import javax.swing.*;
9
10 /**
11  * Этот фрейм содержит метку изображения и кнопки для копирования
12  * и вставки изображения
13 */
14 class ImageTransferFrame extends JFrame
15 {
16   private JLabel label;
17   private Image image;
18   private static final int IMAGE_WIDTH = 300;
19   private static final int IMAGE_HEIGHT = 300;
```

```
20
21    public ImageTransferFrame()
22    {
23       label = new JLabel();
24       image = new BufferedImage(IMAGE_WIDTH, IMAGE_HEIGHT,
25          BufferedImage.TYPE_INT_ARGB);
26       Graphics g = image.getGraphics();
27       g.setColor(Color.WHITE);
28       g.fillRect(0, 0, IMAGE_WIDTH, IMAGE_HEIGHT);
29       g.setColor(Color.RED);
30       g.fillOval(IMAGE_WIDTH / 4, IMAGE_WIDTH / 4,
31          IMAGE_WIDTH / 2, IMAGE_HEIGHT / 2);
32
33       label.setIcon(new ImageIcon(image));
34       add(new JScrollPane(label), BorderLayout.CENTER);
35       JPanel panel = new JPanel();
36
37       JButton copyButton = new JButton("Copy");
38       panel.add(copyButton);
39       copyButton.addActionListener(new ActionListener()
40          {
41             public void actionPerformed(ActionEvent event)
42             {
43                copy();
44             }
45          });
46       JButton pasteButton = new JButton("Paste");
47       panel.add(pasteButton);
48       pasteButton.addActionListener(new ActionListener()
49          {
50             public void actionPerformed(ActionEvent event)
51             {
52                paste();
53             }
54          });
55       add(panel, BorderLayout.SOUTH);
56       pack();
57    }
58
59    /**
60     * Копирует текущее изображение в системный буфер обмена
61    */
62    private void copy()
63    {
64       Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
65       ImageTransferable selection = new ImageTransferable(image);
66       clipboard.setContents(selection, null);
67    }
68
69    /**
70     * Вставляет изображение из системного буфера на место метки изображения
71    */
72    private void paste()
73    {
74       Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
75       DataFlavor flavor = DataFlavor.imageFlavor;
76       if (clipboard.isDataFlavorAvailable(flavor))
77       {
78          try
```

```
79          {
80            image = (Image) clipboard.getData(flavor);
81            label.setIcon(new ImageIcon(image));
82          }
83          catch (UnsupportedFlavorException exception)
84          {
85            JOptionPane.showMessageDialog(this, exception);
86          }
87          catch (IOException exception)
88          {
89            JOptionPane.showMessageDialog(this, exception);
90          }
91       }
92    }
93 }
```

Передача объектов Java через системный буфер обмена

Допустим, объекты требуется копировать и вставлять из одной прикладной программы на Java в другую. Для этого достаточно разместить в системном буфере обмена объекты, сериализированные средствами Java.

Такая возможность демонстрируется в примере программы из листинга 7.19. Данная программа отображает окно селектора цвета. Если щелкнуть на кнопке Copy, она скопирует в системный буфер обмена выбранный цвет в виде сериализированного объекта типа `Color`. А если щелкнуть на кнопке Paste, то данная программа проверит, содержится ли в системном буфере объект типа `Color`, и тогда он извлекается оттуда и становится текущим выбранным цветом.

Сериализированные объекты можно также передавать из одной прикладной программы на Java в другую (рис. 7.43). Чтобы убедиться в этом, запустите два экземпляра программы `SerialTransferTest`, затем щелкните на кнопке Copy в первом ее экземпляре и на кнопке Paste во втором. В итоге объект типа `Color` будет передан из одной виртуальной машины Java в другую.

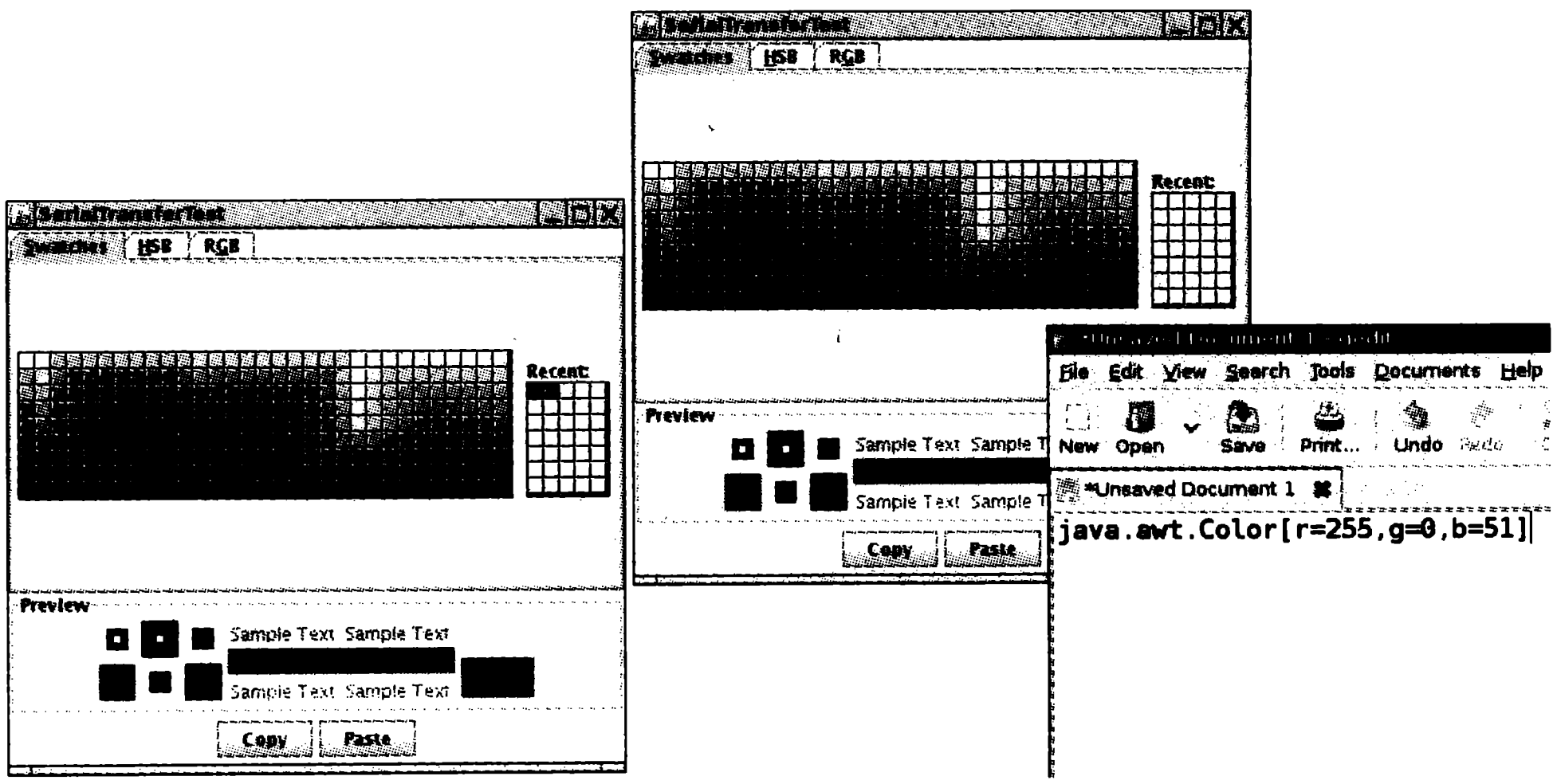


Рис. 7.43. Копирование данных между двумя экземплярами прикладной программы на Java

Для передачи данных на платформе Java предусмотрено размещение двоичных данных сериализированного объекта в системном буфере, чтобы другая программа на Java (не обязательно такого же типа, как и та, что сформировала данные для буфера обмена) смогла извлечь данные из буфера и десериализировать объект. Разумеется, программа, написанная не на Java, вряд ли будет знать, что делать с данными в буфере обмена. Именно поэтому в рассматриваемом здесь примере программы данные из буфера обмена предоставляются в другой разновидности: в текстовом виде. Текст получается в результате выполнения метода `toString()` для передаваемого объекта. Чтобы увидеть текст, запустите данную программу на выполнение, выберите какой-нибудь цвет и щелкните на кнопке **Paste** в текстовом редакторе. Это приведет к появлению в окне текстового редактора строки, аналогичной следующей:

```
java.awt.Color[r=255,g=0,b=51]
```

По существу, никакого дополнительного кода для передачи сериализируемых объектов не требуется. Для этой цели достаточно воспользоваться типом MIME следующим образом:

```
application/x-java-serialized-object;class=className
```

Класс-оболочку для передачи данных, как и прежде, придется создавать самостоятельно. В исходном коде из листинга 7.19 показано, как это делается.

Листинг 7.19. Исходный код из файла `serialTransfer/SerialTransferFrame.java`

```
1   package serialTransfer;
2
3   import java.awt.*;
4   import java.awt.datatransfer.*;
5   import java.awt.event.*;
6   import java.io.*;
7   import javax.swing.*;
8
9   /**
10   * Этот фрейм содержит селектор цвета и кнопки для копирования и вставки
11  */
12  class SerialTransferFrame extends JFrame
13  {
14    private JColorChooser chooser;
15
16    public SerialTransferFrame()
17    {
18      chooser = new JColorChooser();
19      add(chooser, BorderLayout.CENTER);
20      JPanel panel = new JPanel();
21
22      JButton copyButton = new JButton("Copy");
23      panel.add(copyButton);
24      copyButton.addActionListener(new ActionListener()
25        {
26          public void actionPerformed(ActionEvent event)
27          {
28            copy();
29          }
30        });
31      JButton pasteButton = new JButton("Paste");
32      panel.add(pasteButton);
33      pasteButton.addActionListener(new ActionListener()
```

```
34        {
35          public void actionPerformed(ActionEvent event)
36          {
37            paste();
38          }
39        });
40
41      add(panel, BorderLayout.SOUTH);
42      pack();
43    }
44
45    /**
46     * Копирует цвет из селектора в системный буфер обмена
47    */
48    private void copy()
49    {
50      Clipboard clipboard =
51         Toolkit.getDefaultToolkit().getSystemClipboard();
52       Color color = chooser.getColor();
53      SerialTransferable selection = new SerialTransferable(color);
54      clipboard.setContents(selection, null);
55    }
56
57    /**
58     * Вставляет цвет из системного буфера обмена в селектор цвета
59    */
60    private void paste()
61    {
62      Clipboard clipboard =
63         Toolkit.getDefaultToolkit().getSystemClipboard();
64      try
65      {
66        DataFlavor flavor = new DataFlavor(
67           "application/x-java-serialized-object;class=java.awt.Color");
68        if (clipboard.isDataFlavorAvailable(flavor))
69        {
70          Color color = (Color) clipboard.getData(flavor);
71          chooser.setColor(color);
72        }
73      }
75      catch (ClassNotFoundException e)
76      {
77        JOptionPane.showMessageDialog(this, e);
78      }
79      catch (UnsupportedFlavorException e)
80      {
81        JOptionPane.showMessageDialog(this, e);
82      }
83      catch (IOException e)
84      {
85        JOptionPane.showMessageDialog(this, e);
86      }
87    }
88  }
89
90  /**
91   * Этот класс служит оболочкой для передачи данных
92   * в виде сериализируемых объектов
93  */
```

```
94  class SerialTransferable implements Transferable
95  {
96    private Serializable obj;
97
98    /**
99     * Конструирует выбираемый объект
100    * @param o Любой сериализируемый объект
101   */
102   SerialTransferable(Serializable o)
103   {
104     obj = o;
105   }
106
107   public DataFlavor[] getTransferDataFlavors()
108   {
109     DataFlavor[] flavors = new DataFlavor[2];
110     Class<?> type = obj.getClass();
111     String mimeType = "application/x-java-serialized-object;class="
112        + type.getName();
113     try
114     {
115       flavors[0] = new DataFlavor(mimeType);
116       flavors[1] = DataFlavor.stringFlavor;
117       return flavors;
118     }
119     catch (ClassNotFoundException e)
120     {
121       return new DataFlavor[0];
122     }
123   }
124
125   public boolean isDataFlavorSupported(DataFlavor flavor)
126   {
127     return DataFlavor.stringFlavor.equals(flavor)
128       || "application".equals(flavor.getPrimaryType())
129       && "x-java-serialized-object".equals(flavor.getSubType())
130       && flavor.getRepresentationClass().isAssignableFrom(obj.getClass());
131   }
132   public Object getTransferData(DataFlavor flavor)
133      throws UnsupportedFlavorException
134   {
135     if (!isDataFlavorSupported(flavor))
136        throw new UnsupportedFlavorException(flavor);
137
138     if (DataFlavor.stringFlavor.equals(flavor)) return obj.toString();
139
140     return obj;
141   }
142 }
```

Передача ссылок на объекты через локальный буфер обмена

Иногда требуется копировать и вставлять данных таких типов, которые не поддерживаются системным буфером обмена, т.е. не являются сериализируемыми. Для передачи ссылки на произвольный объект в пределах одной и той же виртуальной машины Java (JVM) применяется тип MIME:

`application/x-java-jvm-local-objectref;class=`*`имяКласса`*

Для этого требуется определить объект-оболочку типа `Transferable`. Этот процесс аналогичен процессу определения объекта-оболочки типа `SerialTransferable`, рассмотренному в предыдущем разделе. Ссылка на объект имеет смысл только в пределах одной виртуальной машины. Поэтому копировать такой объект в системный буфер обмена нельзя. Вместо этого следует использовать локальный буфер обмена, как показано ниже. В качестве параметра конструктору приведенного ниже класса следует передать имя буфера обмена.

```
Clipboard clipboard = new Clipboard("local");
```

Но у передачи данных через локальный буфер обмена имеется один серьезный недостаток. Локальный и системный буферы обмена требуется синхронизировать, чтобы пользователи их не путали. На момент написания данной книги подобная синхронизация на платформе Java еще не поддерживалась.

java.awt.datatransfer.Clipboard 1.1

- `Clipboard(String name)`
  Создает локальный буфер обмена с указанным именем.

Перетаскивание объектов

При использовании механизма копирования и вставки для передачи данных между двумя программами в роли посредника всегда выступает буфер обмена. Механизм *перетаскивания* исключает участие такого посредника и позволяет двум программам взаимодействовать напрямую. На платформе Java предусмотрена элементарная поддержка данного механизма. В частности, она позволяет выполнять операции перетаскивания между программами на Java и платформенно-ориентированными программами. В этом разделе демонстрируется пример создания прикладной программы на Java для выполнения функций как источника, так и приемника перетаскивания.

Прежде чем углубляться в изучение поддержки операций перетаскивания, предоставляемой на платформе Java, рассмотрим вкратце пользовательский интерфейс перетаскивания на примере стандартных приложений Windows Explorer и WordPad. На другой платформе вместо этих приложений можно выбрать для экспериментирования другие стандартные прикладные программы с возможностями перетаскивания.

Операция перетаскивания инициируется выполнением соответствующего *жеста* в *источнике перетаскивания*, т.е. сначала выбирается один объект или более, а затем они перетаскиваются из исходного места в какое-нибудь другое место. После отпускания кнопки мыши над местом назначения, т.е. *приемником перетаскивания*, последний запрашивает у источника сведения о перетаскиваемых объектах и выполняет соответствующую операцию. Так, если перетащить пиктограмму файла на пиктограмму какого-нибудь каталога в окне диспетчера файлов, этот файл переместится в выбранный каталог. Но если перетащить пиктограмму файла в текстовый редактор, то этот файл откроется в окне текстового редактора. (Очевидно, что в данном случае подразумевается использование такого диспетчера файлов или текстового редактора, который поддерживает операции перетаскивания, как, например, стандартные приложения Explorer и WordPad в Windows или Nautilus и gedit в Gnome.)

Если при перетаскивании файла в Windows нажать дополнительно клавишу <Ctrl>, вместо операции *перемещения* будет выполнена операция *копирования* в каталог только копии файла. А если *одновременно* нажать клавиши <Ctrl> и <Shift>, то в каталоге будет создан ярлык в качестве *ссылки* на данный файл. (На других платформах для выполнения этих операций могут быть предусмотрены другие комбинации клавиш.)

Таким образом, операции перетаскивания могут подразумевать выполнение трех различных действий:

- перемещение;
- копирование;
- создание ярлыка.

Под созданием ярлыка, по существу, подразумевается создание ссылки на перетаскиваемый объект. Такие ссылки обычно требуют поддержки со стороны обслуживающей операционной системы (вроде символических ссылок на файлы или объектных ссылок на компоненты документа) и обычно не имеют особого смысла в межплатформенных программах. Поэтому в этом разделе основное внимание уделяется применению только тех операций перетаскивания, которые подразумевают копирование и перемещение объектов.

Обычно операции перетаскивания сопровождаются специальными визуальными эффектами. Самым простым из них является изменение вида курсора мыши. При наведении курсора на потенциальный *приемник перетаскивания* его вид изменяется, указывая на возможность выполнения операции перетаскивания. Если перетаскивание возможно, то вид курсора дополнительно указывает, какое именно действие разрешается выполнить. В табл. 7.6 представлены некоторые виды, которые курсор может принимать при перетаскивании.

Таблица 7.6. Виды курсора, которые он может принимать при перетаскивании

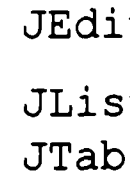
Действие	Значок в Windows	Значок в Gnome
Перемещение		
Копирование		
Создание ярлыка		
Перетаскивание запрещено		

Перетаскивать можно не только пиктограммы файлов, но и другие объекты. Например, можно выделить фрагмент текста в редакторе WordPad или gedit и перетащить его. Попробуйте перетащить фрагменты текста в разные места назначения и посмотрите, что при этом произойдет.

НА ЗАМЕТКУ! Подобный эксперимент демонстрирует недостатки механизма перетаскивания как части пользовательского интерфейса. Он может вызывать у пользователей сомнения следующего характера: какие объекты и куда можно перетаскивать и к чему это приведет. Вследствие того что операция перемещения, выполняемая по умолчанию при перетаскивании, может приводить к удалению оригинала, многие пользователи, по вполне понятным причинам, стараются не особенно экспериментировать с перетаскиванием.

Поддержка передачи данных в Swing

Начиная с версии Java SE 1.4 в ряд компонентов Swing встроена поддержка операций перетаскивания (табл. 7.7). Одни из этих компонентов позволяют выбирать текст для перетаскивания, а другие — вставлять его. Для обеспечения обратной совместимости следует вызвать метод `setDragEnabled()`, активизирующий перетаскивание, тогда как опускание активно всегда.



НА ЗАМЕТКУ! В состав пакета `java.awt.dnd` входит низкоуровневый прикладной интерфейс API для перетаскивания. Он составляет основу для поддержки перетаскивания в библиотеке Swing. Этот прикладной интерфейс в данной книге не рассматривается.

Таблица 7.7. Поддержка передачи данных в компонентах Swing

Компонент	Источник перетаскивания	Приемник перетаскивания
`JFileChooser`	Экспортирует список файлов	Отсутствует
`JColorChooser`	Экспортирует объект, представляющий цвет	Принимает объекты, представляющие разные цвета
`JTextField` `JFormattedTextField`	Экспортирует выбранный текст	Принимает текст
`JPasswordField`	Отсутствует (из соображений безопасности)	Принимает текст

`JTextArea`
`JTextPane`
`JEditorPane`
	Экспортирует выбранный текст	Принимает текст и списки файлов

`JList`
`JTable`
`JTree`
	Экспортирует описание выбранного текста (т.е. только его копию)	Отсутствует

В примере программы, исходный код которой приведен в листинге 7.20, демонстрируется поведение всех этих компонентов. После запуска этой программы на выполнение обратите внимание на ряд моментов.

- Из списка, таблицы и дерева можно выбирать и перетаскивать несколько элементов, как следует из листинга 7.21.
- Перетаскивать элементы из таблицы не совсем удобно: сначала нужно выбрать элемент мышью, затем отпустить кнопку мыши и снова щелкнуть ею и только затем выполнить перетаскивание.
- При перетаскивании элементов в текстовую область можно заметить, как происходит форматирование перетаскиваемых данных. В частности, ячейки таблицы разделяются символами табуляции, а каждая выбранная строка таблицы размещается в отдельной строке (рис. 7.44).
- Из списка, таблицы, селектора файлов и селектора цветов элементы можно только копировать, но не перемещать. Удаление элементов из списка, таблицы и дерева не допускается ни в одной из моделей данных. О том, как реализовать возможность редактирования в модели данных, речь пойдет в следующем разделе.

- Перетаскивать элементы в список, таблицу, дерево или селектор файлов нельзя.
- Если запустить на выполнение две копии программы, то образец цвета можно перетаскивать из одного селектора цвета в другой.
- Перетаскивать текст из текстовой области нельзя, поскольку для нее не вызывается метод `setDragEnabled()`.

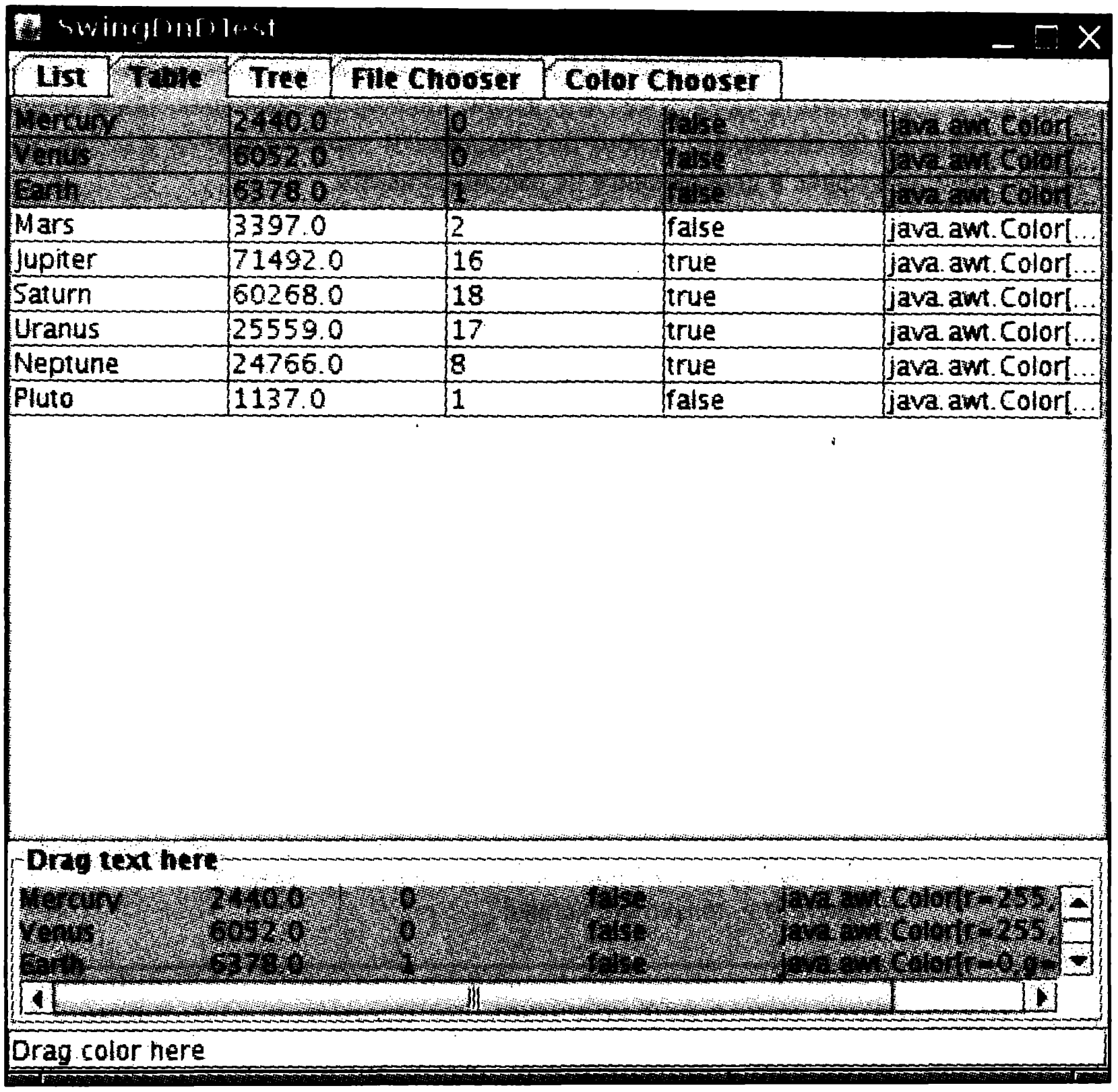


Рис. 7.44. Программа для тестирования поддержки перетаскивания в библиотеке Swing

В библиотеке Swing предусмотрен потенциально полезный механизм для быстрого превращения компонента из источника перетаскивания в приемник. С этой целью можно установить *обработчик передачи данных* для заданного свойства. Так, в рассматриваемом здесь примере программы делается следующий вызов:

```
textField.setTransferHandler(new TransferHandler("background"));
```

Благодаря этому перетаскивание цвета в текстовое поле будет приводить к изменению цвета его фона. При перетаскивании этот обработчик будет проверять, имеется ли у одной из разновидностей данных класс представления `Color`. И если таковой имеется, то вызывается метод `setBackground()`.

Установка упомянутого выше обработчика передачи данных в текстовом поле означает отключение стандартного обработчика передачи данных, из-за чего в этом текстовом поле больше нельзя будет ни вырезать, ни вставлять, ни перетаскивать текст,

но в то же время из него можно будет перетаскивать цвет. Для того чтобы инициировать жест перетаскивания, придется выбрать какой-нибудь фрагмент текста. При перетаскивании этого текста обнаружится, что его можно опустить в селекторе цвета, изменив цвет в его образце на цвет фона текстового поля. Но перетащить текст в саму текстовую область не удастся.

Листинг 7.20. Исходный код из файла dnd/SwingDnDTest.java

```
1  package dnd;
2
3  import java.awt.*;
4  import javax.swing.*;
5  /**
6   * В этой программе демонстрируется элементарная поддержка операций
7   * перетаскивания и опускания объектов в библиотеке Swing
8   * @version 1.10 2007-09-20
9   * @author Cay Horstmann
10 */
11 public class SwingDnDTest
12 {
13   public static void main(String[] args)
14   {
15     EventQueue.invokeLater(new Runnable()
16       {
17         public void run()
18        {
19          JFrame frame = new SwingDnDFrame();
20          frame.setTitle("SwingDnDTest");
21          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22          frame.setVisible(true);
23        }
24      });
25   }
26 }
```

Листинг 7.21. Исходный код из файла dnd/SampleComponents.java

```
1  package dnd;
2
3  import java.awt.*;
4
5  import javax.swing.*;
6  import javax.swing.tree.*;
7
8  public class SampleComponents
9  {
10   public static JTree tree()
11   {
12     DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
13     DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
14     root.add(country);
15     DefaultMutableTreeNode state =
16        new DefaultMutableTreeNode("California");
17     country.add(state);
18     DefaultMutableTreeNode city =
```

```
19          new DefaultMutableTreeNode("San Jose");
20       state.add(city);
21       city = new DefaultMutableTreeNode("Cupertino");
22       state.add(city);
23       state = new DefaultMutableTreeNode("Michigan");
24       country.add(state);
25       city = new DefaultMutableTreeNode("Ann Arbor");
26       state.add(city);
27       country = new DefaultMutableTreeNode("Germany");
28       root.add(country);
29       state = new DefaultMutableTreeNode("Schleswig-Holstein");
30       country.add(state);
31       city = new DefaultMutableTreeNode("Kiel");
32       state.add(city);
33       return new JTree(root);
34    }
35
36    public static JList<String> list()
37    {
38       String[] words = { "quick", "brown", "hungry", "wild", "silent",
39          "huge", "private", "abstract", "static", "final" };
40
41    DefaultListModel<String> model = new DefaultListModel<>();
42    for (String word : words)
43       model.addElement(word);
44    return new JList<>(model);
45    }
46
47    public static JTable table()
48    {
49       Object[][] cells = { { "Mercury", 2440.0, 0, false, Color.YELLOW },
50          { "Venus", 6052.0, 0, false, Color.YELLOW },
51          { "Earth", 6378.0, 1, false, Color.BLUE },
52          { "Mars", 3397.0, 2, false, Color.RED },
53          { "Jupiter", 71492.0, 16, true, Color.ORANGE },
54          { "Saturn", 60268.0, 18, true, Color.ORANGE },
55          { "Uranus", 25559.0, 17, true, Color.BLUE },
56          { "Neptune", 24766.0, 8, true, Color.BLUE },
57          { "Pluto", 1137.0, 1, false, Color.BLACK } };
58
59          String[] columnNames =
60             { "Planet", "Radius", "Moons", "Gaseous", "Color" };
61          return new JTable(cells, columnNames);
62       }
63 }
```

javax.swing.JComponent 1.2

- `void setTransferHandler(TransferHandler handler)` **1.4**

  Устанавливает обработчик для управления операциями передачи данных (т.е. операциями вырезания, копирования, вставки и перетаскивания).

javax.swing.TransferHandler 1.4

- `TransferHandler(String` *propertyName*`)`

  Создает обработчик передачи данных для чтения или записи свойства компонента JavaBeans с указанным именем при выполнении операции передачи данных.

javax.swing.JFileChooser 1.2
javax.swing.JColorChooser 1.2
javax.swing.text.JTextComponent 1.2
javax.swing.JList 1.2
javax.swing.JTable 1.2
javax.swing.JTree 1.2

- `void setDragEnabled(boolean` *b*`)` **1.4**

  Включает или отключает перетаскивание данных из соответствующего компонента.

Источники перетаскивания

В предыдущем разделе было показано, как пользоваться элементарными средствами перетаскивания, которые поддерживаются в Swing. А в этом разделе поясняется, как настроить любой компонент, чтобы он выполнял функции источника перетаскивания. Что же касается приемников перетаскивания, то они будут рассмотрены в следующем разделе, где также приводится пример реализации компонента, выполняющего одновременно функции источника и приемника изображений.

Для того чтобы настроить поведение компонента Swing на перетаскивание, следует создать новый подкласс, производный от класса `TransferHandler`. В этом подклассе нужно сначала переопределить метод `getSourceActions()`, чтобы он указывал, какие именно действия (копирование, перемещение, создание ярлыка) должен поддерживать компонент, а затем — метод `getTransferable()`, чтобы он выдавал объект типа `Transferable` таким же образом, как и при копировании данных в буфер обмена.

В рассматриваемом здесь примере программы изображения должны перетаскиваться из компонента `JList`, заполняемого пиктограммами изображений (рис. 7.45). В приведенном ниже фрагменте кода показано, каким образом метод `createTransferable()` реализуется в данной программе. При этом выбранное изображение заключается в оболочку объекта типа `ImageTransferable`.

```
protected Transferable createTransferable(JComponent source)
{
  JList list = (JList) source;
  int index = list.getSelectedIndex();
  if (index < 0) return null;
  ImageIcon icon = (ImageIcon) list.getModel().getElementAt(index);
  return new ImageTransferable(icon.getImage());
}
```

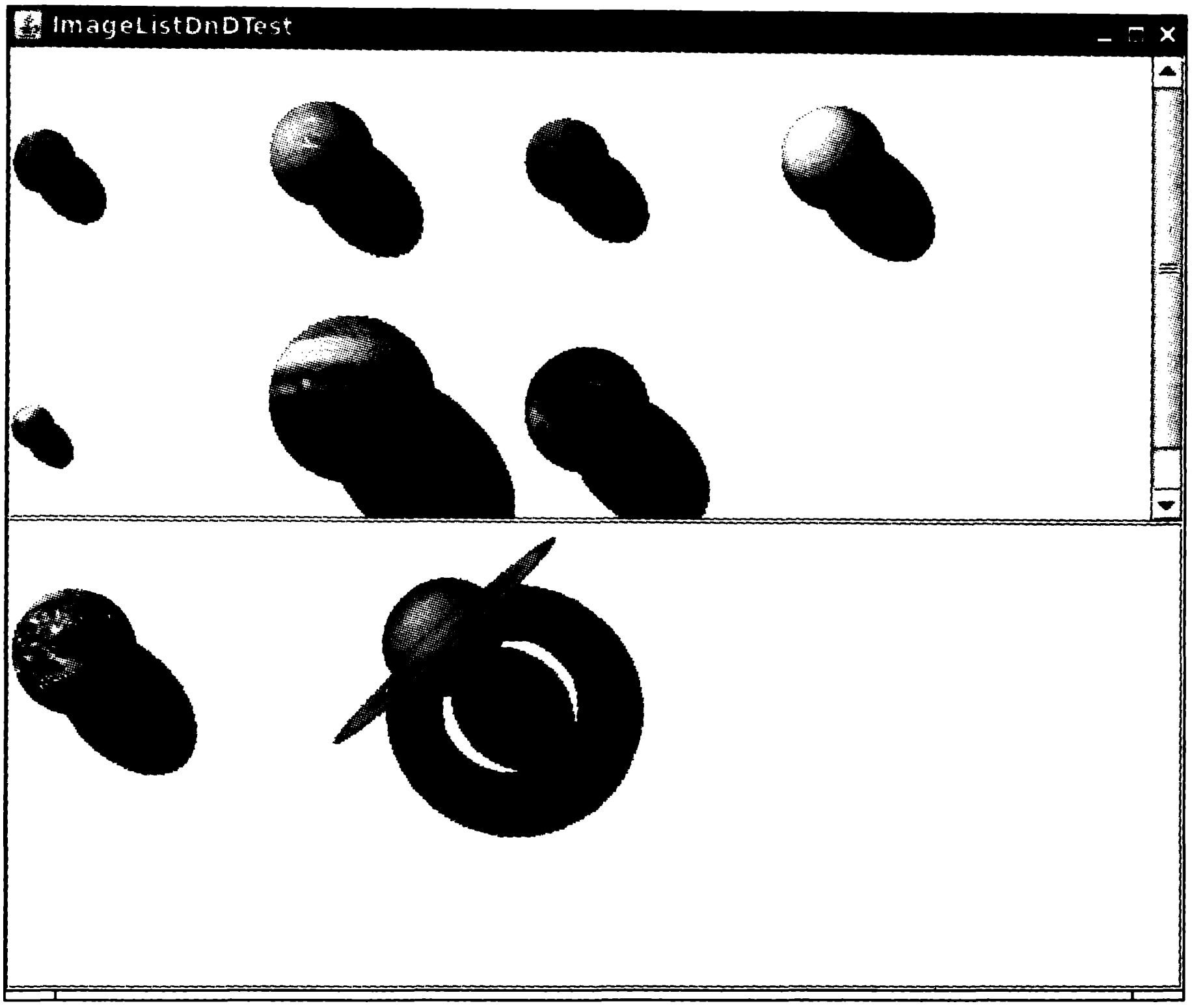


Рис. 7.45. Прикладная программа `ImageList`, демонстрирующая перетаскивание изображений

В рассматриваемом здесь примере программы применяется компонент `JList`, который, к счастью, уже предусматривает возможность распознавания жеста перетаскивания. Остается только активизировать соответствующий механизм передачи данных, вызвав метод `setDragEnabled()`. Если же требуется внедрить поддержку перетаскивания в компонент, который не способен распознавать жест перетаскивания, то передачу данных придется инициировать самостоятельно. В качестве примера в приведенном ниже фрагменте кода показано, как инициировать перетаскивание в компоненте `JLabel`.

```
label.addMouseListener(new MouseAdapter()
{
  public void mousePressed(MouseEvent evt)
  {
    int mode;
    if ((evt.getModifiers() &
       (InputEvent.CTRL_MASK | InputEvent.SHIFT_MASK)) != 0)
        mode = TransferHandler.COPY;
    else mode = TransferHandler.MOVE;
    JComponent comp = (JComponent) evt.getSource();
    TransferHandler th = comp.getTransferHandler();
    th.exportAsDrag(comp, evt, mode);
  }
});
```

В этом фрагменте кода передача данных инициируется, как только пользователь щелкнет на метке. В более сложной реализации перетаскивания можно было бы предусмотреть отслеживание движений мыши при перетаскивании на небольшое расстояние.

По завершении пользователем операции перетаскивания вызывается метод `exportDone()` обработчика передачи данных от источника. Как только пользователь завершит операцию перемещения, в этом методе должно быть произведено удаление переданного объекта. Следовательно, для списка изображений код реализации этого метода должен выглядеть следующим образом:

```
protected void exportDone(JComponent source, Transferable data, int action)
{
  if (action == MOVE)
  {
    JList list = (JList) source;
    int index = list.getSelectedIndex();
    if (index < 0) return;
    DefaultListModel model = (DefaultListModel) list.getModel();
    model.remove(index);
  }
}
```

Таким образом, для превращения компонента в источник перетаскивания достаточно ввести обработчик передачи данных, определяющий следующее:

- какие действия поддерживаются;
- какие данные передаются;
- каким образом исходные данные удаляются после операции перемещения.

Кроме того, если в источник перетаскивания превращается компонент, не входящий в список компонентов, перечисленных в табл. 7.7, придется дополнительно организовать отслеживание движений мыши (жестов) и инициирование передачи данных.

javax.swing.TransferHandler 1.4

- `int getSourceActions(JComponent c)`

  Должен быть переопределен таким образом, чтобы возвращать (в виде поразрядной операции ИЛИ из комбинации констант `COPY`, `MOVE` и `LINK`) сведения о тех действиях, которые источнику разрешено выполнять при перетаскивании из указанного компонента.

- `protected Transferable createTransferable(JComponent source)`

  Должен быть переопределен таким образом, чтобы создавать объект типа `Transferable` для перетаскиваемых данных.

- `void exportAsDrag(JComponent comp, InputEvent e, int action)`

  Начинает операцию перетаскивания из указанного компонента. В качестве параметра *action* может быть указано значение одной из следующих констант: `COPY`, `MOVE` или `LINK`.

- `protected void exportDone(JComponent source, Transferable data, int action)`

  Должен быть переопределен таким образом, чтобы настраивать источник перетаскивания соответствующим образом после успешного завершения передачи данных.

Приемники перетаскивания

В этом разделе поясняется, как реализовать приемник перетаскивания. Для примера снова выбран компонент `JList` с пиктограммами изображений. В данном примере требуется предоставить пользователям возможность перетаскивать изображения в список. Для того чтобы превратить компонент в приемник перетаскивания, следует установить приемник передачи данных типа `TransferHandler` и реализовать методы `canImport()` и `importData()`.

НА ЗАМЕТКУ! Обработчик передачи данных можно ввести и в компонент `JFrame`. Именно этот компонент чаще всего применяется в прикладных программах для опускания объектов (например, файлов) после их перетаскивания. К числу допустимых мест опускания перетаскиваемых объектов относятся элементы оформления фрейма и строка меню, но не компоненты внутри фрейма, поскольку у них имеются свои обработчики передачи данных.

Метод `canImport()` вызывается непрерывно, когда пользователь наводит курсор мыши на компонент, выполняющий функции приемника перетаскивания. Если опускание перетаскиваемого объекта разрешается, этот метод возвращает логическое значение `true`. Именно эти сведения и определяют внешний вид пиктограммы курсора, наглядно показывающий, разрешается ли выполнение операции отпускания.

У метода `canImport()` имеется параметр типа `TransferHandler.TransferSupport`. Этот параметр позволяет получить сведения о действии опускания перетаскиваемого объекта, выбранном пользователем, о месте опускания и, наконец, о данных, которые требуется передать после опускания. (До версии Java SE 6 существовала другая разновидность метода `canImport()`, позволявшая получать только перечень разновидностей данных.)

В методе `canImport()` имеется также возможность переопределить выбранное пользователем действие опускания перетаскиваемого объекта. Так, если пользователь выбрал в качестве такого действия перемещение, но оно не подходит для удаления оригинала, можно принудить обработчик передачи данных выполнить вместо него копирование.

Ниже приведен типичный пример. Компонент списка изображений готов принимать опускаемые на него списки перетаскиваемых файлов и изображений. Но если в этот компонент перетаскивается список файлов, то выбранное пользователем действие `MOVE` (Перемещение) заменяется действием `COPY` (Копирование), чтобы перетаскиваемые файлы изображений не были удалены.

```
public boolean canImport(TransferSupport support)
{
  if (support.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
  {
    if (support.getUserDropAction() == MOVE) support.setDropAction(COPY);
    return true;
  }
  else return support.isDataFlavorSupported(DataFlavor.imageFlavor);
}
```

В более сложной реализации могла бы дополнительно выполняться проверка, действительно ли в файлах содержатся изображения. В таких компонентах Swing, как `JList`, `JTable`, `JTree` и `JTextComponent`, предусмотрено отображение визуальной подсказки для места опускания перетаскиваемого объекта при наведении курсора

мыши на поверхность приемника перетаскивания. По умолчанию для обозначения места опускания применяется выделение (если речь идет о компонентах `JList`, `JTable` и `JTree`) либо символ вставки (если речь идет о компоненте `TextComponent`). Но такой способ не совсем удобен для пользователя и применяется по умолчанию только в целях обратной совместимости. Для выбора более подходящей визуальной подсказки следует вызывать метод `setDropMode()`.

Имеется также возможность определить, должны ли опускаемые объекты перезаписывать существующие объекты или быть вставлены между ними. Например, в рассматриваемом здесь примере программы для предоставления пользователю возможности опустить перетаскиваемый объект прямо на существующий элемент (т.е. заменить его) или вставить объект между двумя элементами (рис. 7.46) выполняется приведенный ниже вызов. В табл. 7.8 перечислены все режимы опускания перетаскиваемых объектов, которые поддерживаются компонентами Swing.

```
setDropMode(DropMode.ON_OR_INSERT);
```

Таблица 7.8. Режимы опускания перетаскиваемых объектов

Компонент	Поддерживаемые режимы опускания
`JList`, `JTree`	`ON`, `INSERT`, `ON_OR_INSERT`, `USE_SELECTION`
`JTable`	`ON`, `INSERT`, `ON_OR_INSERT`, `INSERT_ROWS`, `INSERT_COLS`, `ON_OR_INSERT_ROWS`, `ON_OR_INSERT_COLS`, `USE_SELECTION`
`JTextComponent`	`INSERT`, `USE_SELECTION` (фактически перемещает знак вставки, а не выделенный фрагмент текста)

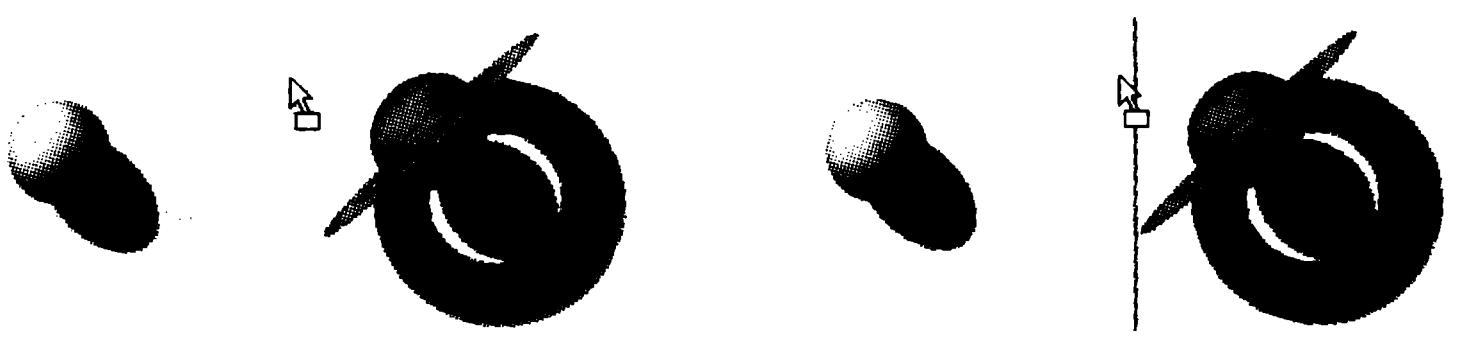


Рис. 7.46. Визуальные подсказки для опускания перетаскиваемого объекта прямо на существующий элемент или вставки объекта между двумя элементами

По завершении пользователем операции опускания перетаскиваемого объекта немедленно вызывается метод `importData()`, требующий получения данных из источника перетаскивания. А в результате вызова метода `getTransferable()` для объекта типа `TransferSupport` получается ссылка на объект типа `Transferable`, который предоставляет тот же самый интерфейс, что и для копирования и вставки объектов.

Для перетаскивания и опускания чаще всего выбирается разновидность данных `DataFlavor.javaFileListFlavor`. Список файлов служит для описания файлов, перетаскиваемых в приемник. Передаваемые данные вводятся в объект типа `List<File>`. Ниже приведен код, требующийся для извлечения файлов из данного списка.

```
DataFlavor[] flavors = transferable.getTransferDataFlavors();
if (Arrays.asList(flavors).contains(DataFlavor.javaFileListFlavor))
{
  List<File> fileList = (List<File>)
     transferable.getTransferData(DataFlavor.javaFileListFlavor);
  for (File f : fileList)
  {
```

```
    сделать что-нибудь с файлом f;
  }
}
```

При опускании перетаскиваемого объекта на какой-нибудь из тех компонентов, которые перечислены в табл. 7.8, требуется точно знать место опускания. Для того чтобы выяснить это место, можно вызвать метод `getDropLocation()` для объекта типа `TransferSupport`. Этот метод возвращает объект подкласса `TransferHandler.DropLocation`. Классы `JList`, `JTable`, `JTree` и `JTextComponent` определяют подклассы, описывающие местоположение в соответствующей модели данных. Например, в списке местоположение описывается просто целочисленным индексом, тогда как в дереве — в пути по дереву. Ниже приведен код, требующийся для выяснения места опускания перетаскиваемых объектов в списке изображений.

```
int index;
if (support.isDrop())
{
  JList.DropLocation location =
    (JList.DropLocation) support.getDropLocation();
   index = location.getIndex();
}
else index = model.size();
```

В подклассе `JList.DropLocation` имеется метод `getIndex()`, возвращающий индекс места опускания. (А в подклассе `JTree.DropLocation` для выполнения той же самой функции предусмотрен метод `getPath()`.)

Метод `importData()` вызывается и при вставке данных в компонент нажатием комбинации клавиш <Ctrl+V>. В таком случае метод `getDropLocation()` сгенерирует исключение типа `IllegalStateException`. Но чтобы этого не произошло, вставляемые данные просто вводятся в конце списка, если метод `isDrop()` возвращает логическое значение `false`.

Кроме того, при вставке данных в список, таблицу или дерево следует непременно проверить, требуется ли вставить эти данные между элементами или заменить ими тот элемент, который находится в месте их опускания. В списке такую проверку можно организовать, вызвав метод `isInsert()` из подкласса `JList.DropLocation`. Что касается остальных компонентов, то сведения о требующихся для них методах и подклассах можно найти в приведенном далее описании прикладного интерфейса API.

Таким образом, для того чтобы превратить компонент в приемник перетаскивания, достаточно ввести обработчик передачи данных, определяющий следующее:

- когда перетаскиваемый элемент может быть принят;
- каким образом должны импортироваться опускаемые данные.

Если поддержка опускания перетаскиваемых объектов внедряется в компонент `JList`, `JTable`, `JTree` или `JTextComponent`, следует также установить соответствующий режим опускания.

В листинге 7.22 приведен исходный код программы, демонстрирующей перетаскивание и опускание объектов. Обратите внимание на то, что класс `ImageList` одновременно выполняет функции источника и приемника перетаскивания. Попробуйте перетащить изображения из одного списка в другой, а также списки файлов изображений из селектора файлов какой-нибудь другой программы.

Листинг 7.22. Исходный код из файла dndImage/imageListDnDFrame.java

```
1   package dndImage;
2
3   import java.awt.*;
4   import java.awt.datatransfer.*;
5   import java.io.*;
6   import java.nio.file.*;
7   import java.util.*;
8   import java.util.List;
9   import javax.imageio.*;
10  import javax.swing.*;
11
12  public class ImageListDnDFrame extends JFrame
13  {
14    private static final int DEFAULT_WIDTH = 600;
15    private static final int DEFAULT_HEIGHT = 500;
16
17    private ImageList list1;
18    private ImageList list2;
19
20    public ImageListDnDFrame()
21    {
22      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24      list1 = new ImageList(Paths.get(getClass().getPackage().getName(),
25         "images1"));
26      list2 = new ImageList(Paths.get(getClass().getPackage().getName(),
27         "images2"));
28
29      setLayout(new GridLayout(2, 1));
30      add(new JScrollPane(list1));
31      add(new JScrollPane(list2));
32    }
33  }
34
35  class ImageList extends JList<ImageIcon>
36  {
37    public ImageList(Path dir)
38    {
39      DefaultListModel<ImageIcon> model = new DefaultListModel<>();
40      try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir))
41      {
42        for (Path entry : entries)
43        model.addElement(new ImageIcon(entry.toString()));
44      }
45      catch (IOException ex)
46      {
47        ex.printStackTrace();
48      }
49
50      setModel(model);
51      setVisibleRowCount(0);
52      setLayoutOrientation(JList.HORIZONTAL_WRAP);
53      setDragEnabled(true);
54      setDropMode(DropMode.ON_OR_INSERT);
55      setTransferHandler(new ImageListTransferHandler());
56    }
```

```
57  }
58
59  class ImageListTransferHandler extends TransferHandler
60  {
61    // поддержка перетаскивания
62
63    public int getSourceActions(JComponent source)
64    {
65      return COPY_OR_MOVE;
66    }
67
68    protected Transferable createTransferable(JComponent source)
69    {
70      ImageList list = (ImageList) source;
71      int index = list.getSelectedIndex();
72      if (index < 0) return null;
73      ImageIcon icon = list.getModel().getElementAt(index);
74      return new ImageTransferable(icon.getImage());
75    }
76
77    protected void exportDone(JComponent source, Transferable data,
78       int action)
79    {
80      if (action == MOVE)
81      {
82        ImageList list = (ImageList) source;
83        int index = list.getSelectedIndex();
84        if (index < 0) return;
85        DefaultListModel<?> model = (DefaultListModel<?>) list.getModel();
86        model.remove(index);
87      }
88    }
89
90    // поддержка опускания
91
92    public boolean canImport(TransferSupport support)
93    {
94      if (support.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
95      {
96        if (support.getUserDropAction() == MOVE)
97          support.setDropAction(COPY);
98        return true;
99      }
100     else return support.isDataFlavorSupported(DataFlavor.imageFlavor);
101   }
102
103   public boolean importData(TransferSupport support)
104   {
105     ImageList list = (ImageList) support.getComponent();
106     DefaultListModel<ImageIcon> model =
107        (DefaultListModel<ImageIcon>) list.getModel();
108
109     Transferable transferable = support.getTransferable();
110     List<DataFlavor> flavors =
111        Arrays.asList(transferable.getTransferDataFlavors());
112
113     List<Image> images = new ArrayList<>();
114
115     try
```

```
116     {
117       if (flavors.contains(DataFlavor.javaFileListFlavor))
118       {
119         @SuppressWarnings("unchecked") List<File> fileList =
120             (List<File>) transferable.getTransferData(
121                DataFlavor.javaFileListFlavor);
122         for (File f : fileList)
123         {
124           try
125           {
126             images.add(ImageIO.read(f));
127           }
128           catch (IOException ex)
129           {
130             // не удалось прочитать изображение — пропустить
131           }
132         }
133       }
134       else if (flavors.contains(DataFlavor.imageFlavor))
135       {
136         images.add((Image) transferable.getTransferData(
137            DataFlavor.imageFlavor));
138       }
139
140       int index;
141       if (support.isDrop())
142       {
143         JList.DropLocation location =
144            (JList.DropLocation) support.getDropLocation();
145         index = location.getIndex();
146         if (!location.isInsert()) model.remove(index); // сменить место
147       }
148       else index = model.size();
149       for (Image image : images)
150       {
151         model.add(index, new ImageIcon(image));
152         index++;
153       }
154       return true;
155     }
156     catch (IOException ex)
157     {
158       return false;
159     }
160     catch (UnsupportedFlavorException ex)
161     {
162       return false;
163     }
164   }
165 }
```

javax.swing.TransferHandler 1.4

- `boolean canImport(TransferSupport` *support*`)` **6**

  Переопределяется таким образом, чтобы указывать, может ли целевой компонент принимать перетаскиваемый объект, описываемый параметром *support*.

- `boolean importData(TransferSupport` *`support`*`)` **6**

  Переопределяется таким образом, чтобы выполнять только операцию опускания или вставки, описываемую параметром *`support`*, и возвращать логическое значение `true`, если импорт завершен успешно.

`javax.swing.JFrame` 1.2

- `void setTransferHandler(TransferHandler` *`handler`*`)` **6**

  Устанавливает обработчик передачи данных для обработки только операций опускания и вставки.

`javax.swing.JList` 1.2
`javax.swing.JTable` 1.2
`javax.swing.JTree` 1.2
`javax.swing.text.JTextComponent` 1.2

- `void setDropMode(DropMode` *`mode`*`)` **6**

  Устанавливает для режима опускания данного компонента одно из значений, перечисленных в табл. 7.8.

`javax.swing.TransferHandler.TransferSupport` 6

- `Component getComponent()`

  Получает сведения о целевом компоненте в текущей операции передачи данных.

- `DataFlavor[] getDataFlavors()`

  Получает сведения о разновидностях передаваемых данных.

- `boolean isDrop()`

  Возвращает логическое значение `true`, если текущая операция передачи данных подразумевает опускание, или логическое значение `false`, если она подразумевает вставку.

- `int getUserDropAction()`

  Получает сведения о действии, выбранном пользователем для опускания: `MOVE`, `COPY` или `LINK`.

- `getSourceDropActions()`

  Получает сведения о тех действиях опускания, которые разрешаются источником перетаскивания.

- `getDropAction()`
- `setDropAction()`

  Получают или устанавливают действие опускания в текущей операции передачи данных. Первоначально это действие выбирается пользователем, но затем оно может переопределяться обработчиком передачи данных.

- `DropLocation getDropLocation()`

  Получает сведения о месте опускания или генерирует исключение типа `IllegalStateException`, если текущая операция передачи данных не подразумевает опускание.

javax.swing.TransferHandler.DropLocation 6

- `Point getDropPoint()`

  Получает сведения о местоположении курсора мыши при опускании в целевом компоненте.

javax.swing.JList.DropLocation 6

- `boolean isInsert()`

  Возвращает логическое значение `true`, если данные должны быть вставлены перед указанным местом, или логическое значение `false`, если они должны заменить существующие данные.

- `int getIndex()`

  Получает индекс в модели для вставки или замены.

javax.swing.JTable.DropLocation 6

- `boolean isInsertRow()`
- `boolean isInsertColumn()`

  Возвращают логическое значение `true`, если данные должны быть вставлены перед строкой или столбцом.

- `int getRow()`
- `int getColumn()`

  Получают индекс строки или столбца в модели для вставки или замены, а если опускание произошло на пустом участке, то значение **-1**.

javax.swing.JTree.DropLocation 6

- `TreePath getPath()`
- `int getChildIndex()`

  Возвращают путь по дереву и индекс того дочернего элемента, который определяет место опускания наряду с режимом опускания целевого компонента, как описано ниже.

Режим опускания	*Операция редактирования в дереве*
`INSERT`	Вставить в качестве дочернего элемента по заданному пути перед дочерним элементом по указанному индексу
`ON` или `USE_SELECTION`	Заменить данные в пути (индекс дочернего элемента не используется)
`INSERT_OR_ON`	Если индекс дочернего элемента равен **-1**, поступить как в режиме `ON`, а иначе — как в режиме `INSERT`

javax.swing.text.JTextComponent.DropLocation 6

- `int getIndex()`

  Возвращает индекс, по которому должны быть вставлены данные.

Интегрирование с платформой

И в завершении этой главы рассмотрим несколько средств, внедренных в версии Java SE 6, чтобы теснее интегрировать прикладные программы на Java с конкретной платформой. Средство создания начальных экранов позволяет отображать начальный экран при запуске виртуальной машины Java. А класс `java.awt.Desktop` обеспечивает возможность запуска собственных приложений платформы вроде используемого по умолчанию браузера или программы электронной почты в прикладных программах на Java. И, наконец, еще одно средство позволяет получать из прикладных программ на Java доступ к системной области и заполнять ее соответствующими пиктограммами таким же образом, как это делают многие собственные приложения платформы.

Начальные экраны

К числу наиболее распространенных претензий к прикладным программам на Java относится медленный их запуск на выполнение. Виртуальной машине Java всегда требуется какое-то время на загрузку всех требуемых классов, и особенно это касается Swing-программ, для которых ей приходится загружать немало кода из библиотек Swing и AWT. Пользователям не нравятся прикладные программы, при запуске которых им приходится ждать появления первого экрана, и поэтому они могут пытаться запускать подобные программы несколько раз, если им кажется, что сделать это с первой попытки не удалось. Удобным выходом из столь затруднительного положения является *начальный экран*, т.е. небольшое окно, быстро появляющееся в виде заставки и уведомляющее пользователя, что приложение успешно запущено.

Безусловно, окно с начальным экраном можно отобразить в самом начале работы метода `main()`. Но ведь выполнение этого метода начинается только после того, как загрузчик классов загрузит все зависимые классы, на что может уйти время. Вместо этого можно предписать виртуальной машине отображать нужное изображение начального экрана сразу же после запуска. Для указания такого изображения имеются два механизма. С одной стороны, можно указать параметр **`-splash`** в командной строке следующим образом:

```
java -splash:myimage.png MyApp
```

А с другой стороны, можно указать нужное изображение в манифесте JAR-файла, как показано ниже.

```
Main-Class: MyApp
SplashScreen-Image: myimage.gif
```

Это изображение будет отображаться сразу при запуске прикладной программы и автоматически исчезать при появлении первого окна AWT. В качестве начального экрана разрешается использовать любые изображения формата GIF, JPEG или PNG. Поддерживается также анимация (в формате GIF) и прозрачность (в форматах GIF и PNG).

Если ваша прикладная программа готова к выполнению, как только достигает метода `main()`, можете пропустить остальную часть этого раздела. Но многие прикладные программы на Java имеют модульную архитектуру, где небольшое ядро

загружает при запуске ряд подключаемых модулей. Характерными тому примерами служат ИСР Eclipse и NetBeans. В этом случае на начальном экране может отображаться ход загрузки подключаемых модулей.

Добиться такого результата можно двумя способами. Первый способ подразумевает рисование прямо на начальном экране, а второй — замену этого экрана фреймом без обрамления с аналогичным содержимым и рисование уже в этом фрейме. Оба способа демонстрируются в рассматриваемом здесь примере программы.

Для рисования прямо на начальном экране сначала нужно получить ссылку на этот экран и сведения о его графическом контексте и размерах, как показано ниже:

```
SplashScreen splash = SplashScreen.getSplashScreen();
Graphics2D g2 = splash.createGraphics();
Rectangle bounds = splash.getBounds();
```

Затем рисование выполняется обычным образом. По его завершении непременно вызывается метод `update()`, чтобы гарантировать обновление полученного рисунка. В рассматриваемом здесь примере программы рисуется простой индикатор выполнения, как показано ниже и на рис. 7.47, *слева*.

```
g.fillRect(x, y, width * percent / 100, height);
splash.update();
```

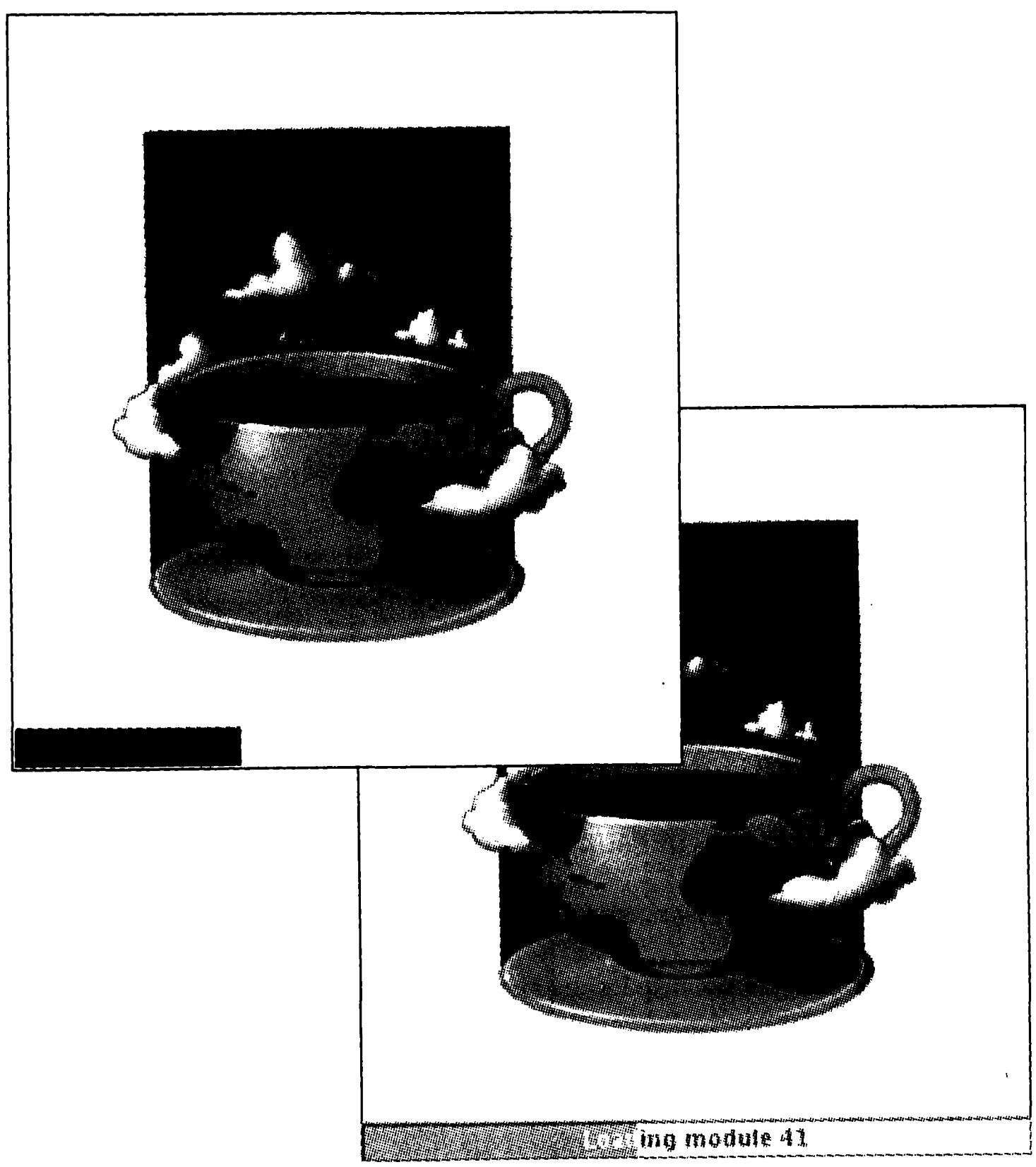


Рис. 7.47. Первоначальный экран и следующее за ним окно без обрамления

НА ЗАМЕТКУ! Начальный экран является одноэлементным объектом (так называемым *одиночкой*). Создать подобный объект самостоятельно нельзя. Если же начальный экран не указан ни в командной строке, ни в манифесте, то метод `getSplashScreen()` возвратит пустое значение `null`.

У рисования прямо на начальном экране имеется следующий существенный недостаток: вычислять позиции всех пикселей утомительно, а рисуемый индикатор выполнения обычно не согласуется с собственным индикатором используемой платформы. Поэтому в самом начале выполнения метода `main()` лучше заменить начальный экран последующим окном с теми же размерами и содержимым. Такое окно может содержать любые компоненты Swing.

Именно такой способ демонстрируется в рассматриваемом здесь примере программы из листинга 7.23. На рис. 7.47, *справа*, показан фрейм без обрамления с панелью, на которой рисуется начальный экран и находится компонент `JProgressBar`. Благодаря добавлению такого фрейма открывается полный доступ к прикладному интерфейсу API библиотеки Swing, что позволяет без особого труда вводить строки сообщений, не утруждая себя вычислением позиций пикселей. Следует также иметь в виду, что удалять начальный экран не нужно. Он постепенно исчезнет автоматически, как только станет появляться следующее за ним окно.

ВНИМАНИЕ! К сожалению, замена начального экрана следующим за ним окном сопровождается заметным мерцанием.

Листинг 7.23. Исходный код из файла `splashScreen/SplashScreenTest.java`

```
1   package splashScreen;
2
3   import java.awt.*;
4   import java.util.List;
5   import javax.swing.*;
6
7   /**
8    * В этой программе демонстрируется прикладной интерфейс API для
9    * отображения начального экрана при запуске прикладной программы
10   * @version 1.00 2007-09-21
11   * @author Cay Horstmann
12  */
13  public class SplashScreenTest
14  {
15    private static final int DEFAULT_WIDTH = 300;
16    private static final int DEFAULT_HEIGHT = 300;
17
18    private static SplashScreen splash;
19
20    private static void drawOnSplash(int percent)
21    {
22      Rectangle bounds = splash.getBounds();
23      Graphics2D g = splash.createGraphics();
24      int height = 20;
25      int x = 2;
26      int y = bounds.height - height - 2;
27      int width = bounds.width - 4;
```

```
28      Color brightPurple = new Color(76, 36, 121);
29      g.setColor(brightPurple);
30      g.fillRect(x, y, width * percent / 100, height);
31      splash.update();
32    }
33
34    /**
35     * Этот метод рисует на начальном экране
36    */
37    private static void init1()
38    {
39      splash = SplashScreen.getSplashScreen();
40      if (splash == null)
41      {
42        System.err.println("Did you specify a splash image with -
43           splash or in the manifest?");
44        System.exit(1);
45      }
46
47      try
48      {
49        for (int i = 0; i <= 100; i++)
50        {
51          drawOnSplash(i);
52          Thread.sleep(100); // сымитировать начало работы программы
53        }
54      }
55      catch (InterruptedException e)
56      {
57      }
58    }
59
60    /**
61      * Этот метод отображает фрейм с тем же самым изображением,
62      * что и на начальном экране
63    */
64    private static void init2()
65    {
66      final Image img = new ImageIcon(splash.getImageURL()).getImage();
67
68      final JFrame splashFrame = new JFrame();
69      splashFrame.setUndecorated(true);
70
71      final JPanel splashPanel = new JPanel()
72        {
73          public void paintComponent(Graphics g)
74          {
75            g.drawImage(img, 0, 0, null);
76          }
77        };
78
79      final JProgressBar progressBar = new JProgressBar();
80      progressBar.setStringPainted(true);
81      splashPanel.setLayout(new BorderLayout());
82      splashPanel.add(progressBar, BorderLayout.SOUTH);
83
84      splashFrame.add(splashPanel);
85      splashFrame.setBounds(splash.getBounds());
```

```
86        splashFrame.setVisible(true);
87
88        new SwingWorker<Void, Integer>()
89        {
90          protected Void doInBackground() throws Exception
91          {
92            try
93            {
94              for (int i = 0; i <= 100; i++)
95              {
96                publish(i);
97                Thread.sleep(100);
98              }
99            }
100           catch (InterruptedException e)
101           {
102           }
103           return null;
104       }
105
106       protected void process(List<Integer> chunks)
107       {
108         for (Integer chunk : chunks)
109         {
110           progressBar.setString("Loading module " + chunk);
111           progressBar.setValue(chunk);
112           splashPanel.repaint(); // перерисовать изображение, т.к.
113                                  // оно загружается асинхронно
114         }
115       }
116
117       protected void done()
118       {
119         splashFrame.setVisible(false);
120
121         JFrame frame = new JFrame();
122         frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
123         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
124         frame.setTitle("SplashScreenTest");
125           frame.setVisible(true);
126         }
127       }.execute();
128     }
129
130     public static void main(String args[])
131     {
132       init1();
133
134       EventQueue.invokeLater(new Runnable()
135         {
136           public void run()
137           {
138             init2();
139           }
140         });
141     }
142 }
```

java.awt.SplashScreen 6

- `static SplashScreen getSplashScreen()`

  Получает ссылку на начальный экран или пустое значение `null`, если таковой отсутствует.

- `URL getImageURL()`
- `void setImageURL(URL` *`imageURL`*`)`

  Получают или устанавливают URL изображения для начального экрана. Установка изображения приводит к обновлению начального экрана.

- `Rectangle getBounds()`

  Получает сведения о границах начального экрана.

- `Graphics2D createGraphics()`

  Получает графический контекст для рисования на начальном экране.

- `void update()`

  Обновляет отображаемый начальный экран.

- `void close()`

  Закрывает начальный экран. Как только появляется первое окно AWT, начальный экран закрывается автоматически.

Запуск настольных приложений

Класс `java.awt.Desktop` позволяет запускать такие используемые по умолчанию настольные приложения, как браузер и программа электронной почты, а также открывать, редактировать и распечатывать файлы с помощью тех приложений, которые зарегистрированы в системе как предназначенные для обработки файлов данного типа. Соответствующий прикладной интерфейс API организован очень просто. Сначала вызывается статический метод `isDesktopSupported()`. Если он возвращает логическое значение `true`, это означает, что на данной платформе поддерживается запуск настольных приложений, и тогда вызывается статический метод `getDesktop()` для получения экземпляра класса `Desktop`.

Не все настольные среды поддерживают полностью операции, предусмотренные в рассматриваемом здесь прикладном интерфейсе API. Например, в настольной среде Gnome ОС Linux допускается открывать файлы, а распечатывать их нельзя, потому что в ней не поддерживается использование "глаголов" при сопоставлении файлов. Выяснить, какие именно операции поддерживаются на данной платформе, можно, вызвав метод `isSupported()` и передав ему в качестве аргумента одно из значений в перечислении `Desktop.Action`. В рассматриваемом здесь примере программы для этой цели организуются проверки, аналогичные приведенной ниже.

```
if (desktop.isSupported(Desktop.Action.PRINT)) printButton.setEnabled(true);
```

Для открытия, редактирования или распечатки файла сначала выполняется проверка, поддерживается ли вообще такая операция, и только после этого вызывается метод `open()`, `edit()` или `print()` соответственно. Для запуска браузера передается URI. (Подробнее об URI см. в главе 3.) Можно также вызывать конструктор `URI` с символьной строкой, содержащей URL типа `http` или `https`. А для запуска

используемой по умолчанию программы электронной почты придется составить URI в определенном формате, как показано ниже.

```
mailto:получатель?запрос
```

Здесь `получатель` обозначает электронный адрес получателя, например `president@whitehouse.gov`, а `запрос` — разделенные знаком **&** пары `имя=значение` с закодированными знаком **%** значениями вроде `subject=dinner%20RSVP&bcc=putin%40kremvax.ru`. (Алгоритм кодирования знаком **%** действует, по существу, таким же образом, как и описанный в главе 3 алгоритм кодирования URL, только пробел в данном случае кодируется не как **+**, а как **%20**.) Данный формат описывается в документе RFC 2368 (`http://www.ietf.org/rfc/rfc2368.txt`).

В рассматриваемом здесь примере программы из листинга 7.24, предоставляется возможность открывать, редактировать или распечатывать любой выбираемый файл, отображать в окне браузера веб-страницу по любому указанному URL, а также запускать избранную программу электронной почты (рис. 7.48).

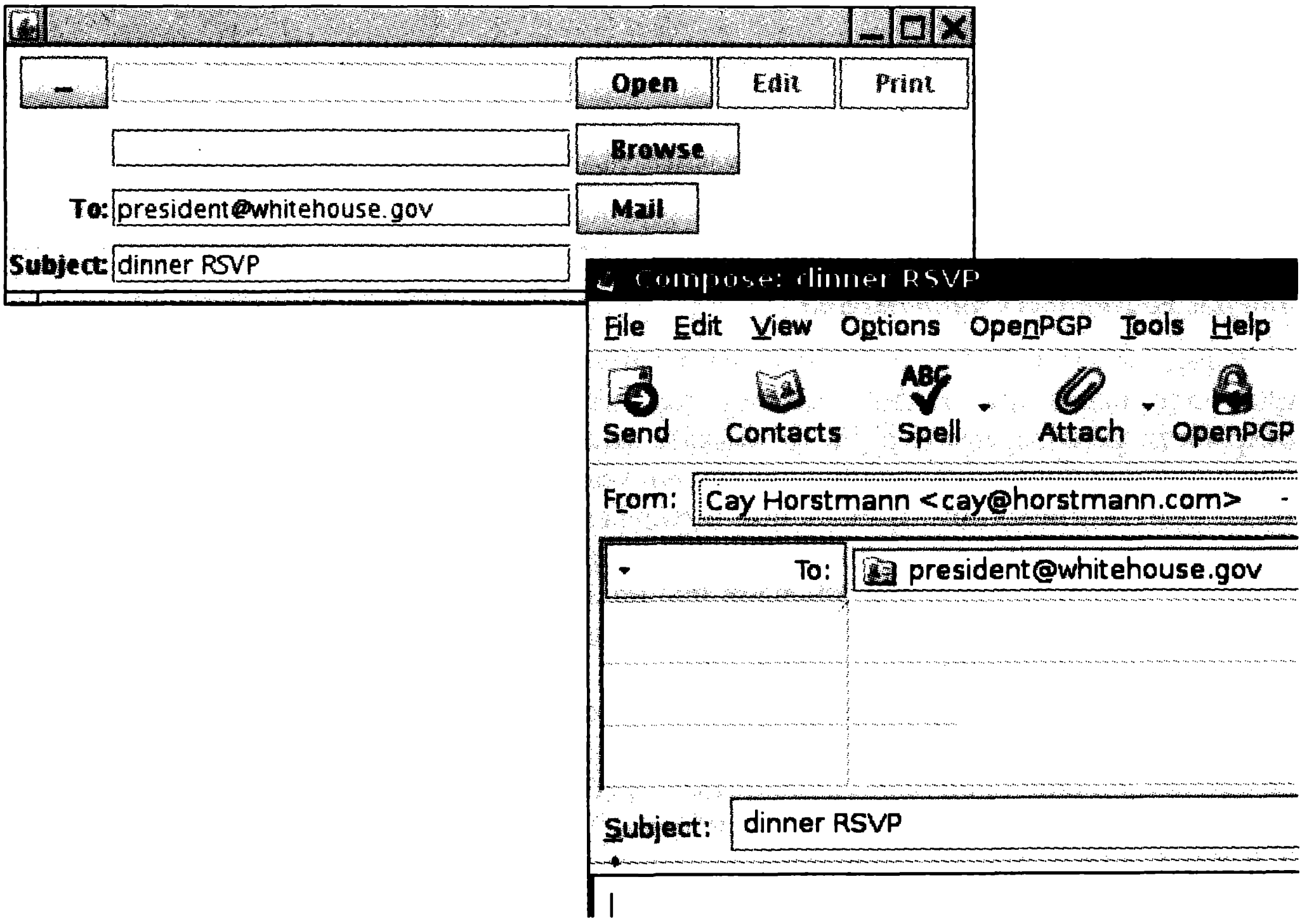


Рис. 7.48. Запуск настольного приложения

Листинг 7.24. Исходный код из файла `desktopApp/DesktopAppFrame.java`

```
1   package desktopApp;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.io.*;
6   import java.net.*;
```

```
7   import javax.swing.*;
8
9   class DesktopAppFrame extends JFrame
10  {
11    public DesktopAppFrame()
12    {
13      setLayout(new GridBagLayout());
14      final JFileChooser chooser = new JFileChooser();
15      JButton fileChooserButton = new JButton("...");
16      final JTextField fileField = new JTextField(20);
17      fileField.setEditable(false);
18      JButton openButton = new JButton("Open");
19      JButton editButton = new JButton("Edit");
20      JButton printButton = new JButton("Print");
21      final JTextField browseField = new JTextField();
22      JButton browseButton = new JButton("Browse");
23      final JTextField toField = new JTextField();
24      final JTextField subjectField = new JTextField();
25      JButton mailButton = new JButton("Mail");
26
27      openButton.setEnabled(false);
28      editButton.setEnabled(false);
29      printButton.setEnabled(false);
30      browseButton.setEnabled(false);
31      mailButton.setEnabled(false);
32      if (Desktop.isDesktopSupported())
33      {
34        Desktop desktop = Desktop.getDesktop();
35        if (desktop.isSupported(Desktop.Action.OPEN))
37          openButton.setEnabled(true);
38        if (desktop.isSupported(Desktop.Action.EDIT))
39          editButton.setEnabled(true);
40        if (desktop.isSupported(Desktop.Action.PRINT))
41          printButton.setEnabled(true);
42        if (desktop.isSupported(Desktop.Action.BROWSE))
43          browseButton.setEnabled(true);
44        if (desktop.isSupported(Desktop.Action.MAIL))
45           mailButton.setEnabled(true);
46      }
47
48      fileChooserButton.addActionListener(new ActionListener()
49        {
50          public void actionPerformed(ActionEvent e)
51          {
52            if (chooser.showOpenDialog(DesktopAppFrame.this) ==
53               JFileChooser.APPROVE_OPTION)
54            fileField.setText(chooser.getSelectedFile().getAbsolutePath());
55          }
56        });
57
58      openButton.addActionListener(new ActionListener()
59        {
60          public void actionPerformed(ActionEvent e)
61          {
62            try
63            {
64              Desktop.getDesktop().open(chooser.getSelectedFile());
65            }
```

```
66            catch (IOException ex)
67            {
68              ex.printStackTrace();
69            }
70          }
71        });
72
73      editButton.addActionListener(new ActionListener()
74        {
75          public void actionPerformed(ActionEvent e)
76          {
77            try
78            {
79              Desktop.getDesktop().edit(chooser.getSelectedFile());
80            }
81            catch (IOException ex)
82            {
83              ex.printStackTrace();
84            }
85          }
86        });
87      printButton.addActionListener(new ActionListener()
88        {
89          public void actionPerformed(ActionEvent e)
90          {
91            try
92            {
93              Desktop.getDesktop().print(chooser.getSelectedFile());
94            }
95            catch (IOException ex)
96            {
97              ex.printStackTrace();
98            }
99          }
100       });
101
102     browseButton.addActionListener(new ActionListener()
103       {
104         public void actionPerformed(ActionEvent e)
105         {
106           try
107           {
108             Desktop.getDesktop().browse(new URI(browseField.getText()));
109           }
110           catch (URISyntaxException ex)
111           {
112             ex.printStackTrace();
113           }
114           catch (IOException ex)
115           {
116             ex.printStackTrace();
117           }
118         }
119       });
120
121     mailButton.addActionListener(new ActionListener()
122       {
123         public void actionPerformed(ActionEvent e)
```

```
124         {
125           try
126           {
127             String subject = percentEncode(subjectField.getText());
128             URI uri = new URI("mailto:" + toField.getText()
129                 + "?subject=" + subject);
130
131             System.out.println(uri);
132             Desktop.getDesktop().mail(uri);
133           }
134           catch (URISyntaxException ex)
135           {
136             ex.printStackTrace();
137           }
138           catch (IOException ex)
139           {
140             ex.printStackTrace();
141           }
142         }
143       });
144
145     JPanel buttonPanel = new JPanel();
146     ((FlowLayout) buttonPanel.getLayout()).setHgap(2);
147     buttonPanel.add(openButton);
148     buttonPanel.add(editButton);
149     buttonPanel.add(printButton);
150
151     add(fileChooserButton, new GBC(0, 0)
152        .setAnchor(GBC.EAST).setInsets(2));
153     add(fileField, new GBC(1, 0).setFill(GBC.HORIZONTAL));
154     add(buttonPanel, new GBC(2, 0).setAnchor(GBC.WEST).setInsets(0));
155     add(browseField, new GBC(1, 1).setFill(GBC.HORIZONTAL));
156     add(browseButton, new GBC(2, 1).setAnchor(GBC.WEST).setInsets(2));
157     add(new JLabel("To:"), new GBC(0, 2).setAnchor(GBC.EAST)
158        .setInsets(5, 2, 5, 2));
159     add(toField, new GBC(1, 2).setFill(GBC.HORIZONTAL));
160     add(mailButton, new GBC(2, 2).setAnchor(GBC.WEST).setInsets(2));
161     add(new JLabel("Subject:"), new GBC(0, 3).setAnchor(GBC.EAST)
162        .setInsets(5, 2, 5, 2));
163     add(subjectField, new GBC(1, 3).setFill(GBC.HORIZONTAL));
164
165     pack();
166   }
167
168   private static String percentEncode(String s)
169   {
170     try
171     {
172       return URLEncoder.encode(s, "UTF-8").replaceAll("[+]", "%20");
173     }
174     catch (UnsupportedEncodingException ex)
175     {
176       return null; // кодировка UTF-8 поддерживается всегда
177     }
178   }
179 }
```

java.awt.Desktop 6

- `static boolean isDesktopSupported()`

  Возвращает логическое значение `true`, если запуск настольных приложений поддерживается на данной платформе.

- `static Desktop getDesktop()`

  Возвращает объект типа `Desktop` для операций запуска настольных приложений. Генерирует исключение типа `UnsupportedOperationException`, если операции запуск настольных приложений не поддерживаются на данной платформе.

- `boolean isSupported(Desktop.Action` *`action`*`)`

  Возвращает логическое значение `true`, если указанное действие поддерживается. А указать может одно из следующих действий: `OPEN`, `EDIT`, `PRINT`, `BROWSE` или `MAIL`.

- `void open(File` *`file`*`)`

  Запускает зарегистрированное приложение для открытия указанного файла.

- `void edit(File` *`file`*`)`

  Запускает зарегистрированное приложение для редактирования указанного файла.

- `void print(File` *`file`*`)`

  Распечатывает указанный файл.

- `void browse(URI` *`uri`*`)`

  Запускает используемый по умолчанию браузер с указанным URI.

- `void mail()`

- `void mail(URI` *`uri`*`)`

  Запускают используемую по умолчанию программу электронной почты. Во втором варианте допускается заполнение отдельных частей электронного сообщения.

Системная область

Во многих настольных средах имеется область для отображения пиктограмм тех программ, которые выполняются в фоновом режиме и время от времени уведомляют пользователей о тех или иных событиях. В Windows такая область называется *системной*, а отображаемые в ней пиктограммы — *системными*. Такая же терминология принята и в прикладном интерфейсе Java API. Типичным примером системной области является монитор, следящий за появлением обновленных версий программного обеспечения. При появлении обновлений монитор может изменить внешний вид отображаемой в системной области пиктограммы или отобразить рядом с ней соответствующее сообщение.

Откровенно говоря, системная область обычно перенасыщена пиктограммами, и поэтому добавление в ней еще одной пиктограммы, как правило, не вызывает у пользователей особого энтузиазма. И рассматриваемая здесь в качестве примера программа, раздающая виртуальные "печенья-гадания", вряд ли станет исключением из этого правила.

Класс `java.awt.SystemTray` является универсальным (независящим от конкретной платформы) проводником в системную область. Как и в рассмотренном

выше классе `Desktop`, в нем сначала вызывается статический метод `isSupported()` для проверки, поддерживается ли системная область на локальной платформе Java. Если она поддерживается, то далее вызывается метод `getSystemTray()` для получения одноэлементного объекта типа `SystemTray`.

Самым важным в классе `SystemTray` является метод `add()`, позволяющий добавлять экземпляр класса `TrayIcon`. У любой системной пиктограммы имеются три основных свойства.

- Изображение пиктограммы.
- Всплывающая подсказка, которая появляется при наведении курсора мыши на пиктограмму.
- Всплывающее меню, которое появляется после щелчка правой кнопкой мыши на пиктограмме.

Всплывающее меню вводится в качестве экземпляра класса `PopupMenu` из библиотеки AWT, представляющего платформенно-ориентированное всплывающее меню, но не меню из библиотеки Swing. А пункты вводятся в меню как экземпляры класса `MenuItem` из библиотеки AWT. У каждого из пунктов меню, как и у их аналогов из библиотеки Swing, имеется свой приемник действий.

И, наконец, рядом с пиктограммой в системной области можно также выводить различные уведомления для пользователей (рис. 7.49). Для этого достаточно вызвать метод `displayMessage()` из класса `TrayIcon`, указав заголовок, текст и тип сообщения:

```
trayIcon.displayMessage("Your Fortune", fortunes.get(index),
    TrayIcon.MessageType.INFO);
```

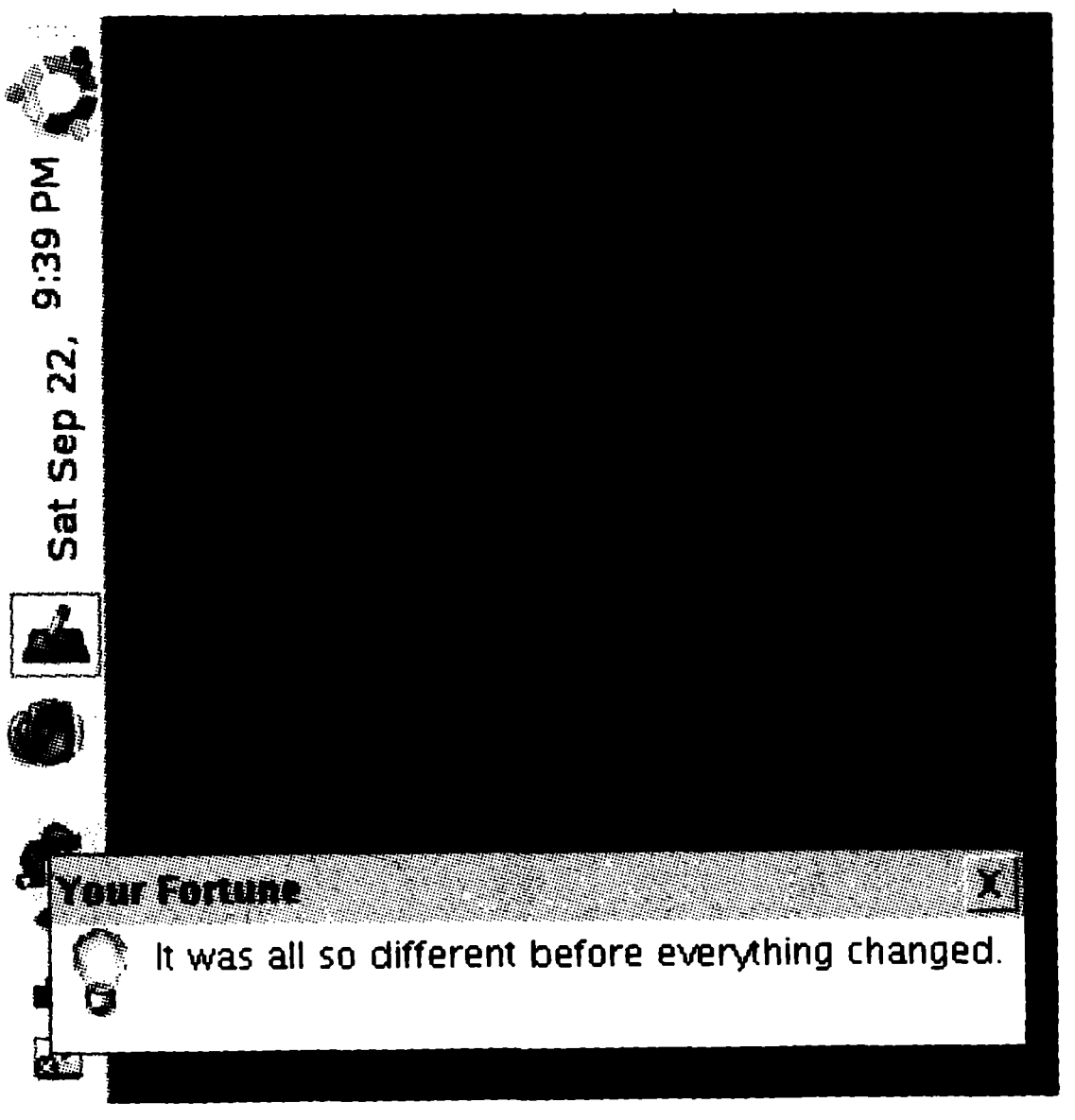


Рис. 7.49. Уведомление от пиктограммы в системной области

В листинге 7.25 приведен исходный код примера прикладной программы, демонстрирующей размещение пиктограммы "печенья-гадания" в системной области. Эта программа считывает файл "печений-гаданий" (из хорошо известной в UNIX программы `fortune`), где каждое гадание оканчивается строкой со знаком `%`, а также отображает новое сообщение каждые десять секунд. Для выхода из данной прикладной программы ее пиктограмма в системной области снабжена всплывающим меню с командой завершения работы. Вот если бы все пиктограммы, отображаемые в системной области, были настолько продуманы!

Листинг 7.25. Исходный код из файла `systemTray/SystemTrayTest.java`

```
1   package systemTray;
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 javax.swing.*;
9   import javax.swing.Timer;
10
11  /**
12   * В этой программе демонстрируется прикладной интерфейс API
13   * для отображения пиктограмм приложений в системной области
14   * @version 1.01 2012-01-26
15   * @author Cay Horstmann
16  */
17  public class SystemTrayTest
18  {
19    public static void main(String[] args)
20    {
21      SystemTrayApp app = new SystemTrayApp();
22      app.init();
23    }
24  }
25
26  class SystemTrayApp
27  {
28    public void init()
29    {
30      final TrayIcon trayIcon;
31
32      if (!SystemTray.isSupported())
33      {
34        System.err.println("System tray is not supported.");
35        return;
36      }
37
38      SystemTray tray = SystemTray.getSystemTray();
39      Image image = new ImageIcon(getClass().getResource("cookie.png"))
40         .getImage();
41
42      PopupMenu popup = new PopupMenu();
43      MenuItem exitItem = new MenuItem("Exit");
44      exitItem.addActionListener(new ActionListener()
45        {
46          public void actionPerformed(ActionEvent e)
```

```
47          {
48            System.exit(0);
49          }
50        });
51      popup.add(exitItem);
52
53      trayIcon = new TrayIcon(image, "Your Fortune", popup);
54
55      trayIcon.setImageAutoSize(true);
56      trayIcon.addActionListener(new ActionListener()
57        {
58          public void actionPerformed(ActionEvent e)
59          {
60            trayIcon.displayMessage("How do I turn this off?",
61               "Right-click on the fortune cookie and select Exit.",
62                TrayIcon.MessageType.INFO);
63          }
64        });
65
66      try
67      {
68        tray.add(trayIcon);
69      }
70      catch (AWTException e)
71      {
72        System.err.println("TrayIcon could not be added.");
73        return;
74      }
75
76      final List<String> fortunes = readFortunes();
77      Timer timer = new Timer(10000, new ActionListener()
78        {
79          public void actionPerformed(ActionEvent e)
80          {
81            int index = (int) (fortunes.size() * Math.random());
82            trayIcon.displayMessage("Your Fortune", fortunes.get(index),
83               TrayIcon.MessageType.INFO);
84          }
85        });
86      timer.start();
87    }
88    private List<String> readFortunes()
89    {
90      List<String> fortunes = new ArrayList<>();
91      try (InputStream inStream =
92         getClass().getResourceAsStream("fortunes"))
93      {
94        Scanner in = new Scanner(inStream);
95        StringBuilder fortune = new StringBuilder();
96        while (in.hasNextLine())
97        {
98          String line = in.nextLine();
99          if (line.equals("%"))
100         {
101           fortunes.add(fortune.toString());
102           fortune = new StringBuilder();
103         }
104         else
105         {
```

```
106             fortune.append(line);
107             fortune.append(' ');
108           }
109         }
110       }
111       catch (IOException ex)
112       {
113         ex.printStackTrace();
114       }
115       return fortunes;
116     }
117 }
```

java.awt.SystemTray 6

- `static boolean isSupported()`

  Возвращает логическое значение `true`, если на данной платформе поддерживается доступ к системной области.

- `static SystemTray getSystemTray()`

  Возвращает объект типа `SystemTray` для получения доступа к системной области. Генерирует исключение типа `UnsupportedOperationException`, если на данной платформе не поддерживается доступ к системной области.

- `Dimension getTrayIconSize()`

  Получает размеры пиктограммы в системной области.

- `void add(TrayIcon` *`trayIcon`*`)`
- `void remove(TrayIcon` *`trayIcon`*`)`

  Вводит или удаляет пиктограмму из системной области.

java.awt.TrayIcon 6

- `TrayIcon(Image` *`image`*`)`
- `TrayIcon(Image` *`image`*`, String` *`tooltip`*`)`
- `TrayIcon(Image` *`image`*`, String` *`tooltip`*`, PopupMenu` *`popupMenu`*`)`

  Создают пиктограмму для отображения в системной области с указанным изображением, всплывающей подсказкой и всплывающим меню.

- `Image getImage()`
- `void setImage(Image` *`image`*`)`
- `String getTooltip()`
- `void setTooltip(String` *`tooltip`*`)`
- `PopupMenu getPopupMenu()`
- `void setPopupMenu(PopupMenu` *`popupMenu`*`)`

  Получают или устанавливают изображение, всплывающую подсказку или всплывающее меню для системной пиктограммы.

- `boolean isImageAutoSize()`
- `void setImageAutoSize(boolean` *`autosize`*`)`

  Получают или устанавливают свойство `imageAutoSize`. Если это свойство устанавливается, то изображение масштабируется таким образом, чтобы уместиться в области пиктограммы всплывающей подсказки. Если же данное свойство не устанавливается (по умолчанию), то изображение обрезается (если оно слишком велико) или располагается по центру (если оно слишком мало).

- `void displayMessage(String` *`caption`*`, String` *`text`*`, TrayIcon.MessageType` *`messageType`*`)`

  Отображает уведомление рядом с системной пиктограммой. Это уведомление может быть одного из следующих видов: `INFO`, `WARNING`, `ERROR` или `NONE`.

- `public void addActionListener(ActionListener` *`listener`*`)`
- `public void removeActionListener(ActionListener` *`listener`*`)`

  Вводят или удаляют приемник действий, когда вызываемый приемник зависит от конкретной платформы. Обычно вызываются при выполнении одиночного щелчка на уведомлении или двойного щелчка на системной пиктограмме.

Вот и подошла к концу эта длинная глава, посвященная расширенным средствам AWT. В следующей главе речь пойдет о спецификации JavaBeans и ее применении для построителей ГПИ.

ГЛАВА 8

Компоненты JavaBeans

В этой главе...

- Назначение компонентов JavaBeans
- Создание компонентов JavaBeans
- Применение компонентов JavaBeans для разработки приложений
- Шаблоны именования свойств и событий в компонентах JavaBeans
- Типы свойств компонентов JavaBeans
- Классы типа `BeanInfo`
- Редакторы свойств
- Настройщики
- Сохраняемость компонентов JavaBeans

Официально компонент в спецификации JavaBeans определяется следующим образом: "Компонент JavaBeans — это многократно используемый элемент программного обеспечения, который определяется в спецификации JavaBeans компании Sun Microsystems и которым можно манипулировать в визуальном режиме работы инструментов построения ГПИ".

После реализации компонента JavaBeans одним разработчиком другие разработчики могут сразу же воспользоваться им в интегрированной среде разработки (ИСР) вроде NetBeans. Вместо того чтобы тратить силы на написание сложного кода, разработчики могут просто перетащить компонент JavaBeans, созданный другим разработчиком, на форму ГПИ и настроить его нужным образом с помощью диалоговых окон. В этой главе поясняется, как реализовать компоненты JavaBeans, чтобы ими было легко пользоваться другим разработчикам.

НА ЗАМЕТКУ! Прежде чем перейти к изложению материала этой главы, следует прояснить следующий момент во избежание всяких недоразумений: рассматриваемые здесь компоненты JavaBeans не имеют практически ничего общего с компонентами Enterprise JavaBeans (EJB). Последние представляют собой компоненты, размещаемые на стороне сервера, и предусматривают поддержку транзакций, сохраняемости, тиражирования и безопасности. Формально с ними можно работать в инструментах построения ГПИ и ИСР, но технология Enterprise JavaBeans более сложна, чем предлагаемая в версии Standard Edition технология JavaBeans.

Сказанное выше совсем не означает, что применение стандартных компонентов JavaBeans ограничивается лишь клиентскими программами. Веб-технологии вроде JavaServer Faces (JSF) и JavaServer Pages (JSP) в немалой степени опираются на модель компонентов JavaBeans.

Назначение компонентов JavaBeans

Тем, у кого имеется опыт программирования на Visual Basic, не нужно объяснять, почему так важны компоненты JavaBeans. Но тем, кто привык программировать в средах, где все нужно делать самому, нередко оказывается трудно поверить в то, что язык Visual Basic является одним из наиболее успешных примеров технологии повторного использования объектов. Поэтому для тех, кому никогда не доводилось программировать на Visual Basic, ниже перечисляются вкратце основные этапы разработки простой прикладной программы на этом языке.

1. Создание пользовательского интерфейса перетаскиванием мышью компонентов (называемых в Visual Basic *элементами управления*) в окно формы.
2. Установка с помощью *инспекторов свойств* значений отдельных свойств этих компонентов, например высоты, цвета или поведения.
3. Перечисление с помощью тех же самых инспекторов свойств событий, на которые эти компоненты могут реагировать. Для привязки к компонентам одних событий используются соответствующие диалоговые окна, а для привязки других событий — пишутся короткие фрагменты кода, обрабатывающего события.

В главе 2 первого тома был представлен пример программы для показа изображений во фрейме. Исходный код этой программы занял больше страницы. Чтобы написать программу с такими же функциональными возможностями на Visual Basic, достаточно было бы выполнить следующие действия.

1. Перетащить в окно разрабатываемого проекта два элемента управления: `Image` — для отображения графики и `CommonDialog` — для отображения окна выбора файлов.
2. Установить свойства `Filter` элемента управления `CommonDialog` таким образом, чтобы отображались только те файлы, которые элемент управления `Image` может обрабатывать (рис. 8.1).
3. Написать на Visual Basic следующие четыре строки кода, активизируемого при первом запуске данного проекта:

```
Private Sub Form_Load()
  CommonDialog1.ShowOpen
  Image1.Picture = LoadPicture(CommonDialog1.FileName)
End Sub
```

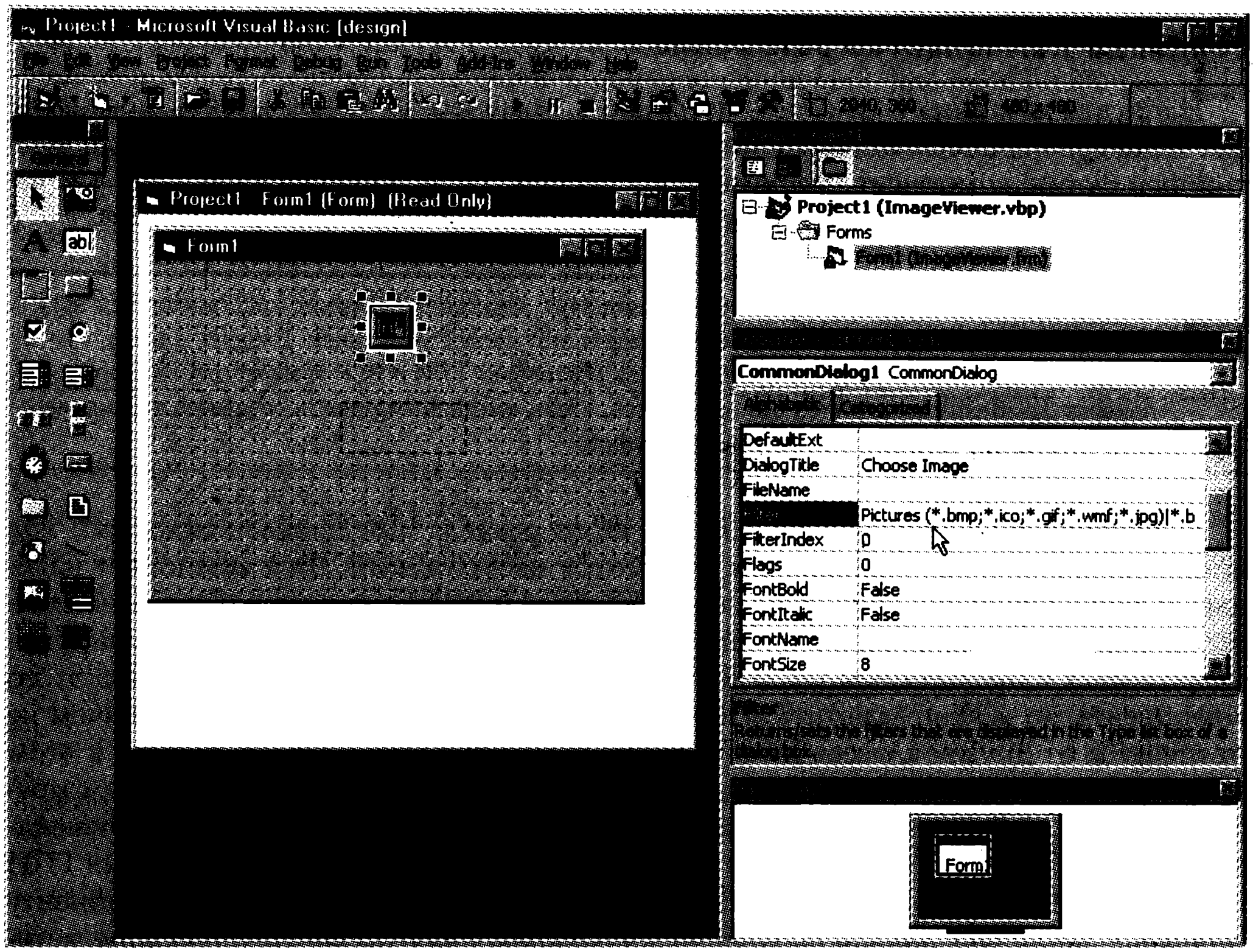


Рис. 8.1. Вид диалогового окна Properties (Свойства) в среде Visual Basic для разработки прикладной программы, показывающей изображения

4. Этот код отображает на экране диалоговое окно выбора только тех файлов, которые имеют подходящее расширение благодаря соответствующей установке свойства `Filter`. Как только пользователь выберет нужный ему файл изображения, код поручает элементу управления `Image` загрузить и показать выбранное изображение.

Вот, собственно, и все! Размещение элементов управления `CommonDialog` и `Image` в окне разрабатываемого проекта и ввод соответствующих операторов позволяют добиться практически такого же результата, как и целая страница кода, написанного на Java. Очевидно, что научиться перетаскивать готовые компоненты и устанавливать их свойства намного легче, чем научиться писать целые страницы кода.

Но сказанное выше совсем не означает, что Visual Basic подходит для решения любых задач. Безусловно, этот язык программирования годится для решения лишь определенного рода задач: создания приложений в Windows с развитым пользовательским интерфейсом. Технология JavaBeans для того и была разработана, чтобы платформа Java могла составить конкуренцию Visual Basic в данной области программирования. Эта технология дает производителям программного обеспечения возможность создавать среды разработки в стиле Visual Basic, позволяющие строить пользовательские интерфейсы, затрачивая минимум труда на программирование.

Создание компонентов JavaBeans

Написать компонент JavaBeans нетрудно с технической точки зрения, поскольку для этого требуется освоить всего лишь несколько новых классов и интерфейсов. В простейшей форме компонент JavaBeans представляет собой обычный класс Java, в котором соблюдается ряд довольно строгих условных обозначений имен его методов.

НА ЗАМЕТКУ! Некоторые требуют, чтобы компонент JavaBeans содержал конструктор без аргументов. Авторы спецификации JavaBeans обходят этот вопрос молчанием. Следует, однако, иметь в виду, что многие инструментальные средства построения ГПИ и ИСР требуют для каждого компонента JavaBeans конструктор без аргументов, чтобы иметь возможность создавать экземпляры соответствующих классов, не задавая параметров.

В листинге 8.1, приведенном в конце этого раздела, представлен исходный код компонента `ImageViewerBean` для просмотра изображений. Этот компонент наделяется при построении ГПИ прикладных программ на Java такими же функциональными возможностями, как и элемент управления изображениями в Visual Basic, упоминавшийся в предыдущем разделе. Если внимательно изучить исходный код этого компонента, то можно заметить, что класс `ImageViewerBean` практически ничем не отличается от других классов Java. В частности, имена всех методов доступа в этом классе начинаются с префикса **`get`**, а имена всех модифицирующих методов — с префикса **`set`**. Как будет показано ниже, именно эти условные обозначения применяются для обращения со *свойствами* в инструментальных средствах построения ГПИ и ИСР. Например, свойство `fileName` относится к упомянутому выше компоненту JavaBeans, поскольку для его получения и установки имеются соответствующие методы типа `get/set`.

Не следует, однако, путать свойство с полем экземпляра. Так, значение свойства `fileName` вычисляется из поля экземпляра `file`. Свойства принципиально находятся на более высоком уровне иерархии, чем поля экземпляра: они являются языковыми средствами интерфейса, тогда как поля экземпляра относятся к реализации этого интерфейса в классе.

При рассмотрении примеров, приведенных в этой главе, следует иметь в виду, что написать реальные компоненты JavaBeans намного сложнее по двум причинам.

1. Компоненты JavaBeans должны быть доступны для применения даже не очень опытными разработчиками. Поэтому им нужно предоставить доступ к функциональным возможностям компонента JavaBeans с помощью визуальных средств разработки без дополнительного программирования.
2. Компоненты JavaBeans должны быть рассчитаны на применение *в самых разных контекстах*. В них должны быть предусмотрены средства для настройки как поведения, так и внешнего вида. А это предполагает наличие большого количества свойств, доступных для установки.

Характерным примером компонента JavaBeans с богатыми функциональными возможностями является `CalendarBean`, созданный Каем Тёдтером (Kai Töedter) и показанный на рис. 8.2. Этот компонент и его исходный код свободно доступны по адресу http://www.toedter.com/en/jcalendar/. Он предоставляет пользователям удобные средства для выбора даты из отображаемого календаря. Но исходный код этого компонента довольной сложный, и поэтому вряд ли стоит писать его сызнова.

Работая с технологией JavaBeans, можно воспользоваться готовыми результатами трудов других программистов, перетаскивая компоненты в инструментальные средства построения ГПИ и ИСР.

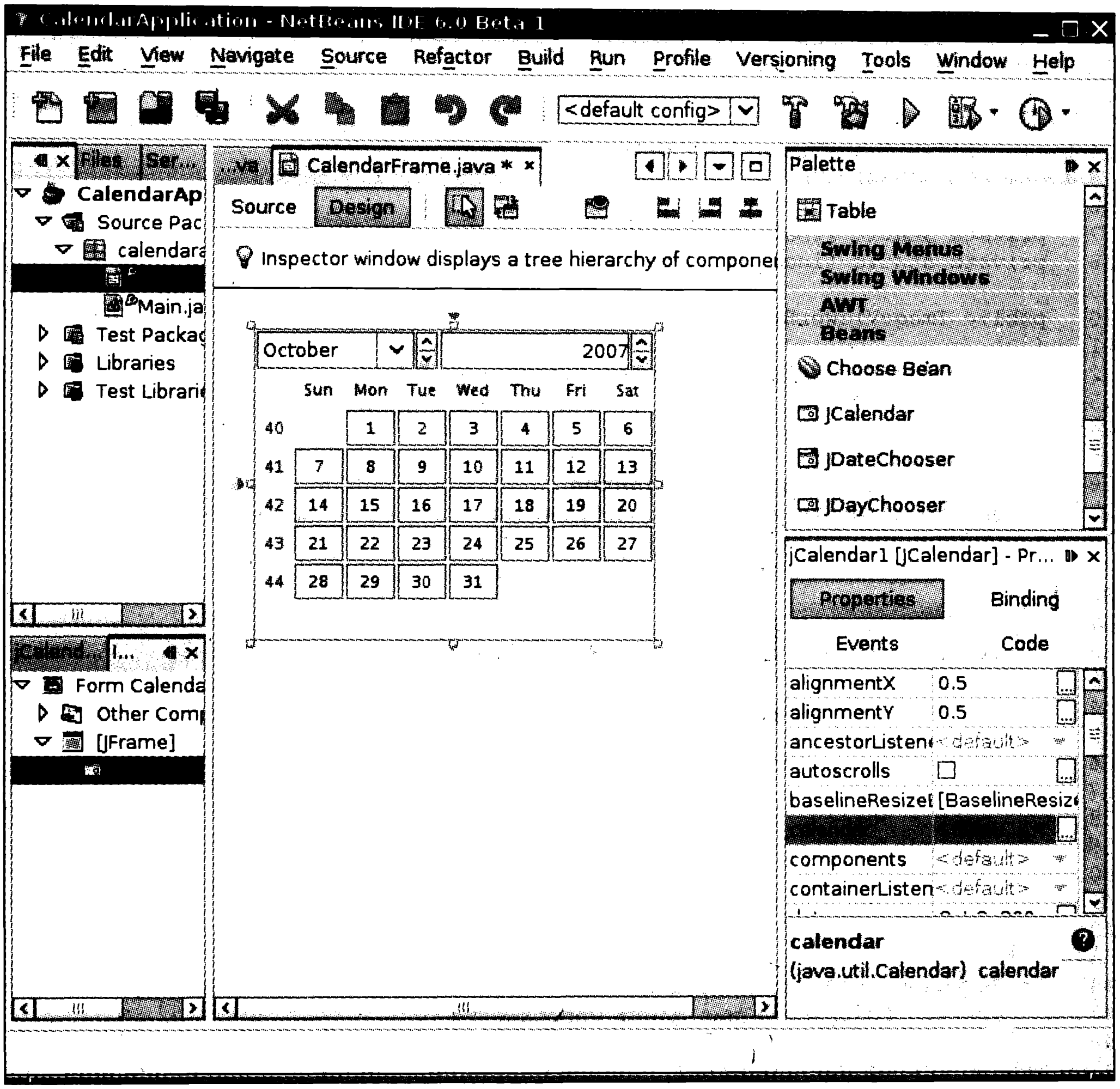


Рис. 8.2. Календарный компонент JavaBeans

Правда, вам придется усвоить лишь немногие понятия, чтобы научиться писать собственные компоненты JavaBeans с богатыми функциональными возможностями. Примеры таких компонентов приводятся в этой главе. И хотя они нетривиальны, тем не менее достаточно просты, чтобы продемонстрировать нужные понятия.

Листинг 8.1. Исходный код из файла `imageViewer/ImageViewerBean.java`

```
1  package imageViewer;
2
3  import java.awt.*;
4  import java.io.*;
```

```
5  import java.nio.file.*;
6  import javax.imageio.*;
7  import javax.swing.*;
8
9  /**
10  * Компонент JavaBeans для просмотра изображений
11  * @version 1.22 2012-06-10
12  * @author Cay Horstmann
13 */
14 public class ImageViewerBean extends JLabel
15 {
16   private Path path = null;
17   private static final int XPREFSIZE = 200;
18   private static final int YPREFSIZE = 200;
19
20   public ImageViewerBean()
21   {
22     setBorder(BorderFactory.createEtchedBorder());
23   }
24
25   /**
26    * Устанавливает свойство fileName
27    * @param fileName Имя файла изображения
28   */
29   public void setFileName(String fileName)
30   {
31     path = Paths.get(fileName);
32     try (InputStream in = Files.newInputStream(path))
33     {
34       setIcon(new ImageIcon(ImageIO.read(in)));
35     }
36     catch (IOException e)
37     {
38       path = null;
39       setIcon(null);
40     }
41   }
42
43   /**
44    * Получает свойство fileName
45    * @return Возвращает имя файла изображения
46   */
47   public String getFileName()
48   {
49     if (path == null) return "";
50     else return path.toString();
51   }
52
53   public Dimension getPreferredSize()
54   {
55     return new Dimension(XPREFSIZE, YPREFSIZE);
56   }
57 }
```

Применение компонентов JavaBeans для разработки приложений

Прежде чем углубляться в технические подробности создания компонентов JavaBeans, рассмотрим, каким образом эти компоненты можно применять и тестировать. Например, компонент `ImageViewerBean` вполне пригоден для использования, но вне пределов ИСР он утрачивает все свои особые возможности.

В каждой ИСР применяются свои методики для облегчения труда разработчиков. Здесь и далее рассматривается лишь одна ИСР — NetBeans, доступная для загрузки по адресу `http://netbeans.org`.

В примере, рассматриваемом в этом разделе, демонстрируется применение двух компонентов JavaBeans: `ImageViewerBean` и `FileNameBean`. Исходный код компонента `ImageViewerBean` приводился и пояснялся в предыдущем разделе, тогда как исходный код компонента `FileNameBean` будет анализироваться далее в этой главе. А до тех пор достаточно сказать, что если щелкнуть на кнопке с меткой многоточия (...), то откроется окно, в котором можно выбрать файл.

Упаковка компонентов JavaBeans в JAR-файлы

Чтобы сделать компонент JavaBeans пригодным для использования в ИСР, все относящиеся к нему файлы классов необходимо упаковать в архивный JAR-файл. В отличие от архивного JAR-файла аплета архивный JAR-файл компонента JavaBeans нуждается в файле манифеста, описывающего те файлы классов, которые представляют в архиве компоненты JavaBeans и должны быть включены в состав панели инструментов ИСР. В качестве примера ниже показано содержимое файла манифеста `ImageViewerBean.mf` для компонента `ImageViewerBean`. Обратите внимание на наличие пустой строки между описанием версии манифеста и названием компонента JavaBeans.

```
Manifest-Version: 1.0

Name: imageViewer/ImageViewerBean.class
Java-Bean: True
```

НА ЗАМЕТКУ! В некоторых ИСР возникают трудности при загрузке компонентов JavaBeans из стандартного пакета, выбираемого по умолчанию. Поэтому разрабатываемые компоненты следует всегда упаковывать в отдельные пакеты.

Если компонент JavaBeans содержит несколько файлов классов, то в файле манифеста следует указать только файлы классов для тех компонентов JavaBeans, которые требуется ввести на панели инструментов ИСР. Например, компоненты `ImageViewerBean` и `FileNameBean` можно упаковать в один JAR-файл и использовать для их описания следующий манифест:

```
Manifest-Version: 1.0

Name: imageViewer/ImageViewerBean.class
Java-Bean: True

Name: filePicker/FilePickerBean.class
Java-Bean: True
```

ВНИМАНИЕ! Некоторые ИСР предъявляют чрезвычайно высокие требования к формату манифеста. В частности, в конце строк не допускаются пробелы, после определения версии и между записями компонентов JavaBeans должна располагаться пустая строка, а за последней строкой должна следовать еще одна пустая строка.

Для того чтобы создать архивный JAR-файл, выполните следующие действия.

1. Отредактируйте файл манифеста.
2. Соберите все необходимые файлы классов в одном каталоге.
3. Запустите на выполнение утилиту архивирования **jar**, используя следующий синтаксис:

```
jar ctm jar-файл файл_манифеста файлы_классов
```

4. Ниже приведен характерный пример команды для создания архива.

```
jar cvfm ImageViewerBean.jar ImageViewerBean.mf imageViewer/*.class
```

В JAR-файлы можно также включать другие объекты, например пиктограммы. Подробнее о пиктограммах компонентов JavaBeans речь пойдет далее в этой главе.

НА ЗАМЕТКУ! Убедитесь в том, что в архивный JAR-файл упакованы все необходимые файлы, включая файлы внутренних классов, например `FileNameBean$1.class`.

ИСР обычно содержит механизмы для включения новых компонентов JavaBeans, как правило, загружаемых из архивных JAR-файлов. Ниже перечислены те действия, которые нужно выполнить для импорта новых компонентов в ИСР NetBeans версии 7. Скомпилируйте сначала классы `ImageViewerBean` и `FileNameBean` и затем упакуйте их в архивный JAR-файл. Далее запустите ИСР NetBeans и выполните следующие действия.

1. Выберите из меню команду Tools⇨Palette⇨Swing/AWT Components (Сервис⇨Палитра⇨Компоненты Swing/AWT).
2. Щелкните на кнопке Add from JAR (Добавить из JAR-файла).
3. Перейдите в открывшемся диалоговом окне к каталогу `ImageViewerBean` и выберите файл `ImageViewerBean.jar`.
4. Откроется диалоговое окно со списком всех компонентов JavaBeans из выбранного JAR-файла. Выберите компонент `ImageViewerBean`.
5. Далее вам будет предложено указать, в какой именно палитре следует разместить компоненты JavaBeans. Выберите панель Beans. (Имеются и другие панели, например, для компонентов Swing, компонентов AWT и т.д.).
6. Обратите внимание на палитру Beans. Теперь она содержит пиктограмму, представляющую новый компонент JavaBeans. Но в данном случае применяется пиктограмма по умолчанию. О том, как включать пиктограммы в состав компонентов JavaBeans, речь пойдет далее в этой главе.

Повторите эти действия, чтобы импортировать компонент `FileNameBean` в ИСР NetBeans. Теперь из полученных компонентов JavaBeans можно составить приложение.

Составление приложения из компонентов JavaBeans в ИСР

Разработка на основе компонентов предполагает составление приложений из готовых компонентов при минимальных затратах труда на программирование. В этом разделе в качестве примера рассматривается составление приложения из компонентов `ImageViewerBean` и `FileNameBean`. Прежде всего перейдите в ИСР NetBeans 7 и выберите команду меню File⇨New Project (Файл⇨Новый проект). Как только на экране появится диалоговое окно New Project, выберите сначала категорию `Java`, а затем проект `Java Application` (Приложение Java), как показано на рис. 8.3.

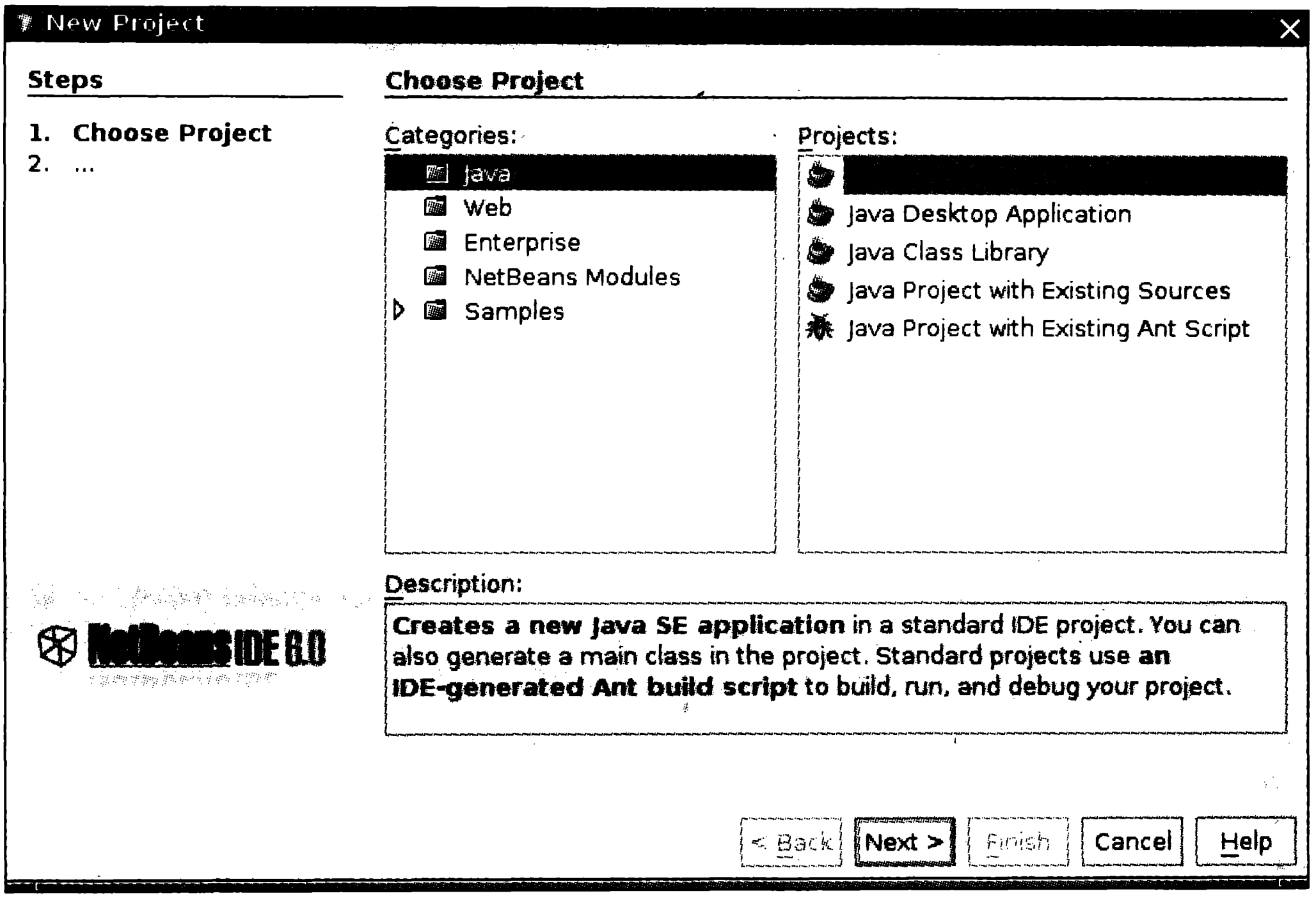


Рис. 8.3. Создание нового проекта в ИСР NetBeans

Щелкните на кнопке Next (Далее). На следующем экране укажите требуемое имя для приложения (например, `ImageViewer`) и щелкните на кнопке Finish (Готово). После этого в левой части экрана появится панель просмотра проектов, а в средней его части — окно редактора исходного кода. Щелкните правой кнопкой мыши на имени проекта в левой части экрана, где находится панель просмотра проектов, и выберите из всплывающего контекстного меню команду New⇨JFrame Form (Создать⇨Форму `JFrame`), как показано на рис. 8.4.

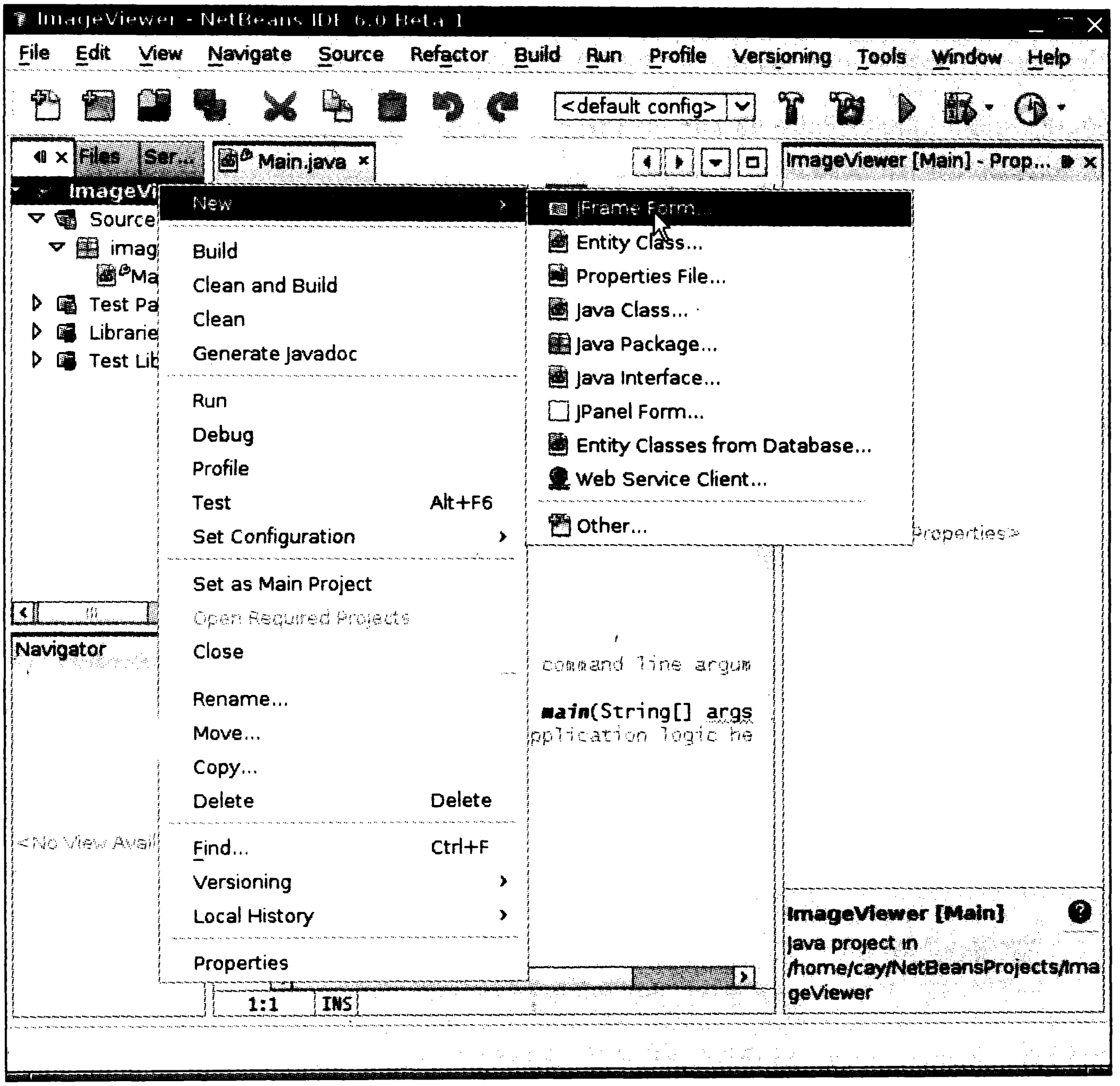


Рис. 8.4. Создание представления формы

Откроется очередное диалоговое окно. Введите в нем имя класса, реализующего фрейм (например, `ImageViewerFrame`), и щелкните на кнопке Finish. После этого появится редактор форм с пустым фреймом. Для того чтобы добавить компонент JavaBeans в форму, выберите его в палитре, находящейся справа от редактора форм, а затем щелкните на фрейме. На рис. 8.5 приведен результат добавления компонента `ImageViewerBean` во фрейм.

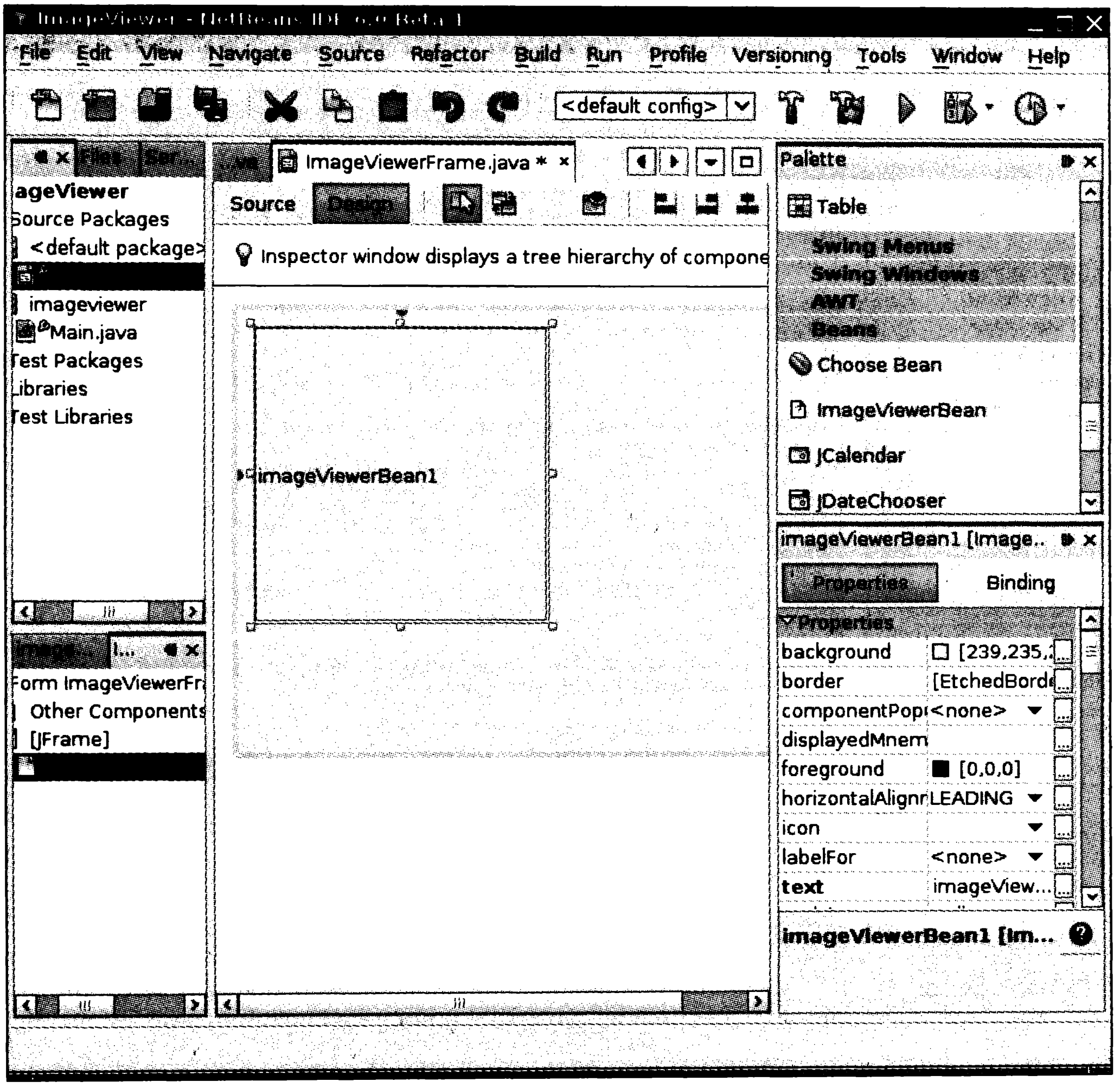


Рис. 8.5. Добавление компонента JavaBeans во фрейм

Если после этого вы заглянете в окно редактора кода, то увидите, что теперь в исходном коде содержатся операторы Java для ввода объектов JavaBeans во фрейм (рис. 8.6). Сам исходный код будет окружен предупреждениями о том, что его нельзя редактировать. Дело в том, что результаты редактирования могут исчезнуть при автоматическом обновлении кода после изменения формы в редакторе.

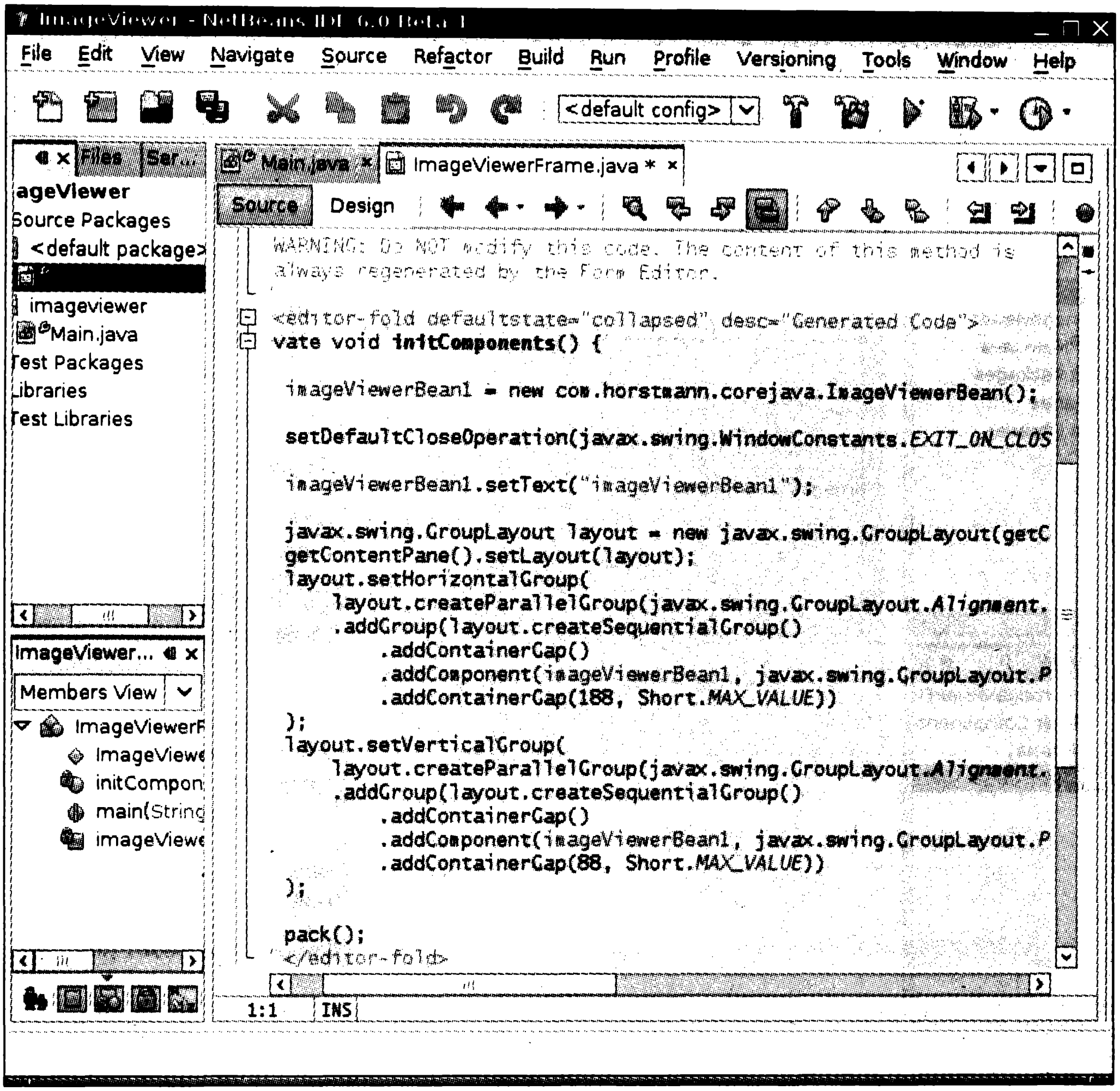


Рис. 8.6. Исходный код, требующийся для добавления компонента JavaBeans

НА ЗАМЕТКУ! ИСР не обязательно будет обновлять исходный код в процессе разработки приложения. Она может сгенерировать код по окончании редактирования, сериализировать компоненты JavaBeans, а возможно, и составить совершенно другое описание выполненных вами действий. Например, экспериментальный инструмент Bean Builder, доступный по адресу `http://java.net/projects/bean-builder`, позволяет создавать приложения с ГПИ вообще без написания исходного кода вручную.

Компоненты JavaBeans не навязывают ИСР свои принципы и методики реализации приложений. Они лишь сообщают о себе, чтобы инструментальные средства ИСР могли тем или иным способом воспользоваться полученными сведениями.

Вернитесь теперь к представлению проектирования на вкладке Design и щелкните на компоненте `ImageViewerBean` в форме. В правой части окна появится панель инспектора свойств со списком имен свойств данного компонента JavaBeans и их текущими значениями. Инспектор свойств является чрезвычайно важной частью инструментов разработки приложений на основе компонентов, поскольку благодаря установке свойств в процессе проектирования задается начальное состояние компонента.

Так, можно изменить свойство `text` используемой для компонента `ImageViewerBean` надписи, введя его новое значение непосредственно на панели инспектора свойств. Изменение свойства `text` выполняется очень легко: достаточно отредактировать строку в соответствующем текстовом поле. Попробуйте сделать это, изменив, например, текст надписи на `"Hello"`. Форма немедленно обновится, отразив внесенные вами изменения (рис. 8.7).

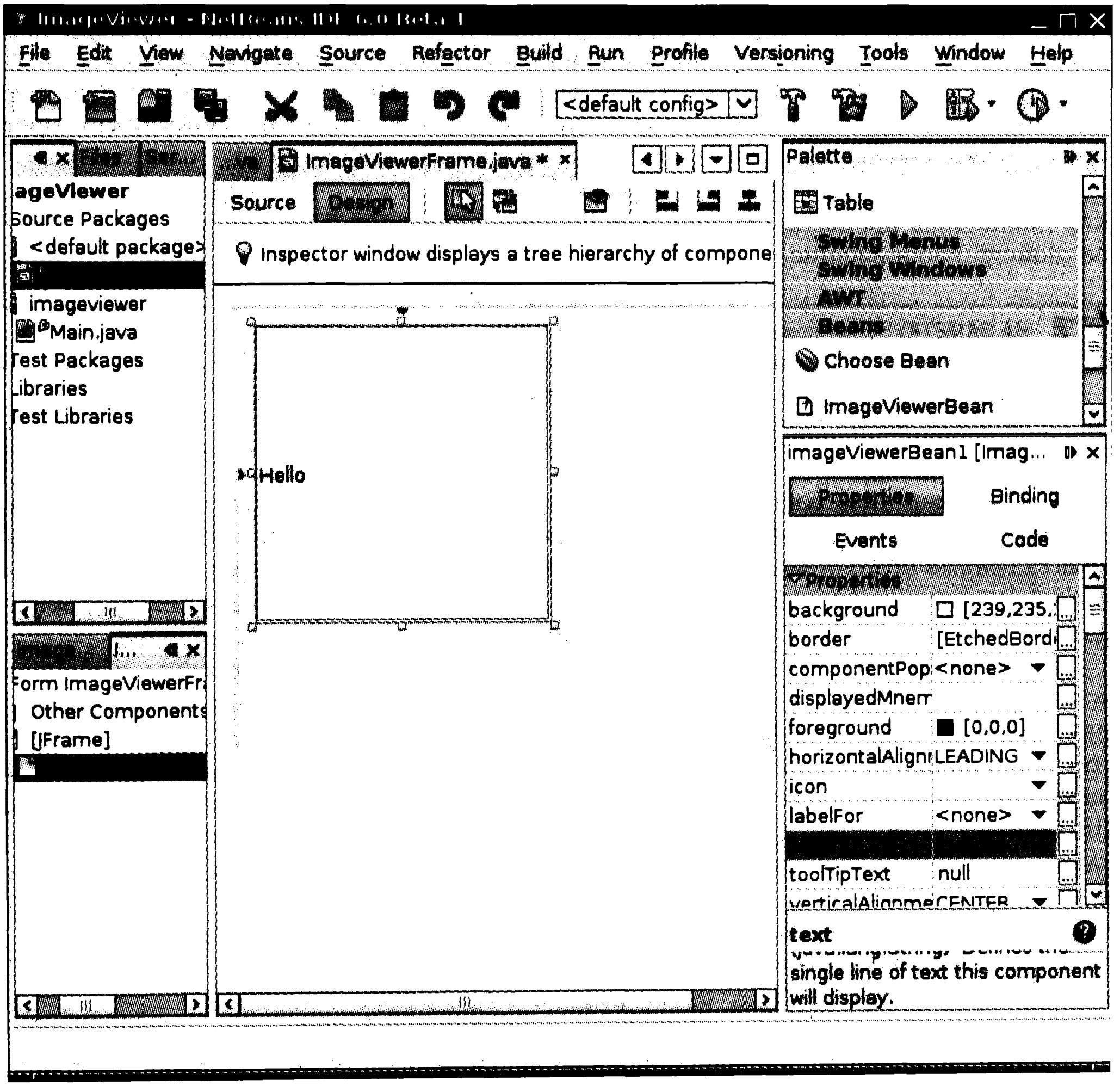


Рис. 8.7. Изменение значения свойства на панели инспектора свойств

НА ЗАМЕТКУ! При изменении значения свойства ИСР NetBeans сразу же обновляет исходный код соответствующим образом. Так, если ввести в поле `text` строку `"Hello"`, в теле метода `initComponents()` будет тотчас введен оператор `imageViewerBean.setText("Hello");`. Как упоминалось ранее, в других ИСР могут применяться иные методики для записи значений устанавливаемых свойств.

Значения свойств не обязательно должны быть только строковыми. Они могут быть любых других типов данных, допустимых в Java. Для того чтобы пользователи могли выбирать значения произвольных типов данных, в ИСР предусмотрены специальные *редакторы свойств*. (Редакторы свойств могут быть встроены в ИСР или в компонент JavaBeans. Далее в этой главе будет показано, как написать собственные редакторы свойств.)

Рассмотрим действие простого редактора свойств на примере установки свойства `foreground`, которое определяет цвет переднего плана и относится к типу `Color`. В редакторе цвета предоставляется поле, содержащее строку `[0,0,0]` и кнопку с меткой многоточия (. . .), для перехода в диалоговое окно редактирования цвета. Цвет текста метки изменится, как только будет отредактировано свойство `foreground`.

Но намного интереснее поэкспериментировать с выбором файла изображения. Как только имя файла изображения будет выбрано в соответствующем поле редактора свойств, изображение появится в компоненте `ImageViewerBean`.

НА ЗАМЕТКУ! Если присмотреться повнимательнее к панели инспектора свойств в NetBeans, то на ней можно обнаружить немало свойств с загадочными названиями вроде `focusCycleRoot` и `paintingForPrint`. Эти свойства наследуются от суперкласса `JLabel`. О том, как их подавлять в инспекторе свойств, речь пойдет далее.

Чтобы завершить составление приложения, рассматриваемого здесь в качестве примера, разместите во фрейме объект типа `FileNameBean`. При изменении свойства `fileName` этого объекта должно загружаться новое изображение. Обеспечить такое поведение можно с помощью события типа `PropertyChangeEvent`. Подробнее о нем и других подобных событиях речь пойдет далее в этой главе.

Для того чтобы организовать должное реагирование на это событие, выберите компонент `FileNameBean` и перейдите на вкладку **Events** (События) в окне инспектора его свойств. Далее щелкните на кнопке с меткой многоточия (. . .) рядом с элементом ввода `propertyChange`. В итоге откроется диалоговое окно, показывающее, что на текущий момент с данным событием пока еще не связан ни один из обработчиков. Щелкните в этом окне на кнопке **Add** (Добавить). После этого появится окно с приглашением ввести имя метода обработки данного события (рис. 8.8). Введите в нем имя `loadImage`.

Теперь обратите внимание на окно редактора исходного кода. Там должен появиться код обработки событий и новый метод, как показано ниже.

```
private void loadImage(java.beans.PropertyChange evt)
{
  // (ЧТО СДЕЛАТЬ: ввести здесь код обработки событий)
}
```

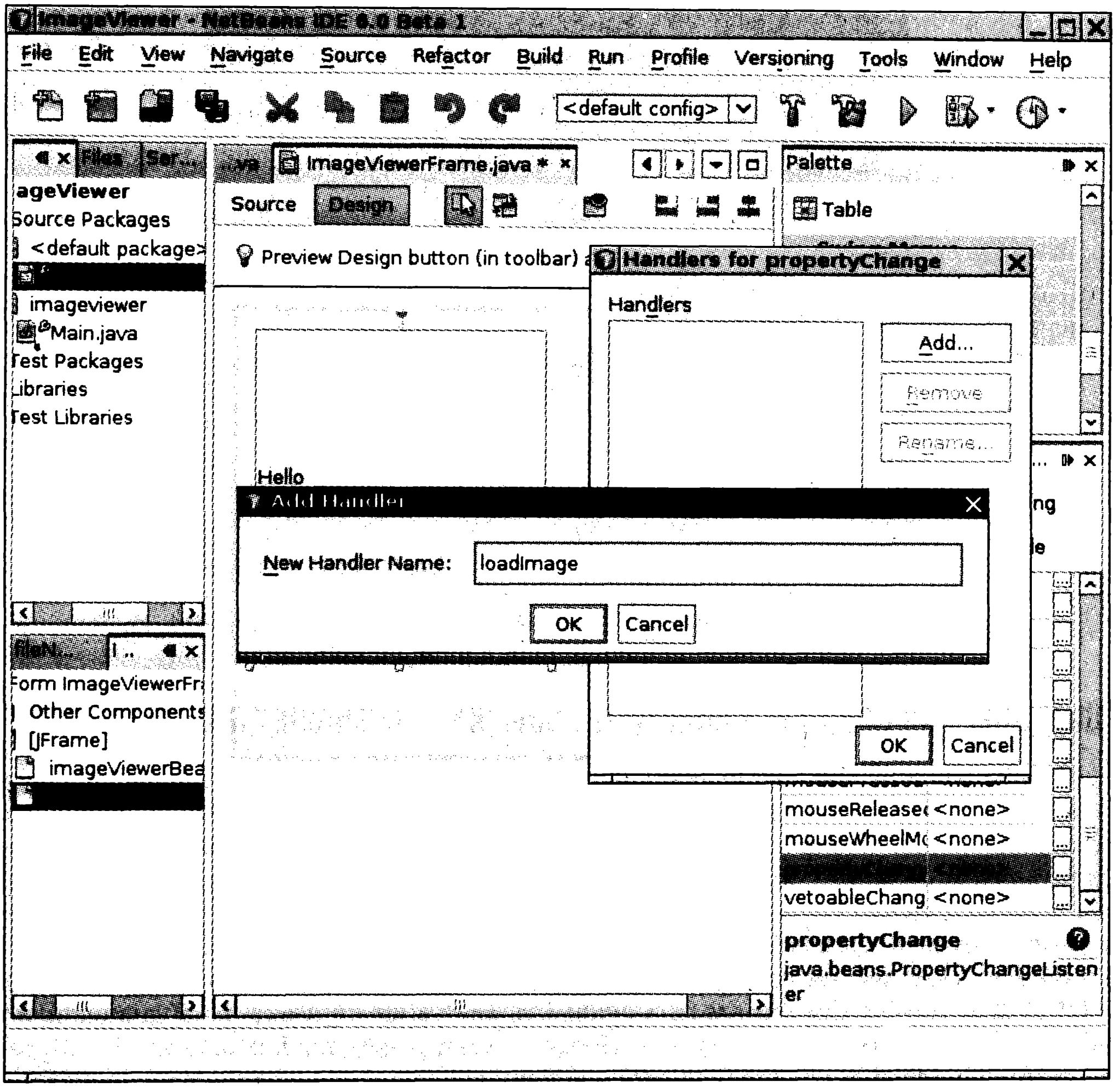


Рис. 8.8. Ввод события в компонент JavaBeans

Введите в теле этого метода следующую строку кода:

```
imageViewerBean1.setFileName(filePickerBean1.getFileName());
```

Затем скомпилируйте и запустите класс фрейма на выполнение. В итоге вы получите в свое распоряжение готовое приложение для просмотра изображений. Щелкните на кнопке с меткой многоточия и выберите файл изображения. Выбранное изображение появится в окне просмотра (рис. 8.9).

Описанный выше процесс демонстрирует порядок составления приложений Java из компонентов JavaBeans. Для этого достаточно указать их свойства и написать небольшие фрагменты кода обработки соответствующих событий.

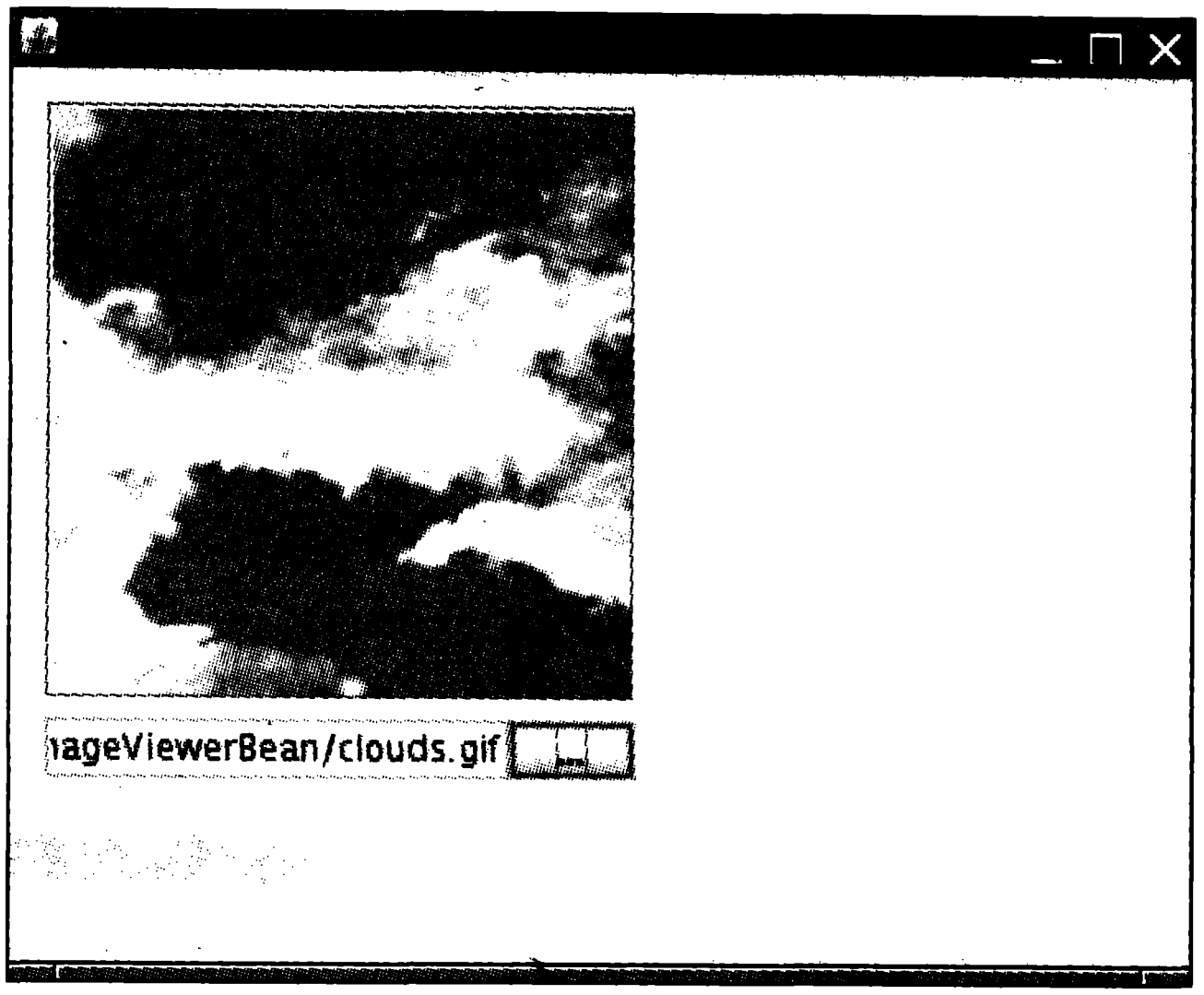


Рис. 8.9. Готовое приложение для просмотра изображений

Шаблоны именования свойств и событий в компонентах JavaBeans

В этом разделе рассматриваются основные правила разработки новых компонентов. Прежде всего необходимо подчеркнуть, что для JavaBeans не существует никакого особого класса, который мог бы выступать в роли суперкласса компонента и определять его поведение. Все визуальные компоненты JavaBeans прямо или косвенно наследуют от класса `Component`, тогда как невизуальные компоненты совсем не обязательно должны наследовать от какого-нибудь суперкласса. Напомним, что компонент JavaBeans представляет собой *произвольный* класс, которым можно манипулировать в ИСР. Для того чтобы определить сущность класса компонента, ИСР анализирует не суперкласс, а имена его методов. Поэтому имена методов компонента JavaBeans должны соответствовать определенным шаблонам.

НА ЗАМЕТКУ! Существует класс `java.beans.Beans`, но все его методы являются статическими. Следовательно, создавать его подклассы не имеет никакого смысла. К тому же нельзя организовать класс, который наследовал бы одновременно от классов `Beans` и `Components`. Поэтому такой подход не годится для визуальных компонентов JavaBeans. В действительности класс `Beans` содержит методы, предназначенные для вызова из ИСР, например, для того, чтобы проверить, находится ли ИСР на стадии разработки или выполнения проекта.

В некоторых языках программирования, например в Visual Basic и C#, предусмотрены ключевые слова `Property` и `Event`. Создатели спецификации языка Java решили не включать специальные ключевые слова для поддержки визуального программирования, поэтому анализ компонентов JavaBeans и определение их свойств и событий осуществляется другими средствами. Если автор компонента придерживается стандартных шаблонов именования, то в ИСР может применяться механизм рефлексии для определения нужных свойств и событий в компонентах JavaBeans.

Кроме того, автор компонента JavaBeans может воспользоваться специальным *классом информации* о компоненте JavaBeans. Такой класс содержит описание свойств и событий, требующихся для ИСР. Начнем с более простого способа — применения шаблонов именования, а затем покажем, как пользоваться классами информации о компонентах JavaBeans.

НА ЗАМЕТКУ! Несмотря на то что в документации стандартные шаблоны именования называются шаблонами проектирования, на самом деле они не имеют ничего общего с шаблонами проектирования, применяемыми в ООП.

Шаблоны именования свойств имеют очень простую структуру. Для чтения и записи отдельного свойства в таком шаблоне предусмотрена следующая пара методов получения и установки:

```
public Type getИмяСвойства()
public void setИмяСвойства(Тип новоеЗначение)
```

Например, у компонента `ImageViewerBean` имеется только одно свойство, допускающее чтение и запись. Это свойство содержит имя файла, а для его получения и установки используются следующие методы:

```
public String getFileName()
public void setFileName(String newValue)
```

Если в шаблоне именования предусмотрен лишь метод получения, то свойство доступно только для чтения. А если в шаблоне именования предусмотрен лишь метод установки, то свойство доступно только для записи.

НА ЗАМЕТКУ! Созданные методы получения и установки, как и любые другие методы в Java, могут выполнять произвольные действия, а не только возвращать и устанавливать значения в полях данных. Например, метод `setFileName()` из класса `ImageVieverBean` не только устанавливает значение в поле `fileName`, но и открывает файл и загружает изображение.

НА ЗАМЕТКУ! Для обращения со свойствами в Visual Basic и C# также используются методы получения и установки. Но в обоих языках явно указывается определение свойства, чтобы в ИСР можно было определить намерения программирующего. Средства обращения со свойствами, применяемые в этих языках, имеют еще одно преимущество: указание имени свойства в левой части выражения автоматически связано с вызовом метода установки, а указание имени свойства в самом выражении автоматически связано с вызовом метода получения. Например, в Visual Basic можно написать следующую строку кода:

```
imageBean.fileName = "corejava.gif";
```

вместо такой строки:

```
imageBean.setFileName("corejava.gif");
```

При создании языка Java рассматривался вопрос о целесообразности подобного подхода, но разработчики посчитали, что не следует скрывать имена методов за синтаксическими конструкциями, похожими на обычный доступ к полю данных.

Из правила для шаблонов именования методов получения и установки имеется следующее исключение: для свойств логического типа вместо префиксов **get/set** следует использовать префиксы **is/set**, как показано ниже.

```
public boolean isИмяСвойства()
public void setИмяСвойства(boolean b)
```

Например, компонент JavaBeans для анимации может иметь свойство `running` с двумя приведенными ниже методами. Таким образом, метод `setRunning()` может запускать и останавливать анимацию, а метод `isRunning()` — сообщать о текущем состоянии анимации.

```
public boolean isRunning()
public void setRunning(boolean b)
```

НА ЗАМЕТКУ! Для доступа к логическим свойствам допускается использовать методы с префиксом **get**, но префикс **is** все же предпочтительнее.

Создавая методы, обращайте внимание на использование прописных букв в их именах. Так, рассматриваемое здесь в качестве примера свойство должно иметь имя `fileName`, где буква **f** — строчная, тогда как в именах методов получения и установки указывается прописная буква **F**: `getFileName()`, `setFileName()`. Для определения имени свойства анализатор компонентов использует процедуру так называемого *приведения к нижнему регистру*. (В этом случае к нижнему регистру приводится первый символ, следующий за префиксом **get** или **set**.) Благодаря этому получаются имена свойств и методов, привычные для большинства программистов.

Если же прописными оказываются *две* первые буквы в имени свойства (такому свойству, например, соответствует метод `getURL()`), то первая прописная буква вообще *не* преобразуется в строчную. В самом деле, свойство под именем `uRL` выглядело бы, по меньшей мере, странно.

НА ЗАМЕТКУ! А что делать, если в классе имеются методы получения и установки, не имеющие никакого отношения к свойствам, доступным для манипулирования в инспекторе свойств? Если класс создается сызнова, то для таких методов лучше всего придумать другие имена. Но может случиться так, что методы с подобными именами наследуются от суперкласса. Так, если компонент создается как подкласс, производный от класса `JPanel` или `JLabel`, то в инспекторе свойств окажутся доступными многие малоинтересные свойства. Далее в этой главе будет показано, как отменить процесс автоматического обнаружения свойств, предоставив нужную *информацию о компоненте*. Подобным способом можно явно указать, какие именно свойства компонента JavaBeans должны быть доступны в ИСР.

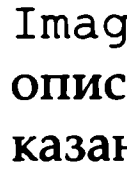
Шаблон именования событий еще проще. Интегрированная среда разработки определяет события, генерируемые компонентом JavaBeans, при наличии методов ввода и удаления их обработчиков. Имена классов событий должны оканчиваться суффиксом **Event**, а сами классы должны расширять класс `EventObject`.

Допустим, компонент JavaBeans генерирует событие типа *ИмяСобытия*`Event`. В таком случае интерфейс обработчика (приемника) подобных событий должен называться *ИмяСобытия*`Listener`, а методы управления их приемниками — следующим образом:

```
public void add<i>ИмяСобытия</i>Listener(<i>ИмяСобытия</i>Listener e)
public void remove<i>ИмяСобытия</i>Listener(<i>ИмяСобытия</i>Listener e)
public <i>ИмяСобытия</i>Listener get<i>ИмяСобытия</i>Listeners()
```

Если проанализировать исходный код компонента `ImageViewerBean`, то можно обнаружить, что события в нем не предусмотрены. Но многие компоненты Swing генерируют события, именуемые по данному шаблону. Например, класс `AbstractButton` генерирует объекты типа `ActionEvent` и содержит следующие методы для управления объектами типа `ActionListener`:

```
public void add<b>Action</b>Listener(<b>Action</b>Listener e)
public void remove<b>Action</b>Listener(<b>Action</b>Listener e)
<b>Action</b>Listener[] get<b>Action</b>Listeners()
```



ВНИМАНИЕ! Несмотря на то что класс события не является производным от класса `EventObject`, он, скорее всего, будет скомпилирован, поскольку ни один из методов из класса `EventObject` в действительности не требуется. Но не исключено возникновение необъяснимой ошибки из-за того, что механизм самоанализа не распознает события.

Типы свойств компонентов JavaBeans

В сложном компоненте JavaBeans доступно немало различных свойств и событий. Эти свойства могут быть как очень простыми подобно свойству `fileName` компонентов `ImageViewerBean` и `FileNameBean`, так и очень сложными, аналогично палитре цветов или массиву точек на графике — оба этих случая будут более подробно рассматриваться далее. В спецификации JavaBeans допускается использовать свойства четырех типов, поочередно описываемых в последующих разделах.

Простые свойства

Это свойства, которые содержат одно значение, например символьную строку или число. Примером простого свойства служит свойство `fileName` компонента `ImageViewerBean`. Простое свойство довольно легко программируется с помощью описанного выше шаблона именования методов его получения и установки. Как показано в листинге 8.1, для создания простого свойства типа `String` достаточно создать приведенные ниже методы получения и установки.

```
public void setFileName(String f)
{
  fileName = f;
  image = . . .
  repaint();
}

public String getFileName()
{
  if (file == null) return "";
  else return file.getPath();
}
```

Индексируемые свойства

Индексируемое свойство позволяет задать массив. Если используется индексируемое свойство, то следует предоставить две пары методов получения и установки: одну — для массива, а другую — для отдельных его элементов. Они должны следовать приведенному ниже шаблону именования.

```
Тип[] get<i>ИмяСвойства</i>()
void set<i>ИмяСвойства</i>(<i>Тип</i>[] <i>новоеЗначение</i>)
Тип get<i>ИмяСвойства</i>(int i)
void set<i>ИмяСвойства</i>(int i, <i>Тип новоеЗначение</i>)
```

Например, в компоненте `FileNameBean` индексируемое свойство используется для расширений файлов. Для манипулирования этим свойством предоставляются следующие четыре метода:

```
private String[] extensions;
. . .
public String[] getExtensions() { return extensions; }
public void setExtensions(String[] newValue) { extensions = newValue; }
public String getExtensions(int i)
{
  if (0 <= i && i < extensions.length) return extensions[i];
  else return "";
}
public void setExtensions(int i, String newValue)
{
  if (0 <= i && i < extensions.length) extensions[i] = value;
}
```

Использовать метод set*ИмяСвойства*(int, *Тип*) для расширения массива нельзя. Для этого следует сначала создать вручную новый массив, а затем передать его методу set*ИмяСвойства*(*Тип*[]).

Связанные свойства

Связанные свойства уведомляют приемники событий об изменении их значений. Так, свойство `fileName` компонента `FileNameBean` является связанным. При изменении имени файла компонент `ImageViewerBean` автоматически уведомляется об этом и загружает новый файл.

Для того чтобы сделать свойство связанным, нужно реализовать два механизма.

1. При изменении значения свойства компонент JavaBeans должен посылать всем зарегистрированным приемникам событие типа `PropertyChangeEvent`. Такое изменение может происходить при вызове метода установки или изменении значения свойства каким-нибудь другим методом (например, приемником действий кнопки с меткой многоточия).
2. Для предоставления заинтересованным приемникам событий возможности автоматически регистрироваться компонент JavaBeans должен реализовать два следующих метода:

   ```
   void addPropertyChangeListener(PropertyChangeListener listener)
   void removePropertyChangeListener(PropertyChangeListener listener)
   ```

3. Рекомендуется также (но не требуется) предоставить следующий метод:

   ```
   PropertyChangeListener[] getPropertyChangeListeners()
   ```

В состав пакета `java.beans` входит служебный класс `PropertyChangeSupport`, способный самостоятельно (без участия разработчика) управлять приемниками событий. Чтобы воспользоваться им, нужно сначала ввести поле экземпляра этого класса следующим образом:

```
private PropertyChangeSupport changeSupport = new
PropertyChangeSupport(this);
```

Затем этому объекту следует делегировать выполнение операций ввода и удаления приемников событий изменения свойств, как показано ниже.

```
public void addPropertyChangeListener(PropertyChangeListener listener)
{
  changeSupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener)
{
  changeSupport.removePropertyChangeListener(listener);
}

public PropertyChangeListener[] getPropertyChangeListeners()
{
  return changeSupport.getPropertyChangeListeners();
}
```

После этого следует вызвать метод `firePropertyChange()` для объекта `PropertyChangeSupport`, чтобы при каждом изменении значения свойства соответствующее событие передавалось всем зарегистрированным приемникам. Этот метод принимает три параметра: имя свойства, прежнее и новое значения. Ниже приведен шаблонный код типичного метода получения связанного свойства.

```
public void setValue(Тип newValue)
{
  Тип oldValue = getValue();
  value = newValue;
  changeSupport.firePropertyChange("propertyName", oldValue, newValue);
}
```

Для уведомления об изменении индексируемого свойства делается следующий вызов:

```
changeSupport.fireIndexedPropertyChange(
    "propertyName", index, oldValue, newValue);
```

СОВЕТ. Если компонент JavaBeans расширяет какой-нибудь класс, который в конечном счете расширяет класс `JComponent`, то реализовывать методы `addPropertyChangeListener()`, `removePropertyChangeListener()` и `getPropertyChangeListeners()` не нужно, поскольку они уже реализованы в суперклассе `JComponent`. Для того чтобы уведомить приемники событий изменения свойств, достаточно вызвать метод `firePropertyChange()` из суперкласса `JComponent`. Уведомления об изменении индексируемых свойств, к сожалению, не поддерживаются.

Другие компоненты JavaBeans, которые требуется уведомлять об изменении свойств, должны иметь интерфейс `PropertyChangeListener`. Этот интерфейс содержит следующий единственный метод:

```
void propertyChange(PropertyChangeEvent event)
```

Объект типа `PropertyChangeEvent` содержит имя свойства, а также его старое и новое значения, получаемые с помощью методов `getPropertyName()`, `getOldValue()` и `getNewValue()`. Если тип свойства не относится к типу класса, то объекты, представляющие значения этого свойства, являются экземплярами обычных классов-оболочек.

Ограниченные свойства

Данный тип свойств называется *ограниченным* потому, что *любой* приемник событий может накладывать запрет на предлагаемые изменения, вынуждая возвращаться к предыдущим установкам. В библиотеке Java можно найти лишь несколько примеров ограниченных свойств. К их числу относится свойство `closed` из класса `JInternalFrame`. При попытке вызвать метод `setClosed(true)` для внутреннего фрейма оповещаются все его приемники событий типа `VetoableChangeListener`. И если хотя бы один из них генерирует исключение типа `PropertyVetoException`, то свойство `closed` не изменяется, а метод `setClosed()` генерирует точно такое же исключение. В частности, приемник событий типа `VetoableChangeListener` может накладывать запрет на закрытие фрейма, если его содержимое не было еще сохранено.

Для создания ограниченного свойства у компонента JavaBeans должны быть следующие два метода манипулирования объектами типа `VetoableChangeListener`:

```
public void addVetoableChangeListener(VetoableChangeListener listener);
public void removeVetoableChangeListener(VetoableChangeListener listener);
```

Кроме того, у него должен быть приведенный ниже метод для получения всех приемников событий.

```
VetoableChangeListener[] getVetoableChangeListeners()
```

Подобно служебному классу для управления приемниками событий изменения свойств, существует служебный класс `VetoableChangeSupport`, управляющий приемниками, которые могут накладывать запрет на изменение свойств. Поэтому разрабатываемый компонент должен содержать экземпляр такого класса:

```
private VetoableChangeSupport vetoSupport = new VetoableChangeSupport(this);
```

Ввод и удаление приемников событий делегируются данному объекту. Ниже приведен характерный пример его применения.

```
public void addVetoableChangeListener(VetoableChangeListener listener)
{
  vetoSupport.addVetoableChangeListener(listener);
}

public void removeVetoableChangeListener(VetoableChangeListener listener)
{
  vetoSupport.removeVetoableChangeListener(listener);
}
```

Обновление значения ограниченного свойства выполняется в компоненте JavaBeans в три этапа.

1. Уведомление всех приемников запрещаемых изменений о *намерении* изменить значение данного ограниченного свойства. (Для этого вызывается метод `fireVetoableChange()` из класса `VetoableChangeSupport`.)

2. Если ни один из приемников запрещаемых изменений не сгенерирует исключение типа `PropertyVetoException`, значение данного ограниченного свойства обновляется.

3. Уведомление всех приемников событий изменения свойств *подтвердить*, что изменения произошли.

В приведенном ниже фрагменте кода показано, каким образом устанавливается значение ограниченного свойства.

```
public void setValue(Тип newValue) throws PropertyVetoException
{
  Тип oldValue = getValue();
  vetoSupport.fireVetoableChange("value", oldValue, newValue);
  // завершено успешно, следовательно, запрет не наложен
  value = newValue;
  changeSupport.firePropertyChange("value", oldValue, newValue);
}
```

Очень важно, чтобы значение свойства оставалось прежним до тех пор, пока все зарегистрированные приемники запрещаемых изменений не согласятся с предложенными изменениями. С другой стороны, ни один приемник запрещаемых изменений не должен допускать, чтобы изменения, на которые он дает согласие, уже происходили. Поэтому единственный надежный способ уведомления в тот момент, когда значение свойства фактически изменяется, обеспечивает приемник событий изменения данного свойства.

НА ЗАМЕТКУ! Если разрабатываемый компонент JavaBeans расширяет класс `JComponent`, отдельный объект типа `VetoableChangeSupport` не требуется. Вместо него следует вызывать метод `fireVetoableChange()` из суперкласса `JComponent`. Следует, однако, иметь в виду, что установить приемник запрещаемых изменений для конкретного свойства в суперклассе `JComponent` нельзя. Приниматься и обрабатываться должны непременно все запрещаемые изменения.

В завершение обсуждения свойств компонентов JavaBeans приводится весь исходный код компонента `FileNameBean` (*листинг 8.2*). Этот компонент обладает индексируемым свойством `extension` и ограниченным свойством `filename`. А поскольку он расширяет класс `JPanel`, то объект типа `PropertyChangeSupport` явно не используется. Вместо этого применяются предусмотренные в классе `JPanel` возможности для управления приемниками событий изменения свойств.

Листинг 8.2. Исходный код из файла `filePicker/FilePickerBean.java`

```
1   package filePicker;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.io.*;
6   import java.util.*;
7   import javax.swing.*;
8   import javax.swing.filechooser.*;
9
10  /**
11    * Компонент JavaBeans для выбора файлов по именам
```

```
12    * @version 1.31 2012-06-10
13    * @author Cay Horstmann      \
14   */
15   public class FilePickerBean extends JPanel
16   {
17     private static final int XPREFSIZE = 200;
18     private static final int YPREFSIZE = 20;
19
20     private JButton dialogButton;
21     private JTextField nameField;
22     private JFileChooser chooser;
23     private String[] extensions = { "gif", "png" };
24     public FilePickerBean()
25     {
26       dialogButton = new JButton("...");
27        nameField = new JTextField(30);
28
29       chooser = new JFileChooser();
30       setPreferredSize(new Dimension(XPREFSIZE, YPREFSIZE));
31
32       setLayout(new GridBagLayout());
33       GridBagConstraints gbc = new GridBagConstraints();
34       gbc.weightx = 100;
35       gbc.weighty = 100;
36       gbc.anchor = GridBagConstraints.WEST;
37       gbc.fill = GridBagConstraints.BOTH;
38       gbc.gridwidth = 1;
39       gbc.gridheight = 1;
40       add(nameField, gbc);
41
42       dialogButton.addActionListener(new ActionListener()
43         {
44           public void actionPerformed(ActionEvent event)
45           {
46             chooser.setFileFilter(new FileNameExtensionFilter(
47                Arrays.toString(extensions), extensions));
48             int r = chooser.showOpenDialog(null);
49             if (r == JFileChooser.APPROVE_OPTION)
50             {
51               File f = chooser.getSelectedFile();
52               String name = f.getAbsolutePath();
53               setFileName(name);
54             }
55           }
56         });
57       nameField.setEditable(false);
58
59       gbc.weightx = 0;
60       gbc.anchor = GridBagConstraints.EAST;
61       gbc.fill = GridBagConstraints.NONE;
62       gbc.gridx = 1;
63       add(dialogButton, gbc);
64     }
65
66     /**
67      * Устанавливает свойство fileName
68      * @param newValue Новое имя файла
```

```
69     */
70     public void setFileName(String newValue)
71     {
72       String oldValue = nameField.getText();
73       nameField.setText(newValue);
74       firePropertyChange("fileName", oldValue, newValue);
75     }
76
77     /**
78      * Получает свойство fileName
79      * @return Возвращает имя выбранного файла
80     */
81     public String getFileName()
82     {
83       return nameField.getText();
84     }
85
86     /**
87      * Получает свойство extensions
88      * @return Возвращает расширения по умолчанию из селектора файлов
89     */
90     public String[] getExtensions()
91     {
92       return extensions;
93     }
94
95     /**
96      * Устанавливает свойство extensions
97      * @param newValue Новые расширения по умолчанию
98     */
99     public void setExtensions(String[] newValue)
100    {
101      extensions = newValue;
102    }
103
104    /**
105     * Получает одно из значений свойства extensions
106     * @param i Индекс значения свойства
107     * @return Возвращает значение по указанному индексу
108    */
109    public String getExtensions(int i)
110    {
111      if (0 <= i && i < extensions.length) return extensions[i];
112      else return "";
113    }
114
115    /**
116     * Устанавливает одно из значений свойства extensions
117     * @param i Индекс значения свойства
118     * @param newValue Новое значение по указанному индексу
119    */
120    public void setExtensions(int i, String newValue)
121    {
122      if (0 <= i && i < extensions.length) extensions[i] = newValue;
123    }
124 }
```

java.beans.PropertyChangeListener 1.1

- `void propertyChange(PropertyChangeEvent` *event*`)`

  Вызывается, когда инициируется событие изменения свойства.

java.beans.PropertyChangeSupport 1.1

- `PropertyChangeSupport(Object` *sourceBean*`)`

  Создает объект типа `PropertyChangeSupport`, управляющий приемниками событий изменения значений связанных свойств данного компонента.

- `void addPropertyChangeListener(PropertyChangeListener` *listener*`)`
- `void addPropertyChangeListener(String` *propertyName*`, PropertyChangeListener` *listener*`)` **1.2**

  Регистрируют обработчик событий изменения значений всех связанных свойств или конкретного связанного свойства.

- `void removePropertyChangeListener(PropertyChangeListener` *listener*`)`
- `void removePropertyChangeListener(String` *propertyName*`, PropertyChangeListener` *listener*`)` **1.2**

  Удаляют зарегистрированный ранее приемник событий изменения связанного свойства.

- `void firePropertyChange(String` *propertyName*`, Object` *oldValue*`, Object` *newValue*`)`
- `void firePropertyChange(String` *propertyName*`, int` *oldValue*`, int` *newValue*`)` **1.2**
- `void firePropertyChange(String` *propertyName*`, boolean` *oldValue*`, boolean newValue)` **1.2**

  Отправляют событие типа `PropertyChangeEvent` зарегистрированным приемникам.

- `void fireIndexedPropertyChange(String` *propertyName*`, int` *index*`, Object` *oldValue*`, Object` *newValue*`)` **5.0**
- `void fireIndexedPropertyChange(String` *propertyName*`, int` *index*`, int` *oldValue*`, int` *newValue*`)` **5.0**
- `void fireIndexedPropertyChange(String` *propertyName*`, int` *index*`, boolean` *oldValue*`, boolean` *newValue*`)` **5.0**

  Отправляет событие типа `IndexedPropertyChangeEvent` зарегистрированным приемникам.

- `PropertyChangeListener[] getPropertyChangeListeners()` **1.4**
- `PropertyChangeListener[] getPropertyChangeListeners(String` *propertyName*`)` **1.4**

  Получают приемники событий изменения всех связанных свойств или только одного связанного свойства с указанным именем.

java.beans.PropertyChangeEvent 1.1

- `PropertyChangeEvent(Object sourceBean, String propertyName, Object oldValue, Object newValue)`

  Создает объект типа `PropertyChangeEvent`, описывающий изменение значения свойства с `oldValue` на `newValue`.

- `String getPropertyName()`

  Возвращает имя свойства.

- `Object getOldValue();`

  Возвращает прежнее значение свойства.

- `Object getNewValue()`

  Возвращает новое значение свойства.

java.beans.IndexedPropertyChangeEvent 5.0

- `IndexedPropertyChangeEvent(Object sourceBean, String propertyName, Object oldValue, Object newValue, int index)`

  Создает новый объект типа `IndexedPropertyChangeEvent`, описывающий изменение значения данного свойства с `oldValue` на `newValue`.

- `int getIndex()`

  Возвращает индекс, по которому произошло изменение.

java.beans.VetoableChangeListener 1.1

- `void vetoableChange(PropertyChangeEvent event)`

  Вызывается при намерении изменить свойство. Если изменения запрещены, генерируется исключение типа `PropertyVetoException`.

java.beans.VetoableChangeSupport 1.1

- `VetoableChangeSupport(Object sourceBean)`

  Создает объект `PropertyChangeSupport`, управляющий приемниками событий изменения ограниченных свойств данного компонента.

- `void addVetoableChangeListener(VetoableChangeListener listener)`

- `void addVetoableChangeListener(String propertyName, VetoableChangeListener listener)` **1.2**

  Регистрируют приемник событий, заинтересованный в изменениях всех ограниченных свойств или только одного ограниченного свойства с указанным именем

- void removeVetoableChangeListener(VetoableChangeListener *listener*)
- void removeVetoableChangeListener(String *propertyName*, VetoableChangeListener *listener*) **1.2**

  Удаляют зарегистрированный ранее приемник запрещаемых изменений ограниченных свойств.
- void fireVetoableChange(String *propertyName*, Object *oldValue*, Object *newValue*)
- void fireVetoableChange(String *propertyName*, int *oldValue*, int *newValue*) **1.2**
- void fireVetoableChange(String *propertyName*, boolean *oldValue*, boolean *newValue*) **1.2**

  Отправляют объект типа VetoableChangeEvent зарегистрированным приемникам событий.
- VetoableChangeListener[] getVetoableChangeListeners() **1.4**
- VetoableChangeListener[] getVetoableChangeListeners(String propertyName) **1.4**

  Получают приемники изменений всех ограниченных свойств или только одного ограниченного свойства с указанным именем.

java.awt.Component 1.0

- void addPropertyChangeListener(PropertyChangeListener *listener*) **1.2**
- void addPropertyChangeListener(String *propertyName*, PropertyChangeListener *listener*) **1.2**

  Регистрируют приемник событий изменения всех связанных свойств или только одного связанного свойства с указанным именем.
- void removePropertyChangeListener(PropertyChangeListener *listener*) **1.2**
- void removePropertyChangeListener(String *propertyName*, PropertyChangeListener *listener*) **1.2**

  Удаляют зарегистрированный ранее приемник событий изменения связанных свойств.
- void firePropertyChange(String *propertyName*, Object *oldValue*, Object *newValue*) **1.2**

  Отправляет зарегистрированным приемникам событие типа PropertyChangeEvent.

javax.swing.JComponent 1.2

- void addVetoableChangeListener(VetoableChangeListener *listener*)

  Регистрирует приемник событий, заинтересованный в изменениях всех ограниченных свойств с ограничениями или только одного ограниченного свойства с указанным именем.
- void removeVetoableChangeListener(VetoableChangeListener *listener*)

  Удаляет зарегистрированный ранее приемник запрещаемых изменений ограниченных свойств.

- `void fireVetoableChange(String` *`propertyName`*`, Object` *`oldValue`*`, Object` *`newValue`*`)`

  Отправляет зарегистрированным приемникам событие типа `VetoableChangeEvent`.

`java.beans.PropertyVetoException 1.1`

- `PropertyVetoException(String` *`message`*`, PropertyChangeEvent` *`event`*`)`

  Генерирует новое исключение типа `PropertyVetoException`.

- `PropertyChangeEvent getPropertyChangeEvent()`

  Возвращает событие типа `PropertyChangeEvent`, связанное с изменением свойства, на которое был наложен запрет.

Классы типа `BeanInfo`

При использовании стандартных шаблонов именования методов в ИСР имеется возможность применять рефлексию для распознавания свойств и событий. На первый взгляд такой подход к разработке компонентов JavaBeans кажется простым, но в дальнейшем проявляются ограничения, присущие шаблонам именования. По мере усложнения компонента JavaBeans увеличивается количество его свойств, а среди них могут быть и такие, которые не обнаруживаются в шаблонах именования. Более того, у многих компонентов JavaBeans, как упоминалось выше, могут быть пары методов получения и установки, не имеющих *никакого* отношения к свойствам самих компонентов.

Если требуется более гибкий механизм для предоставления сведений о компоненте JavaBeans, то следует определить объект, класс которого реализует интерфейс `BeanInfo`. Когда предоставляется такой объект, инструменты ИСР обращаются к нему за всеми сведениями о средствах, поддерживаемых компонентом JavaBeans.

Имя класса со сведениями о компоненте JavaBeans должно образовываться добавлением к имени компонента суффикса **`BeanInfo`**. Например, класс сведений о компоненте JavaBeans, связанный с классом `ImageViewerBean`, должен называться `ImageViewerBeanBeanInfo`. Кроме того, этот класс должен входить в состав того же самого пакета, что и сам компонент.

Как правило, реализовывать все методы, объявленные в интерфейсе **`BeanInfo`**, не требуется. Вместо этого можно создать подкласс уже существующего служебного класса `SimpleBeanInfo`, в котором реализованы все методы интерфейса `BeanInfo`.

Чаще всего класс типа `BeanInfo` создается для получения контроля над свойствами компонента JavaBeans. Для каждого свойства формируется объект типа `PropertyDescriptor`, конструктору которого передается имя свойства и содержащий его класс компонента, как показано ниже. Затем в классе типа `BeanInfo` реализуется метод `getPropertyDescriptors()`, возвращающий массив всех дескрипторов свойств.

```
PropertyDescriptor descriptor = new PropertyDescriptor(
    "fileName", FilePickerBean.class);
```

Допустим, разработчику компонента `ImageViewerBean` требуется скрыть от пользователя ИСР все свойства, унаследованные от суперкласса `JLabel`, и предоставить ему только свойство `fileName`. Сделать это позволит приведенный ниже класс типа `BeanInfo`.

```
// класс сведений о компоненте ImageViewerBean
public class ImageViewerBeanBeanInfo extends SimpleBeanInfo
{
  private PropertyDescriptor[] propertyDescriptors;

  public ImageViewerBeanBeanInfo()
  {
    try
    {
       propertyDescriptors = new PropertyDescriptor[]
       {
         new PropertyDescriptor("fileName", FilePickerBean.class);
       }
    }
    catch (IntrospectionException e)
    {
      e.printStackTrace();
    }
  }

  public PropertyDescriptor[] getPropertyDescriptors()
  {
    return propertyDescriptors;
  }
}
```

Существуют также методы, возвращающие массивы `EventSetDescriptor[]` и `MethodDescriptor[]`, но они применяются значительно реже. Если один из этих методов возвращает пустое значение `null` (именно так реализованы методы из класса `SimpleBeanInfo`), то применяется стандартный шаблон именования. Но если переопределить метод таким образом, чтобы он возвращал не пустое значение `null`, а массив, то в этот массив придется включить *все* свойства, события или методы компонента JavaBeans.

НА ЗАМЕТКУ! Иногда требуется обобщенный код для распознавания свойств или событий произвольного компонента JavaBeans. В таком случае вызывается статический метод `getBeanInfo()` из класса `Introspector`. Класс `Introspector` конструирует класс типа `BeanInfo`, который полностью описывает компонент, принимая во внимание сведения из сопутствующих классов типа `BeanInfo`.

Заслуживает также внимания метод `getIcon()`, объявленный в интерфейсе `BeanInfo`. Он позволяет связать с компонентом JavaBeans пиктограмму, предоставляемую пользователем. Эта пиктограмма отображается на панели инструментов ИСР. По существу, можно задать четыре отдельных растра пиктограммы. В интерфейсе `BeanInfo` объявлены следующие четыре константы, определяющие стандартные размеры пиктограмм:

```
ICON_COLOR_16x16
ICON_COLOR_32x32
ICON_MONO_16x16
ICON_MONO_32x32
```

Ниже приведен пример использования служебного метода `loadImage()` в классе `SimpleBeanInfo` для загрузки изображений пиктограмм.

```
public class ImageViewerBeanBeanInfo extends SimpleBeanInfo
{
  private Image iconColor16;
  private Image iconColor32;
  private Image iconMono16;
  private Image iconMono32;

  public ImageViewerBeanBeanInfo()
  {
    iconColor16 = loadImage("ImageViewerBean_COLOR_16x16.gif");
    iconColor32 = loadImage("ImageViewerBean_COLOR_32x32.gif");
    iconMono16 = loadImage("ImageViewerBean_MONO_16x16.gif");
    iconMono32 = loadImage("ImageViewerBean_MONO_32x32.gif");
  }

  public Image getIcon(int iconType)
  {
    if (iconType == BeanInfo.ICON_COLOR_16x16) return iconColor16;
    else if (iconType == BeanInfo.ICON_COLOR_32x32) return iconColor32;
    else if (iconType == BeanInfo.ICON_MONO_16x16) return iconMono16;
    else if (iconType == BeanInfo.ICON_MONO_32x32) return iconMono32;
    else return null;
  }
}
```

java.beans.Introspector 1.1

- `static BeanInfo getBeanInfo(Class<?> beanClass)`

  Возвращает сведения об указанном классе компонента JavaBeans.

java.beans.BeanInfo 1.1

- `PropertyDescriptor[] getPropertyDescriptors()`

  Возвращает дескрипторы свойств компонента JavaBeans. Возврат пустого значения `null` указывает на то, что для поиска свойств должны использоваться шаблоны именования.

- `Image getIcon(int iconType)`

  Возвращает объект изображения типа `Image`, который может представлять компонент JavaBeans на панелях инструментов и других подобных элементах пользовательского интерфейса. Как упоминалось выше, для стандартных типов пиктограмм предусмотрены четыре константы.

java.beans.SimpleBeanInfo 1.1

- `Image loadImage(String resourceName)`

  Возвращает файл объекта изображения, связанный с данным ресурсом. Имя ресурса представляет собой путь, взятый относительно каталога, содержащего класс информации о компоненте JavaBeans.

java.beans.FeatureDescriptor 1.1

- `String getName()`
- `void setName(String name)`

  Получают или устанавливают имя элемента, используемое в программе.
- `String getDisplayName()`
- `void setDisplayName(String displayName)`

  Получают или устанавливают отображаемое имя элемента. По умолчанию принимается имя, возвращаемое методом `getName()`. В настоящее время поддержка имен элементов на разных языках отсутствует.
- `String getShortDescription()`
- `void setShortDescription(String text)`

  Получают или устанавливают символьную строку, которую можно использовать для краткого описания данного элемента в ИСР. По умолчанию принимается значение, возвращаемое методом `getDisplayName()`.
- `boolean isExpert()`
- `void setExpert(boolean b)`

  Получают или устанавливают признак, который может использоваться в ИСР, чтобы решить, следует ли предоставлять начинающему пользователю доступ к данному элементу.
- `boolean isHidden()`
- `void setHidden(boolean)`

  Получают или устанавливают признак, указывающий на то, что данный элемент следует скрыть в ИСР.

java.beans.PropertyDescriptor 1.1

- `PropertyDescriptor(String propertyName, Class<?> beanClass)`
- `PropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod, String setMethod)`

  Создают объект типа `PropertyDescriptor`. Генерируют исключение типа `IntrospectionException`, если в процессе самоанализа возникает ошибка. В первом конструкторе предполагается, что для методов получения и установки используются стандартные шаблоны именования.
- `Class<?> getPropertyType()`

  Возвращает объект типа `Class` для представления типа свойства.
- `Method getReadMethod()`
- `Method getWriteMethod()`

  Возвращают метод для получения или установки свойства.

java.beans.IndexedPropertyDescriptor 1.1

- `IndexedPropertyDescriptor(String` *`propertyName`*`, Class<?>` *`beanClass`*`)`
- `IndexedPropertyDescriptor(String` *`propertyName`*`, Class<?>` *`beanClass`*`, String` *`getMethod`*`, String` *`setMethod`*`, String` *`indexedGetMethod`*`, String` *`indexedSetMethod`*`)`

  Создают объект типа `IndexedPropertyDescriptor` для индексируемого свойства. В первом конструкторе предполагается, что для методов получения и установки используются стандартные шаблоны именования.

- `Method getIndexedReadMethod()`
- `Method getIndexedWriteMethod()`

  Возвращают метод для получения или установки индексируемого значения в свойстве.

Редакторы свойств

При добавлении в компонент JavaBeans свойства целочисленного или строкового типа оно автоматически появляется в инспекторе свойств. Что же произойдет при добавлении свойства более сложного типа, значения которого нельзя отредактировать просто в текстовом поле, например даты или цвета? Для этого придется предоставить специальный компонент, с помощью которого пользователь смог бы видоизменять значения подобных свойств. Такие компоненты называются *редакторами свойств*. Например, редактор свойств объекта даты может быть представлен в виде календаря, в котором пользователь выбирает дату. А редактор свойств объекта цвета должен предоставить пользователю возможность задавать красную, зеленую и синюю составляющие выбираемого цвета.

На самом деле в ИСР NetBeans имеется редактор свойств для выбора цвета. В нем также имеются редакторы свойств для объектов таких основных типов, как `String` (текстовое поле) и `boolean` (флажок).

Для того чтобы предоставить пользователю новый редактор свойств, необходимо выполнить ряд действий. Сначала следует создать сопутствующий класс информации о компоненте JavaBeans. Для этого нужно переопределить метод `getPropertyDescriptors()`, который возвращает массив объектов типа `PropertyDescriptor`. Необходимо создать по одному такому объекту для каждого свойства, которое должно отображаться в окне редактора, — *даже для тех свойств, для которых требуется используемый по умолчанию редактор*. Для построения объекта типа `PropertyDescriptor` следует предоставить имя свойства и содержащий его класс компонента JavaBeans, как показано ниже.

```
PropertyDescriptor descriptor = new PropertyDescriptor(
    "titlePosition", ChartBean.class);
```

Затем следует вызвать метод `setPropertyEditorClass()` из класса `PropertyDescriptor` следующим образом:

```
descriptor.setPropertyEditorClass(TitlePositionEditor.class);
```

Далее нужно создать массив описаний свойств разрабатываемого компонента JavaBeans. Например, компонент **ChartBean**, который рассматривается в этом разделе, имеет следующие пять свойств:

- свойство `graphColor` типа `Color`;
- свойство `title` типа `String`;
- свойство `titlePosition` типа `int`;
- свойство `values` типа `double[]`;
- свойство `inverse` типа `boolean`.

В листинге 8.3 приведен исходный код класса `ChartBeanBeanInfo`, в котором определены редакторы этих свойств. В этом классе достигаются следующие цели.

1. Метод `getPropertyDescriptors()` возвращает дескриптор каждого свойства. Для свойств `title` и `graphColor` используются редакторы по умолчанию. Это средства для редактирования символьных строк и выбора цвета, входящих в состав ИСР.
2. Для свойств `titlePosition`, `values` и `inverse` применяются специальные редакторы типа `TitlePositionEditor`, `DoubleArrayEditor` и `InverseEditor` соответственно.

Листинг 8.3. Исходный код из файла `chart/ChartBeanBeanInfo.java`

```
1  package chart;
2
3  import java.awt.*;
4  import java.beans.*;
5
6  /**
7   * Класс информации о компоненте JavaBeans для построения диаграмм.
8   * В этом классе определяются редакторы свойств
9   * @version 1.20 2007-10-05
10  * @author Cay Horstmann
11 */
12 public class ChartBeanBeanInfo extends SimpleBeanInfo
13 {
14   private PropertyDescriptor[] propertyDescriptors;
15   private Image iconColor16;
16   private Image iconColor32;
17   private Image iconMono16;
18   private Image iconMono32;
19   public ChartBeanBeanInfo()
20   {
21     iconColor16 = loadImage("ChartBean_COLOR_16x16.gif");
22     iconColor32 = loadImage("ChartBean_COLOR_32x32.gif");
23     iconMono16 = loadImage("ChartBean_MONO_16x16.gif");
24     iconMono32 = loadImage("ChartBean_MONO_32x32.gif");
25
26     try
27     {
28       PropertyDescriptor titlePositionDescriptor =
29          new PropertyDescriptor("titlePosition", ChartBean.class);
30       titlePositionDescriptor.setPropertyEditorClass(
```

```
31          TitlePositionEditor.class);
32       PropertyDescriptor inverseDescriptor =
33          new PropertyDescriptor("inverse", ChartBean.class);
34       inverseDescriptor.setPropertyEditorClass(InverseEditor.class);
35       PropertyDescriptor valuesDescriptor = new PropertyDescriptor(
36          "values", ChartBean.class);
37       valuesDescriptor.setPropertyEditorClass(DoubleArrayEditor.class);
38       propertyDescriptors = new PropertyDescriptor[] {
39          new PropertyDescriptor("title", ChartBean.class),
40          titlePositionDescriptor, valuesDescriptor,
41          new PropertyDescriptor("graphColor", ChartBean.class),
42          inverseDescriptor };
43    }
44    catch (IntrospectionException e)
45    {
46       e.printStackTrace();
47    }
48 }
49
50 public PropertyDescriptor[] getPropertyDescriptors()
51 {
52    return propertyDescriptors;
53 }
54
55 public Image getIcon(int iconType)
56 {
57    if (iconType == BeanInfo.ICON_COLOR_16x16) return iconColor16;
58    else if (iconType == BeanInfo.ICON_COLOR_32x32) return iconColor32;
59    else if (iconType == BeanInfo.ICON_MONO_16x16) return iconMono16;
60    else if (iconType == BeanInfo.ICON_MONO_32x32) return iconMono32;
61    else return null;
62 }
63 }
```

java.beans.PropertyDescriptor 1.1

- `PropertyDescriptor(String` *name*`, Class<?>` *beanClass*`)`

  Создает объект типа `PropertyDescriptor`.

Параметры:	*name*	Имя свойства
	beanClass	Класс компонента JavaBeans, которому принадлежит данное свойство

- `void setPropertyEditorClass(Class<?>` *editorClass*`)`

  Устанавливает класс редактора свойств, используемого для данного свойства.

java.beans.BeanInfo 1.1

- `PropertyDescriptor[] getPropertyDescriptors()`

  Возвращает дескрипторы для каждого свойства, которое должно отображаться в инспекторе свойств данного компонента JavaBeans.

На рис. 8.10 показан внешний вид компонента `ChartBean`. Вверху на этом рисунке отображается заголовок, который определяется свойством `title`. Он может размещаться слева, справа или по центру. Свойство `values` определяет значения точек на строящемся графике диаграммы. Если свойство `inverse` принимает логическое значение `true`, то фон становится цветным, а столбцы диаграммы — белыми. Код этого компонента можно найти среди загружаемого кода для данной книги. По существу, он представляет собой просто видоизмененный аплет диаграммы из главы 10 первого тома.

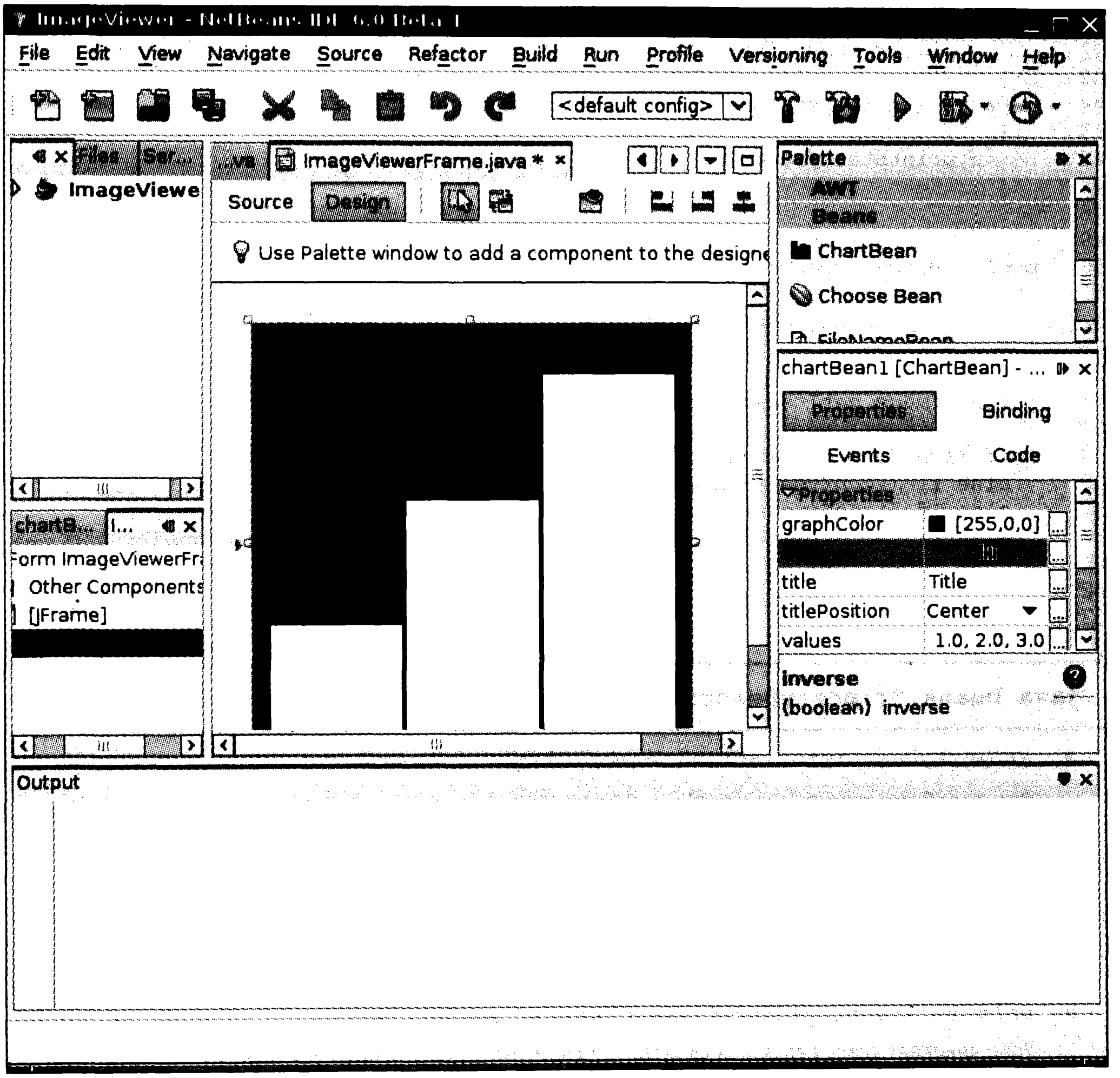


Рис. 8.10. Компонент `ChartBean`

Создание редакторов свойств

Прежде чем углубляться в особенности создания редакторов свойств, следует обратить внимание на то, что каждый такой редактор находится под управлением ИСР, а не компонента JavaBeans. В частности, при отображении инспектора свойств ИСР выполняет следующие действия для каждого свойства компонента JavaBeans.

1. Получает экземпляр редактора свойств.
2. Запрашивает компонент JavaBeans сообщить текущее значение свойства.
3. Запрашивает далее редактор свойств отобразить это значение.

Редактор свойств должен иметь конструктор без аргументов, а также обязательно реализовать интерфейс `PropertyEditor`. Обычно предпочтение отдается расширению такого служебного класса, как `PropertyEditorSupport`, который предоставляет версии методов, способные делать это по умолчанию.

Для каждого создаваемого редактора свойств выбирается один из трех следующих способов для отображения и редактирования значения свойства.

- В виде текстовой строки (с помощью методов `getAsText()` и `setAsText()`).
- В виде поля выбора (с помощью методов `getAsText()`, `setAsText()` и `getTags()`).
- Графически путем его прорисовки (с помощью методов `isPaintable()`, `paintValue()`, `supportsCustomEditor()` и `getCustomEditor()`).

Каждый из этих способов более подробно рассматривается в последующих подразделах.

Строковые редакторы свойств

Простые редакторы свойств работают с текстовыми строками. С этой целью переопределяются методы `setAsText()` и `getAsText()`. Например, у рассматриваемого здесь компонента `ChartBean` имеется свойство, которое позволяет выбирать один из следующих вариантов для размещения заголовка: слева, справа или по центру. Эти варианты реализуются с помощью такого перечисления:

```
public enum Position { LEFT, CENTER, RIGHT };
```

Разумеется, они не должны отображаться в виде строк с прописными буквами (`LEFT`, `CENTER`, `RIGHT`), поскольку это неприемлемо для пользовательского интерфейса. Вместо этого определяется редактор свойств, метод `getAsText()` которого подбирает для отображения строки в более приглядном с точки зрения разработчика виде:

```
class TitlePositionEditor extends PropertyEditorSupport
{
  private String[] tags = { "Left", "Center", "Right" };
  . . .
  public String getAsText()
  {
    int index = ((ChartBean.Position) getValue()).ordinal();
    return tags[index];
  }
}
```

В идеальном случае эти строки должны отображаться на языке текущих региональных настроек, а не только на английском. Но мы оставляем вам реализовать эту возможность самостоятельно в качестве упражнения.

С другой стороны, требуется предоставить следующий метод, преобразующий текстовую строку обратно в значение свойства:

```
public void setAsText(String s)
{
  int index = Arrays.asList(tags).indexOf(s);
```

```
  if (index >= 0) setValue(ChartBean.Position.values()[index]);
}
```

Если просто предоставить оба упомянутых выше метода, в инспекторе свойств появится текстовое поле. Оно будет инициализироваться при вызове метода `getAsText()`, а по завершении редактирования в этом поле будет вызван метод `setAsText()`. Для свойства `titlePosition`, конечно, такой вариант является далеко не самым подходящим. Поэтому лучше сделать так, чтобы все действительные значения данного свойства отображались не в текстовом поле, а в комбинированном списке, как показано на рис. 8.11.

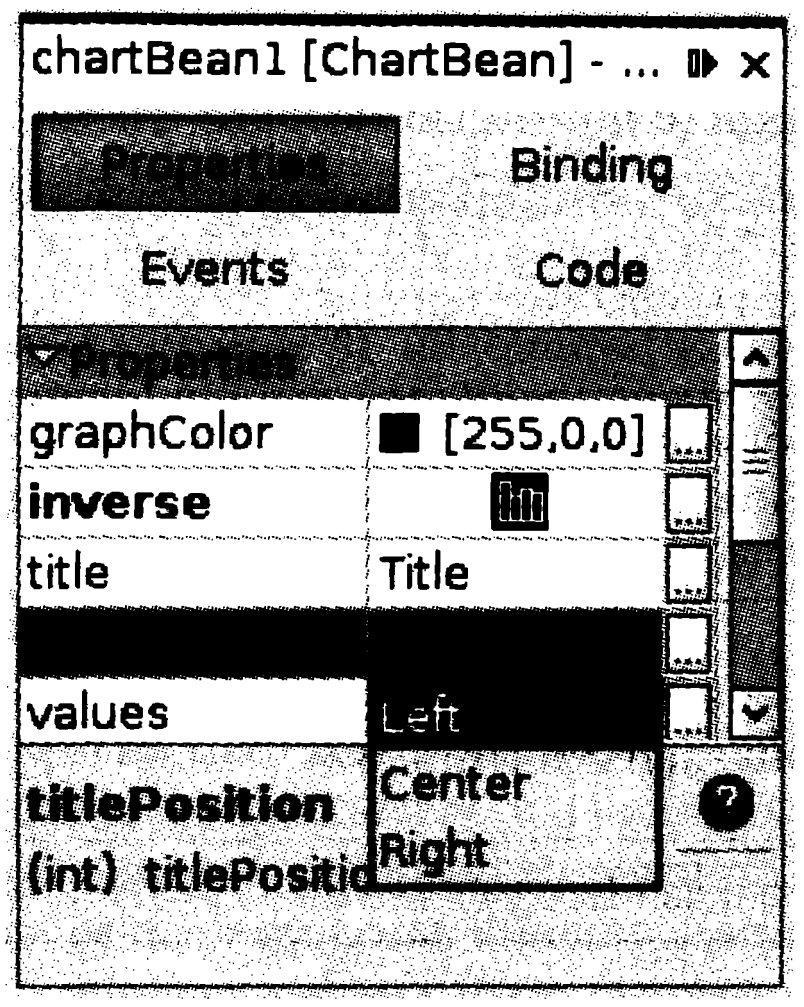


Рис. 8.11. Специальный редактор свойств в действии

В классе `PropertyEditorSupport` предоставляется простой механизм, позволяющий указать на необходимость использовать комбинированный список. Для этого достаточно написать приведенный ниже метод `getTags()`, возвращающий массив символьных строк. А используемый по умолчанию метод `getTags()` возвращает пустое значение `null`, указывающее на то, что для редактирования значения свойства подходит текстовое поле.

```
public String[] getTags() { return tags; }
```

Даже если предоставляется метод `getTags()`, то вместе с ним все равно нужно предоставить методы `getAsText()` и `setAsText()`. Метод `getTags()` просто перечисляет символьные строки, которые должны предлагаться пользователю на выбор, а методы `getAsText()` и `setAsText()` отвечают за прямое и обратное преобразование этих строк в тип данных редактируемого свойства (этот тип может быть строковым, целым, перечислимым или еще каким-нибудь другим).

И наконец, в редакторах свойств должен быть также реализован метод `getJavaInitializationString()`. С помощью этого метода можно передать в ИСР строку кода Java, устанавливающую текущее значение свойства, чтобы использовать ее для автоматического генерирования кода. Ниже показано, как должен выглядеть этот метод для рассматриваемого здесь редактора типа `TitlePositionEditor`. Он

возвращает символьную строку, подобную следующей: "com.horstmann.corejava.ChartBean.Position.LEFT".

```
public String getJavaInitializationString()
{
  return ChartBean.Position.class.getName().replace('$', '.')
     + "." + getValue();
}
```

Опробуйте его в ИСР NetBeans. При редактировании свойства titlePosition в ИСР NetBeans должна быть вставлена следующая строка кода:

```
chartBean1.setTitlePosition(chart.ChartBean.Position.LEFT);
```

В данном случае код получается несколько громоздким, поскольку метод ChartBean.Position.class.getName() возвращает символьную строку "com.horstmann.corejava.ChartBean$Position". Далее остается только заменить знак $ точкой и сложить результат вызова метода toString() с перечисляемым значением.

НА ЗАМЕТКУ! При наличии у свойства специального редактора, не реализующего метод getJavaInitializationString(), ИСР NetBeans не будет известно, как генерировать код, и поэтому будет создан метод установки с параметром ???.

Весь исходный код рассматриваемого здесь редактора свойств типа TitlePositionEditor приведен в листинге 8.4.

Листинг 8.4. Исходный код из файла chart/TitlePositionEditor.java

```
1  package chart;
2
3  import java.beans.*;
4  import java.util.*;
5
6  /**
7   * Специальный реактор свойства titlePosition компонента ChartBean.
8   * Этот редактор позволяет выбрать одно из следующих значений данного
9   * свойства: Left, Center или Right
10 * @version 1.20 2007-12-14
11 * @author Cay Horstmann
12 */
13   public class TitlePositionEditor extends PropertyEditorSupport
14   {
15     private String[] tags = { "Left", "Center", "Right" };
16
17     public String[] getTags()
18     {
19       return tags;
20     }
21
22     public String getJavaInitializationString()
23     {
24      return ChartBean.Position.class.getName().replace('$', '.')
25         + "." + getValue();
26     }
27
28     public String getAsText()
29     {
```

```
30        int index = ((ChartBean.Position) getValue()).ordinal();
31        return tags[index];
32      }
33
34      public void setAsText(String s)
35      {
36        int index = Arrays.asList(tags).indexOf(s);
37        if (index >= 0) setValue(ChartBean.Position.values()[index]);
38    }
39 }
```

Графические редакторы свойств

Сложные свойства не могут быть представлены в текстовом виде. Они должны иметь графическое представление и отображаться в инспекторе свойств в виде небольшой области на том же месте, где в противном случае могло бы находиться текстовое поле или комбинированный список. Если щелкнуть на этой области, появится диалоговое окно специального редактора, аналогичное приведенному на рис. 8.12.

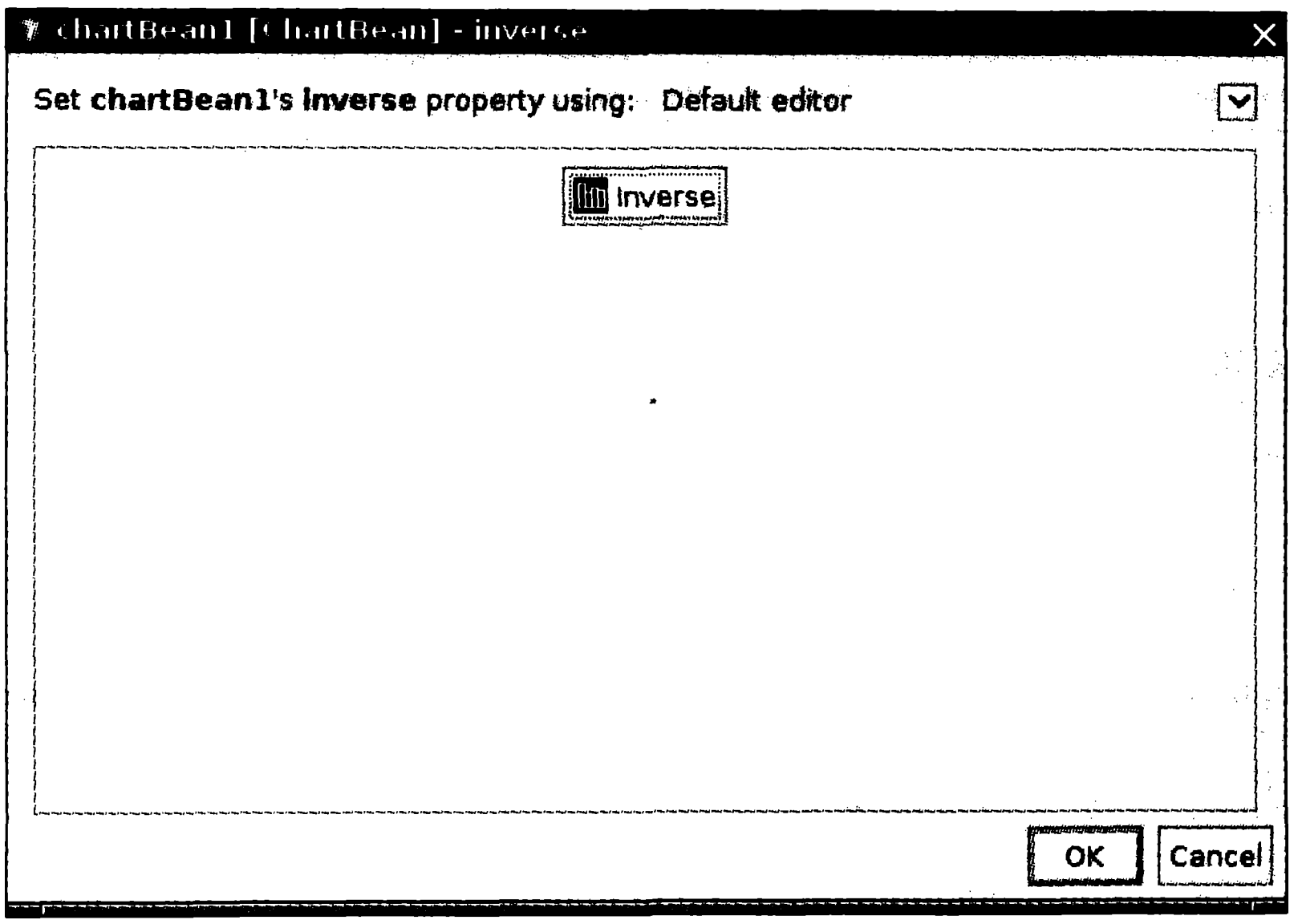


Рис. 8.12. Диалоговое окно специального графического редактора

В этом диалоговом окне содержится компонент для редактирования значения свойства, предоставляемый самим редактором свойств, а также различные кнопки, предоставляемые ИСР. В рассматриваемом здесь примере демонстрируется довольно простой графический редактор с единственной кнопкой. Но среди загружаемого кода к данной книге можно найти пример и более сложного графического редактора, позволяющего редактировать значения точек на строящемся графике диаграммы.

При создании графического редактора первым делом инспектору свойств сообщается, что значение будет отображаться в графическом, а не строковом режиме. Затем метод `getAsText()` из интерфейса `PropertyEditor` переопределяется таким образом, чтобы возвращать пустое значение `null`, а метод `isPaintable()` — чтобы возвращать логическое значение `true`.

После этого реализуется приведенный ниже метод `paintValue()`. Он получает контекст типа `Graphics` и координаты прямоугольника, внутри которого можно рисовать. Следует, однако, иметь в виду, что этот прямоугольник обычно имеет довольно мелкие размеры, поэтому нарисовать в нем что-нибудь сложное едва ли возможно. В рассматриваемом здесь примере в этом прямоугольнике рисуется одна из двух возможных пиктограмм, показанных ранее на рис. 8.11.

```
public void paintValue(Graphics g, Rectangle box)
{
  ImageIcon icon = (Boolean) getValue() ? inverseIcon : normalIcon;
  int x = bounds.x + (bounds.width - icon.getIconWidth()) / 2;
  int y = bounds.y + (bounds.height - icon.getIconHeight()) / 2;
  g.drawImage(icon.getImage(), x, y, null);
}
```

Безусловно, такое графическое представление не предназначено для редактирования. Пользователь должен просто щелкнуть на нем, чтобы открыть диалоговое окно специального редактора. А для того чтобы указать на применение специального редактора, следует переопределить метод `supportCustomEditor()` из интерфейса `PropertyEditor` таким образом, чтобы он возвращал логическое значение `true`. Далее метод `getCustomEditor()` из интерфейса `PropertyEditor` создает и возвращает объект класса этого специального редактора.

В листинге 8.5 приведен исходный код графического редактора типа `InverseEditor`, отображающий текущее значение свойства в инспекторе свойств, а в листинге 8.6 — исходный код панели этого специального редактора, на которой можно отредактировать значение свойства.

Листинг 8.5. Исходный код из файла `chart/InverseEditor.java`

```
1  package chart;
2
3  import java.awt.*;
4  import java.beans.*;
5  import javax.swing.*;
6
7  /**
8   * Редактор свойства inverse компонента ChartBean. Свойство inverse
9   * сменяет окраску столбиков диаграммы на окраску фона, и обратно,
10 * @version 1.30 2007-10-03
11 * @author Cay Horstmann
12 */
13 public class InverseEditor extends PropertyEditorSupport
14 {
15   private ImageIcon normalIcon = new ImageIcon(getClass().getResource(
16      "ChartBean_MONO_16x16.gif"));
17
18   private ImageIcon inverseIcon = new ImageIcon(getClass().getResource(
19      "ChartBean_INVERSE_16x16.gif"));
20
21   public Component getCustomEditor()
22   {
23     return new InverseEditorPanel(this);
24   }
25
26   public boolean supportsCustomEditor()
```

```
27   {
28     return true;
29   }
30
31   public boolean isPaintable()
32   {
33     return true;
34   }
35   public String getAsText()
36   {
37     return null;
38   }
39
40   public String getJavaInitializationString()
41   {
42     return "" + getValue();
43   }
44
45   public void paintValue(Graphics g, Rectangle bounds)
46   {
47     ImageIcon icon = (Boolean) getValue() ? inverseIcon : normalIcon;
48     int x = bounds.x + (bounds.width - icon.getIconWidth()) / 2;
49     int y = bounds.y + (bounds.height - icon.getIconHeight()) / 2;
50     g.drawImage(icon.getImage(), x, y, null);
51   }
52 }
```

Листинг 8.6. Исходный код из файла `chart/InverseEditorPanel.java`

```
1  package chart;
2
3  import java.awt.event.*;
4  import java.beans.*;
5  import javax.swing.*;
6
7  /**
8   * Панель для установки свойства inverse. Содержит кнопку
9   * для замены обычной окраски на противоположную
10 * @version 1.30 2007-10-03
11 * @author Cay Horstmann
12 */
13 public class InverseEditorPanel extends JPanel
14 {
15   private JButton button;
16   private PropertyEditorSupport editor;
17   private ImageIcon normalIcon = new ImageIcon(
18      getClass().getResource("ChartBean_MONO_16x16.gif"));
19
20   public InverseEditorPanel(PropertyEditorSupport ed)
21   {
22     editor = ed;
23     button = new JButton();
24     updateButton();
25     button.addActionListener(new ActionListener()
26       {
27         public void actionPerformed(ActionEvent event)
28         {
```

```
29            editor.setValue(!(Boolean) editor.getValue());
30            updateButton();
31          }
32        });
33      add(button);
34    }
35
36    private void updateButton()
37    {
38      if ((Boolean) editor.getValue())
39      {
40        button.setIcon(inverseIcon);
41        button.setText("Inverse");
42      }
43      else
44      {
45        button.setIcon(normalIcon);
46        button.setText("Normal");
47      }
48    }
49
50    private ImageIcon inverseIcon = new ImageIcon(getClass().getResource(
51       "ChartBean_INVERSE_16x16.gif"));
52  }
```

java.beans.PropertyEditor 1.1

- `Object getValue()`

  Возвращает текущее значение свойства. Основные типы данных упаковываются в оболочки соответствующих объектов.

- `void setValue(Object` *`newValue`*`)`

  Устанавливает новое значение свойства. Основные типы данных должны упаковываться в оболочки соответствующих объектов.

Параметры:	*`newValue`*	Новое значение объекта; это должен быть вновь созданный объект, которым может владеть данное свойство

- `String getAsText()`

  Этот метод должен быть переопределен таким образом, чтобы возвращать строковое представление текущего значения свойства. По умолчанию он возвращает пустое значение `null`, указывающее на то, что данное свойство не может быть представлено в виде символьной строки.

- `void setAsText(String` *`text`*`)`

  Этот метод должен быть переопределен таким образом, чтобы устанавливать новое значение свойства, получаемое путем синтаксического анализа текста. Он может генерировать исключение типа `IllegalArgumentException`, если текст не представляет допустимое значение или же если данное свойство не может быть представлено в виде символьной строки.

- `String[] getTags()`

  Этот метод должен быть переопределен таким образом, чтобы возвращать массив всех возможных строковых представлений значения свойства для их отображения в комбинированном списке.

  По умолчанию он возвращает пустое значение `null`, указывающее на то, что конечного множества строковых значений не существует.

- `boolean isPaintable()`

  Этот метод должен быть переопределен таким образом, чтобы возвращать логическое значение `true`, если для отображения свойства в классе используется метод `paintValue()`.

- `void paintValue(Graphics g, Rectangle` *`bounds`*`)`

  Этот метод должен быть переопределен таким образом, чтобы представлять значение свойства путем его рисования в графическом контексте области, специально отведенной для этой цели в компоненте, используемом для инспектора свойств.

- `boolean supportsCustomEditor()`

  Этот метод должен быть переопределен таким образом, чтобы возвращать логическое значение `true`, если у редактора свойств имеется какой-нибудь специальный редактор.

- `Component getCustomEditor()`

  Этот метод должен быть переопределен таким образом, чтобы возвращать компонент, реализующий специальный ГПИ для редактирования значения свойства.

- `String getJavaInitializationString()`

  Этот метод должен быть переопределен таким образом, чтобы возвращать строку кода Java, пригодную для генерирования кода, инициализирующего значение свойства, например `"0"`, `"new Color(64, 64, 64)"`.

Настройщики

Редактор свойств дает пользователю возможность устанавливать свойства только по очереди. Но в тех случаях, когда определенные свойства компонента JavaBeans связаны вместе, пользователям лучше предоставлять возможность редактировать сразу несколько свойств. Для обеспечения такой возможности вместо редакторов свойств (или в дополнение к ним) может предоставляться специальный *настройщик*.

Более того, некоторые компоненты JavaBeans могут иметь средства, которые нельзя предоставлять в виде свойств, а следовательно, редактировать с помощью инспектора свойств. Именно для таких компонентов настройщик имеет очень большое значение.

В рассматриваемом здесь примере программы демонстрируется настройщик компонента `ChartBean`, позволяющий устанавливать значения сразу нескольких свойств данного компонента в одном диалоговом окне, как показано на рис. 8.13.

Для того чтобы ввести настройщик в компонент JavaBeans, нужно создать класс типа `BeanInfo` и переопределить его метод `getBeanDescriptor()`:

```
public ChartBean2BeanInfo extends SimpleBeanInfo
{
  private BeanDescriptor beanDescriptor = new BeanDescriptor(
     ChartBean2.class, ChartBean2Customizer.class);
  . . .
  public BeanDescriptor getBeanDescriptor() { return beanDescriptor; }
}
```

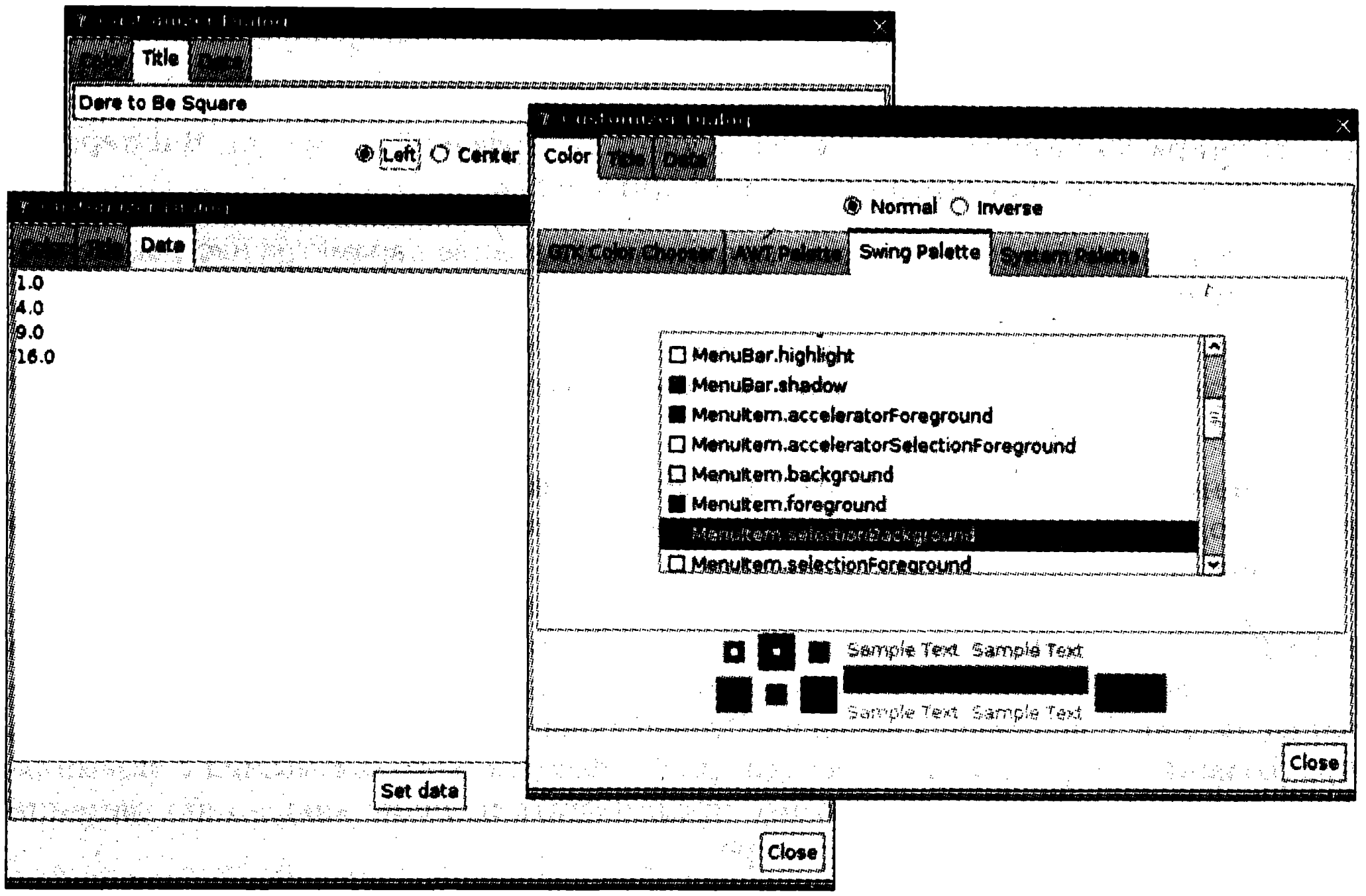


Рис. 8.13. Настройщик компонента `ChartBean`

Следует иметь в виду, что при создании класса настройщика придерживаться какого-то определенного шаблона именования совсем не обязательно. Тем не менее такие классы принято именовать следующим образом: *ИмяКомпонента*`JavaBeansCustomizer`. Реализация класса настройщика рассматривается в следующем разделе.

`java.beans.BeanInfo` 1.1

- `BeanDescriptor getBeanDescriptor()`

  Возвращает объект типа `BeanDescriptor`, описывающий возможности компонента JavaBeans.

`java.beans.BeanDescriptor` 1.1

- `BeanDescriptor(Class<?> beanClass, Class<?> customizerClass)`

  Создает объект типа `BeanDescriptor` для компонента JavaBeans, обладающего настройщиком.

  Параметры:

`beanClass`	Объект типа `Class` для компонента JavaBeans
`customizerClass`	Объект типа `Class` для настройщика компонента JavaBeans

Создание класса настройщика

Всякий создаваемый класс настройщика должен иметь конструктор без аргументов, расширять класс `Component` и реализовать интерфейс `Customizer`. В интерфейсе `Customizer` предусмотрено всего лишь три метода.

- Метод `setObject()`, который принимает в качестве параметра настраиваемый компонент `JavaBeans`.
- Методы `addPropertyChangeListener()` и `removePropertyChangeListener()`, управляющие рядом приемников событий, уведомляя их об изменении свойств в настройщике.

Внешний вид целевого компонента JavaBeans рекомендуется обновлять при изменении пользователем значения любого свойства, а не только по окончании процесса настройки.

В отличие от редакторов свойств, настройщики не отображаются автоматически. В ИСР NetBeans для вызова настройщика нужно щелкнуть правой кнопкой мыши на компоненте и выбрать из контекстного меню команду Customize (Настроить). В ответ на это в ИСР вызывается метод `setObject()` настройщика. Следует иметь в виду, что создание настройщика происходит до его фактического связывания с экземпляром компонента JavaBeans. Поэтому предполагать наличие каких-либо сведений о состоянии компонента в конструкторе нельзя.

Настройщики обычно предоставляют пользователям возможность выбора нескольких вариантов для работы со свойствами компонента JavaBeans, и зачастую им удобнее всего пользоваться панелью с несколькими вкладками. Именно такой подход и принят в рассматриваемом здесь примере программы. С этой целью класс настройщика расширяет класс `JTabbedPane`.

Настройщик размещает информацию на трех вкладках, обеспечивающих следующие возможности выбора:

- цвета и режима отображения;
- заголовка и его расположения;
- данных, отображаемых на диаграмме.

Безусловно, для создания такого пользовательского интерфейса придется выполнить немало рутинной работы. В данном примере один только конструктор состоит из более чем 100 строк кода. Но для его создания понадобятся только самые элементарные навыки программирования компонентов Swing, и поэтому мы не будем рассматривать несущественные подробности данного процесса.

При создании настройщика следует взять на вооружение следующий полезный прием. В настройщике нередко требуется редактировать значения свойств. И вместо того чтобы реализовывать новый интерфейс для установки значения свойства конкретного класса, можно просто найти уже существующий редактор свойств и добавить его в свой пользовательский интерфейс! В рассматриваемом здесь примере настройщика компонента `ChartBean` требуется устанавливать цвет диаграммы. Но, как известно, в NetBeans имеется отличный редактор свойств для выбора цветов, поэтому достаточно найти его следующим образом:

```
PropertyEditor colorEditor = PropertyEditorManager.findEditor(Color.Class);
Component colorEditorComponent = colorEditor.getCustomEditor();
```

После размещения компонентов их значения инициализируются с помощью метода `setObject()`, который вызывается при отображении настройщика. В качестве его параметра указывается настраиваемый компонент JavaBeans. Ссылка на этот компонент сохраняется, поскольку она понадобится в дальнейшем для уведомления компонента об изменении его свойств. Затем инициализируется каждый компонент пользовательского интерфейса. Фрагмент кода, выполняющий эту инициализацию в методе `setObject()` настройщика компонента `ChartBean`, выглядит следующим образом:

```
public void setObject(Object obj)
{
  bean = (ChartBean) obj;
  titleField.setText(bean.getTitle());
  colorEditor.setValue(bean.getGraphColor());
  . . .
}
```

И наконец, присоединяются обработчики событий для отслеживания действий пользователя. Всякий раз, когда пользователь изменяет значение свойства компонента, последний генерирует событие, которое настройщик должен непременно обработать. В частности, он должен делать следующее: обновлять значение соответствующего свойства в компоненте и дополнительно генерировать событие типа `PropertyChangeEvent`, чтобы другие обработчики (вроде инспектора свойств) также могли выполнить соответствующее обновление. Проследим этот процесс на примере нескольких элементов пользовательского интерфейса в настройщике компонента `ChartBean`.

Когда пользователь вводит новый заголовок, свойство заголовка требуется обновить. Поэтому обработчик типа `DocumentListener` присоединяется к тому текстовому полю, в котором пользователь вводит заголовок, как показано ниже.

```
titleField.getDocument().addDocumentListener(new DocumentListener()
  {
    public void changedUpdate(DocumentEvent event)
    {
      setTitle(titleField.getText());
    }
    public void insertUpdate(DocumentEvent event)
    {
      setTitle(titleField.getText());
    }
    public void removeUpdate(DocumentEvent event)
    {
      setTitle(titleField.getText());
    }
  });
```

Все три метода этого обработчика вызывают метод `setTitle()` настройщика. Метод `setTitle()`, в свою очередь, вызывает компонент JavaBeans, чтобы обновить значение его свойства, а затем генерирует событие изменения свойства. (Такое обновление необходимо только для несвязанных свойств.) Ниже приведен исходный код данного метода.

```
public void setTitle(String newValue)
{
  if (bean == null) return;
  String oldValue = bean.getTitle();
  bean.setTitle(newValue);
  firePropertyChange("title", oldValue, newValue);
}
```

Если значение цвета изменяется в редакторе свойств цвета, то обновление цвета диаграммы должно происходить в компоненте. Поэтому для отслеживания изменений цвета к этому редактору присоединяется приемник событий, как показано ниже. Как ни странно, этот редактор сам рассылает события, уведомляя об изменении значения свойства.

```
colorEditor.addPropertyChangeListener(new PropertyChangeListener()
   {
     public void propertyChange(PropertyChangeEvent event)
     {
       setGraphColor((Color) colorEditor.getValue());
     }
   });
```

Весь исходный код настройщика компонента `ChartBean` приведен в листинге 8.7.

Листинг 8.7. Исходный код из файла `chart2/ChartBeanCustomizer.java`

```
1   package chart2;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.beans.*;
6   import java.util.*;
7   import javax.swing.*;
8   import javax.swing.event.*;
9
10  /**
11   * Настройщик компонента JavaBeans для построения диаграмм. Позволяет
12   * править все свойства компонента в одном диалоговом окне с вкладками
13   * @version 1.12 2007-10-03
14   * @author Cay Horstmann
15  */
16  public class ChartBeanCustomizer extends JTabbedPane implements Customizer
17  {
18    private ChartBean bean;
19    private PropertyEditor colorEditor;
20    private JTextArea data;
21    private JRadioButton normal;
22    private JRadioButton inverse;
23    private JRadioButton[] position;
24    private JTextField titleField;
25
26    public ChartBeanCustomizer()
27    {
28      data = new JTextArea();
29      JPanel dataPane = new JPanel();
30      dataPane.setLayout(new BorderLayout());
31      dataPane.add(new JScrollPane(data), BorderLayout.CENTER);
32      JButton dataButton = new JButton("Set data");
33      dataButton.addActionListener(new ActionListener()
34        {
35          public void actionPerformed(ActionEvent event)
36          {
37            setData(data.getText());
38          }
39        });
40      JPanel panel = new JPanel();
```

```
41        panel.add(dataButton);
42        dataPane.add(panel, BorderLayout.SOUTH);
43
44        JPanel colorPane = new JPanel();
45        colorPane.setLayout(new BorderLayout());
46
47        normal = new JRadioButton("Normal", true);
48        inverse = new JRadioButton("Inverse", false);
49        panel = new JPanel();
50        panel.add(normal);
51        panel.add(inverse);
52        ButtonGroup group = new ButtonGroup();
53        group.add(normal);
54        group.add(inverse);
55        normal.addActionListener(new ActionListener()
56          {
57            public void actionPerformed(ActionEvent event)
58            {
59              setInverse(false);
60            }
61          });
62        inverse.addActionListener(new ActionListener()
63          {
64            public void actionPerformed(ActionEvent event)
65            {
66              setInverse(true);
67            }
68          });
69
70        colorEditor = PropertyEditorManager.findEditor(Color.class);
71        colorEditor.addPropertyChangeListener(new PropertyChangeListener()
72          {
73            public void propertyChange(PropertyChangeEvent event)
74            {
75              setGraphColor((Color) colorEditor.getValue());
76            }
77          });
78
79        colorPane.add(panel, BorderLayout.NORTH);
80        colorPane.add(colorEditor.getCustomEditor(), BorderLayout.CENTER);
81
82        JPanel titlePane = new JPanel();
83        titlePane.setLayout(new BorderLayout());
84
85        group = new ButtonGroup();
86        position = new JRadioButton[3];
87        position[0] = new JRadioButton("Left");
88        position[1] = new JRadioButton("Center");
89        position[2] = new JRadioButton("Right");
90
91        panel = new JPanel();
92        for (int i = 0; i < position.length; i++)
93        {
94          final ChartBean.Position pos = ChartBean.Position.values()[i];
95          panel.add(position[i]);
96          group.add(position[i]);
97          position[i].addActionListener(new ActionListener()
98            {
99              public void actionPerformed(ActionEvent event)
100             {
```

```
101             setTitlePosition(pos);
102           }
103         });
104     }
105
106     titleField = new JTextField();
107     titleField.getDocument().addDocumentListener(new DocumentListener()
108       {
109         public void changedUpdate(DocumentEvent evt)
110         {
111           setTitle(titleField.getText());
112         }
113
114         public void insertUpdate(DocumentEvent evt)
115         {
116           setTitle(titleField.getText());
117         }
118
119         public void removeUpdate(DocumentEvent evt)
120         {
121           setTitle(titleField.getText());
122         }
123       });
124
125     titlePane.add(titleField, BorderLayout.NORTH);
126     JPanel panel2 = new JPanel();
127     panel2.add(panel);
128     titlePane.add(panel2, BorderLayout.CENTER);
129     addTab("Color", colorPane);
130     addTab("Title", titlePane);
131     addTab("Data", dataPane);
132   }
133
134   /**
135    * Задает данные для отображения на диаграмме
136    * @param s Строка с отображаемыми числами, разделяемыми пробелом
137   */
138   public void setData(String s)
139   {
140     StringTokenizer tokenizer = new StringTokenizer(s);
141
142     int i = 0;
143     double[] values = new double[tokenizer.countTokens()];
144     while (tokenizer.hasMoreTokens())
145     {
146       String token = tokenizer.nextToken();
147       try
148       {
149         values[i] = Double.parseDouble(token);
150         i++;
151       }
152       catch (NumberFormatException e)
153       {
154       }
155     }
156     setValues(values);
157   }
158   /**
159    * Устанавливает заголовок диаграммы
160    * @param newValue Новый заголовок
```

```
161   */
162   public void setTitle(String newValue)
163   {
164     if (bean == null) return;
165     String oldValue = bean.getTitle();
166     bean.setTitle(newValue);
167     firePropertyChange("title", oldValue, newValue);
168   }
169
170   /**
171    * Задает положение заголовка диаграммы
172    * @param i Новое положение (ChartBean.LEFT, ChartBean.CENTER
173    *          или ChartBean.RIGHT)
174   */
175   public void setTitlePosition(ChartBean.Position pos)
176   {
177     if (bean == null) return;
178     ChartBean.Position oldValue = bean.getTitlePosition();
179     bean.setTitlePosition(pos);
180     firePropertyChange("titlePosition", oldValue, pos);
181   }
182
183   /**
184    * Устанавливает противоположную окраску диаграммы
185    * @param b Принимает логическое значение true, если окраска
186    *          диаграммы и фона меняется на противоположную
187   */
188   public void setInverse(boolean b)
189   {
190     if (bean == null) return;
191     boolean oldValue = bean.isInverse();
192     bean.setInverse(b);
193     firePropertyChange("inverse", oldValue, b);
194   }
195
196   /**
197     * Устанавливает значения, отображаемые на диаграмме
198     * @param newValue Массив новых значений
199   */
200   public void setValues(double[] newValue)
201   {
202     if (bean == null) return;
203     double[] oldValue = bean.getValues();
204     bean.setValues(newValue);
205     firePropertyChange("values", oldValue, newValue);
206   }
207   /**
208    * Устанавливает цвет окраски диаграммы
209    * @param newValue Новый цвет
210   */
211   public void setGraphColor(Color newValue)
212   {
213     if (bean == null) return;
214     Color oldValue = bean.getGraphColor();
215     bean.setGraphColor(newValue);
216     firePropertyChange("graphColor", oldValue, newValue);
217   }
218   public void setObject(Object obj)
219   {
220     bean = (ChartBean) obj;
```

```
221
222     data.setText("");
223     for (double value : bean.getValues())
224     data.append(value + "\n");
225
226     normal.setSelected(!bean.isInverse());
227     inverse.setSelected(bean.isInverse());
228
229     titleField.setText(bean.getTitle());
230
231     for (int i = 0; i < position.length; i++)
232     position[i].setSelected(i == bean.getTitlePosition().ordinal());
233
234     colorEditor.setValue(bean.getGraphColor());
235   }
236 }
```

java.beans.Customizer 1.1
• `void setObject(Object` *bean*`)` Определяет настраиваемый компонент JavaBeans.

Сохраняемость компонентов JavaBeans

Сохраняемость компонентов JavaBeans подразумевает использование свойств для их сохранения в потоке вывода и последующего восстановления на данной или на другой виртуальной машине. В этом отношении сохраняемость компонентов JavaBeans подобна сериализации объектов (см. главу 1). Но у этих механизмов имеется существенное отличие: сохраняемость компонентов JavaBeans *пригодна для долговременного хранения*.

При сериализации объектов поля экземпляра записываются в поток вывода. Если реализация класса изменится, могут измениться и его поля. В результате обычное чтение из файла, содержащего сериализированный объект старой версии, может дать некорректный результат при обращении с новой версией класса. В некоторых случаях удается учесть отличия между версиями и перенести новые представления данных в старые, но этот процесс требует слишком много времени и усилий и не всегда приводит к успеху. Следовательно, сериализация непригодна для длительного хранения объектов. Поэтому в документации для всех компонентов Swing делается следующее предупреждение: экземпляры данного класса могут оказаться несовместимыми с последующими реализациями, а хранение сериализированных объектов допустимо лишь на время, требующееся для их передачи из одного приложения в другое по протоколу RMI (Вызов удаленных методов).

В качестве выхода из этого затруднительного положения был изобретен механизм долговременной сохраняемости, первоначально предназначавшийся для разработки ГПИ методом перетаскивания. Инструментальные средства сохраняли в файле результаты действий мышью над фреймами, панелями, кнопками и другими компонентами Swing. А выполнявшаяся программа открывала этот файл и считывала его содержимое. Такой механизм позволял обойтись без обширных исходных кодов, предназначавшихся для размещения компонентов Swing и ввода данных. К сожалению, он не нашел широкого применения.

Основной принцип, положенный в основу сохраняемости компонентов JavaBeans, довольно прост. Допустим, объект типа `JFrame` требуется сохранить в файле, чтобы восстановить его впоследствии. Если проанализировать исходный код класса `JFrame` и его суперклассов, то в нем можно обнаружить десятки полей экземпляра. Если же сериализировать объект фрейма, то придется сохранить текущие значения всех его полей. Но обратите внимание на следующий процесс создания фрейма:

```
JFrame frame = new JFrame();
frame.setTitle("My Application");
frame.setVisible(true);
```

Конструктор без аргументов инициализирует все переменные экземпляра и устанавливает несколько свойств. При архивировании объекта `frame` механизм сохраняемости компонентов JavaBeans зафиксирует те же самые операторы в формате XML следующим образом:

```
<object class="javax.swing.JFrame">
  <void property="title">
    <string>My Application</string>
  </void>
  <void property="visible">
    <boolean>true</boolean>
  </void>
</object>
```

При восстановлении объекта операторы *выполняются*: создается объект типа `JFrame` и устанавливаются значения его свойств `title` и `visible`. При этом не имеет значения, изменилась ли внутренняя структура объекта типа `JFrame`. Подобным образом можно восстановить любой объект, устанавливая его свойства.

Следует заметить, что архивироваться должны только те свойства, значения которых отличаются от устанавливаемых по умолчанию. Объект типа `XMLEncoder` создает объект типа `JFrame` и сравнивает его свойства с архивируемым фреймом. А операторы установки формируются только для тех свойств, значения которых отличаются от устанавливаемых по умолчанию. Этот механизм называется *устранением избыточности*. В итоге размер архива оказывается значительно меньше, чем при сериализации. (Это особенно заметно при сериализации компонентов Swing. Их состояние определяется многими полями экземпляра, значения которых почти никогда не отличаются от устанавливаемых по умолчанию.)

Описанный выше механизм, конечно, не лишен недостатков. Рассмотрим, например, следующий вызов метода, который нельзя считать установкой значения свойства:

```
frame.setSize(600, 400);
```

Тем не менее класс `XMLEncoder` справляется и с этой задачей. Он записывает следующий оператор:

```
<void property="bounds">
  <object class="java.awt.Rectangle">
    <int>0</int>
    <int>0</int>
    <int>600</int>
    <int>400</int>
  </object>
</void>
```

Для того чтобы записать объект в поток вывода, необходимо создать экземпляр класса `XMLEncoder` следующим образом:

```
XMLEncoder out = new XMLEncoder(new FileOutputStream(. . .));
out.writeObject(frame);
out.close();
```

Аналогично, для чтения сохраненного объекта создается экземпляр класса `XMLDecoder`:

```
XMLDecoder in = new XMLDecoder(new FileInputStream(. . .));
JFrame newFrame = (JFrame) in.readObject();
in.close();
```

В примере программы, исходный код которой приведен в листинге 8.8, демонстрируется порядок сохранения и восстановления фрейма (рис. 8.14). Инициатором записи и чтения состояния является *сам* фрейм.

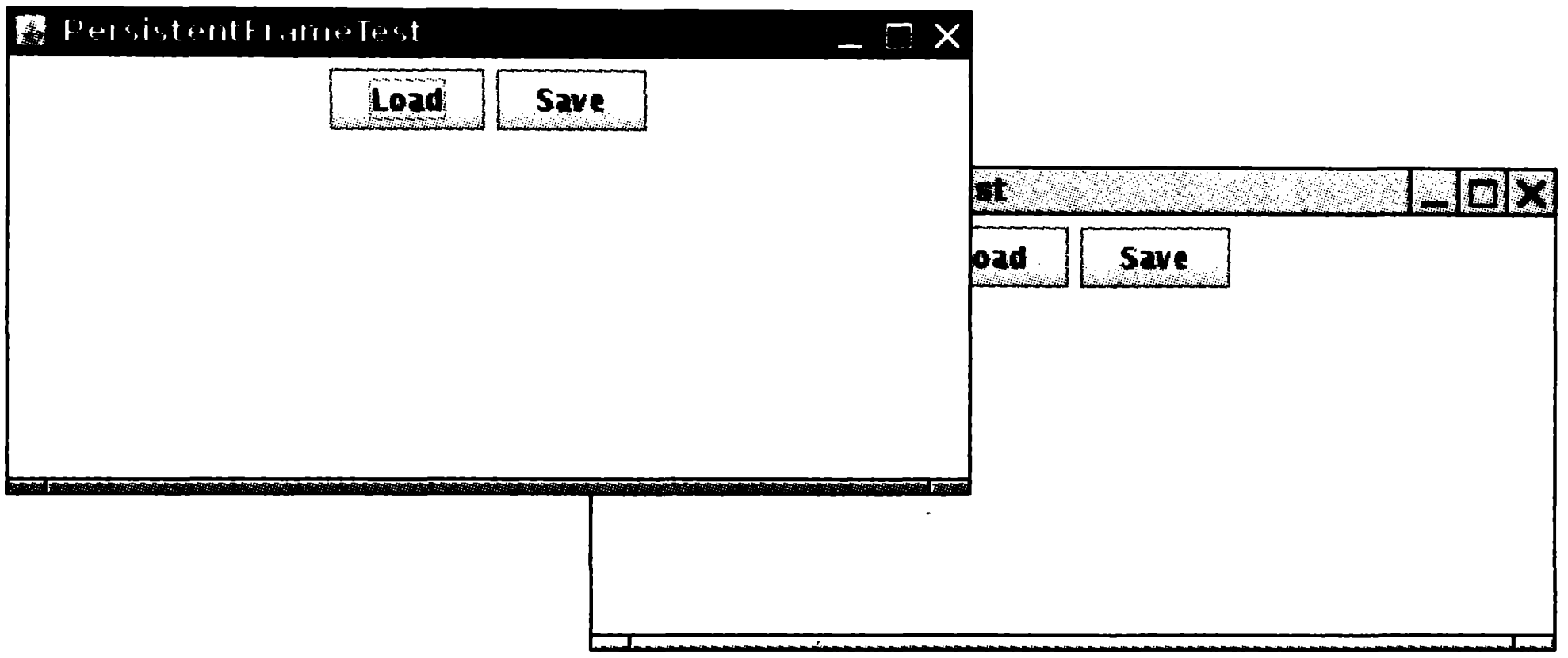


Рис. 8.14. Программа `PersistentFrameTest` в действии

Запустив эту программу на выполнение, щелкните сначала на кнопке Save и сохраните фрейм в каком-нибудь файле. Затем переместите исходный фрейм в другое место и щелкните на кнопке Load, чтобы посмотреть, как на том месте, где прежде находился исходный фрейм, появится новый. Загляните в XML-файл, который сгенерирует эта программа.

Если внимательно проанализировать данные, выводимые из рассматриваемой здесь программы в формате XML, то можно заметить, что объект типа `XMLEncoder` выполняет поразительно большой объем работы при сохранении фрейма. В частности, он генерирует операторы, выполняющие следующие действия.

- Устанавливают различные свойства фрейма (наподобие `size`, `layout`, `defaultCloseOperation`, `title` и т.д.).
- Вводят в фрейм различные кнопки.
- Присоединяют к кнопкам обработчики событий.

В данном случае обработчики событий (или приемники действий кнопок) пришлось создавать с помощью класса `EventHandler`. Ведь архивировать произвольные внутренние классы объект типа `XMLEncoder` не способен, но в то же время ему известно, как обрабатывать объекты типа `EventHandler`.

Листинг 8.8. Исходный код из файла `persistentFrame/PersistentFrameTest.java`

```
1  package persistentFrame;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.beans.*;
6  import java.io.*;
7  import javax.swing.*;
8
9  /**
10  * В этой программе демонстрируется применение кодера и
11  * декодера XML для сохранения и восстановления фрейма
12  * @version 1.01 2007-10-03
13  * @author Cay Horstmann
14 */
15 public class PersistentFrameTest
16 {
17   private static JFileChooser chooser;
18   private JFrame frame;
19
20   public static void main(String[] args)
21   {
22     chooser = new JFileChooser();
23     chooser.setCurrentDirectory(new File("."));
24     PersistentFrameTest test = new PersistentFrameTest();
25     test.init();
26   }
27   public void init()
28   {
29     frame = new JFrame();
30     frame.setLayout(new FlowLayout());
31     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
32     frame.setTitle("PersistentFrameTest");
33     frame.setSize(400, 200);
34
35     JButton loadButton = new JButton("Load");
36     frame.add(loadButton);
37     loadButton.addActionListener(EventHandler.create(
38        ActionListener.class, this, "load"));
39
40     JButton saveButton = new JButton("Save");
39     frame.add(saveButton);
40     saveButton.addActionListener(EventHandler.create(
41        ActionListener.class, this, "save"));
42
43     frame.setVisible(true);
44   }
45
46   public void load()
47   {
48     // показать диалоговое окно для выбора файлов
49     int r = chooser.showOpenDialog(null);
50
51     // если файл выбран, открыть его
52     if(r == JFileChooser.APPROVE_OPTION)
53     {
```

```
54        try
55        {
56          File file = chooser.getSelectedFile();
57          XMLDecoder decoder = new XMLDecoder(new FileInputStream(file));
58          decoder.readObject();
59          decoder.close();
60        }
61        catch (IOException e)
62        {
63          JOptionPane.showMessageDialog(null, e);
64        }
65      }
66    }
67
68    public void save()
69    {
70      if (chooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION)
71      {
72        try
73        {
74          File file = chooser.getSelectedFile();
75          XMLEncoder encoder = new XMLEncoder(new FileOutputStream(file));
76          encoder.writeObject(frame);
77          encoder.close();
78        }
79        catch (IOException e)
80        {
81          JOptionPane.showMessageDialog(null, e);
82        }
83      }
84    }
85  }
```

Применение сохраняемости компонентов JavaBeans к произвольным данным

Область применения сохраняемости компонентов JavaBeans не ограничивается только сохранением компонентов Swing. Этот механизм можно использовать и для записи в поток вывода *любой* совокупности объектов, удовлетворяющих простым условиям. В последующих подразделах будет показано, как пользоваться механизмом сохраняемости компонентов JavaBeans для долговременного хранения данных.

В класс `XMLEncoder` встроена поддержка следующих типов данных:

- Пустое значение `null`.
- Все простые типы и их оболочки.
- Перечисления (начиная с версии Java SE 6)
- Тип `String`.
- Массивы.
- Коллекции и отображения.
- Типы рефлексии `Class`, `Field`, `Method` и `Proxy`.
- Типы `Color`, `Cursor`, `Dimension`, `Font`, `Insets`, `Point`, `Rectangle` и `ImageIcon` из библиотеки AWT.
- Компоненты AWT и Swing, рамки, диспетчеры компоновки и модели.
- Обработчики событий.

Создание делегата сохраняемости для конструирования объекта

Применение сохраняемости компонентов JavaBeans не составляет особого труда, если состояние объектов изменяется в результате установки их свойств. Но поведение реальных программ не всегда вписывается в эти рамки. Возьмем, например, класс `Employee`, рассматривавшийся в главе 4 первого тома. Очевидно, что его нельзя считать полноценным компонентом JavaBeans. У него нет ни конструктора без аргументов, ни методов `setName()`, `setSalary()`, `setHireDay()`. Преодолеть этот недостаток можно, определив для этого класса так называемый *делегат сохраняемости*. Такой делегат отвечает за кодирование объекта типа `Employee` в формате XML.

Делегат сохраняемости для класса `Employee` должен переопределять метод `instantiate()` и порождать *выражение*, конструирующее объект данного класса, как показано ниже.

```
PersistenceDelegate delegate = new DefaultPersistenceDelegate()
  {
    protected Expression instantiate(Object oldInstance, Encoder out)
    {
      Employee e = (Employee) oldInstance;
      GregorianCalendar c = new GregorianCalendar();
      c.setTime(e.getHireDay());
      return new Expression(oldInstance, Employee.class, "new",
         new Object[]
         {
           e.getName(),
           e.getSalary(),
           c.get(Calendar.YEAR),
           c.get(Calendar.MONTH),
           c.get(Calendar.DATE)
         });
    }
  };
```

Приведенный выше фрагмент кода означает следующее: "*Для восстановления* объекта `oldInstance` вызывается метод `new` (т.е. конструктор) объекта типа `Employee.class` и предоставляются заданные параметры". *Имя параметра* `oldInstance` может в какой-то мере ввести в заблуждение, но под ним на самом деле подразумевается сохраняемый *экземпляр класса*.

Установить делегат сохраняемости можно двумя способами. Во-первых, сопоставить его с конкретным средством записи типа `XMLWriter`:

```
out.setPersistenceDelegate(Employee.class, delegate);
```

И во-вторых, установить атрибут `persistenceDelegate` *дескриптора компонента* JavaBeans типа `BeanInfo`:

```
BeanInfo info = Introspector.getBeanInfo(GregorianCalendar.class);
info.getBeanDescriptor().setValue("persistenceDelegate", delegate);
```

После установки делегата сохраняемости можно переходить непосредственно к сохранению объектов типа `Employee`. Например, следующие операторы:

```
Object myData = new Employee("Harry Hacker", 50000, 1989, 10, 1);
out.writeObject(myData);
```

выдают такие выводимые данные:

```
<object class="Employee">
  <string>Harry Hacker</string>
```

```
  <double>50000.0</double>
  <int>1989</int>
  <int>10</int>
  <int>1</int>
</object>
```

НА ЗАМЕТКУ! Настройке подлежит только процесс *кодирования*. Специальные методы декодирования не используются. Для декодирования достаточно выполнить операторы и выражения, обнаруживаемые среди данных, вводимых в формате XML.

Конструирование объекта из свойств

Если все параметры конструктора можно получить, обратившись к свойствам `oldInstance`, то создавать метод `instantiate()` не придется. Достаточно сконструировать объект типа `DefaultPersistenceDelegate` и предоставить ему имена свойств. Например, приведенный ниже оператор устанавливает делегат сохраняемости для класса `Rectangle2D.Double`.

```
out.setPersistenceDelegate(Rectangle2D.Double.class,
    new DefaultPersistenceDelegate(new String[]
    { "x", "y", "width", "height" }));
```

Этот фрагмент кода означает следующее: "Для кодирования объекта типа `Rectangle2D.Double` получить четыре его свойства, `x`, `y`, `width` и `height`, и передать их значения конструктору". В итоге будет сгенерирован следующий элемент разметки в коде XML:

```
<object class="java.awt.geom.Rectangle2D$Double">
  <double>5.0</double>
  <double>10.0</double>
  <double>20.0</double>
  <double>30.0</double>
</object>
```

Авторы классов могут поступить еще лучше, снабдив конструктор аннотацией `@ConstructorProperties`. Так, если бы у класса `Employee` был конструктор с тремя параметрами, определяющими, например, имя сотрудника, его заработную плату и день приема на работу, то его можно было бы снабдить подобной аннотацией следующим образом:

```
@ConstructorProperties({"name", "salary", "hireDay"})
public Employee(String n, double s, Date d)
```

Такая аннотация указывала бы кодеру на необходимость вызвать методы `getName()`, `getSalary()` и `getHireDay()` и записать результирующие значения в выражение `object`.

Аннотация `@ConstructorProperties` впервые появилась в версии Java SE 6 и до сих пор применялась только в классах из прикладного интерфейса JMX API, где JMX означает Java Management Extensions — управляющие расширения Java.

Конструирование объекта фабричным методом

Иногда приходится сохранять объекты, для создания которых использовались не конструкторы, а фабричные методы. Рассмотрим, например, следующий процесс формирования объекта типа `InetAddress`:

```
byte[] bytes = new byte[] { 127, 0, 0, 1};
InetAddress address = InetAddress.getByAddress(bytes);
```

В этом случае метод `instantiate()` объекта типа `PersistenceDelegate` должен содержать вызов фабричного метода:

```
protected Expression instantiate(Object oldInstance, Encoder out)
{
  return new Expression(oldInstance, InetAddress.class, "getByAddress",
  new Object[] { ((InetAddress) oldInstance).getAddress() });
}
```

В итоге формируются следующие выводимые данные:

```
<object class="java.net.Inet4Address" method="getByAddress">
  <array class="byte" length="4">
    <void index="0">
      <byte>127</byte>
    </void>
    <void index="3">
      <byte>1</byte>
    </void>
  </array>
</object>
```

ВНИМАНИЕ! При установке делегата сохраняемости следует использовать конкретный подкласс, например, `Inet4Address` вместо абстрактного класса `InetAddress`.

Порядок действий после конструирования объекта

Состояние некоторых классов формируется путем вызова методов, не устанавливающих свойства. В этом случае можно переопределить метод `initialize()` из класса `DefaultPersistenceDelegate`. Метод `initialize()` вызывается после метода `instantiate()`. С его помощью можно сгенерировать последовательность *операторов* и записать их в архив.

Рассмотрим в качестве примера класс `BitSet`. Для восстановления объекта типа `BitSet` нужно установить те же самые биты, что и в исходном объекте. Приведенный ниже метод `initialize()` генерирует необходимые для этого операторы.

```
protected void initialize(Class<?> type, Object oldInstance,
    Object newInstance, Encoder out)
{
    super.initialize(type, oldInstance, newInstance, out);
    BitSet bs = (BitSet) oldInstance;
    for (int i = bs.nextSetBit(0); i >= 0; i = bs.nextSetBit(i + 1))
    out.writeStatement(new Statement(bs, "set", new Object[]
       { i, i + 1, true } ));
}
```

В итоге формируются следующие выводимые данные:

```
<object class="java.util.BitSet">
  <void method="set">
    <int>1</int>
    <int>2</int>
    <boolean>true</boolean>
  </void>
  <void method="set">
    <int>4</int>
    <int>5</int>
```

```
    <boolean>true</boolean>
  </void>
</object>
```

НА ЗАМЕТКУ! На первый взгляд, было бы целесообразнее написать код `new Statement(bs, "set", new Object[] { i })`, но в этом случае средство записи типа `XMLWriter` сформирует довольно неуклюжий для восприятия оператор, устанавливающий свойство с пустым именем.

Переходные свойства

Иногда в классах встречаются свойства, методы получения и установки которых распознаются декодером типа `XMLDecoder`, но значения которых нежелательно включать в архив. Для того чтобы запретить запись свойства в архив, следует пометить его в дескрипторе как *переходное*. Например, приведенный ниже оператор помечает подобным образом свойство `removeMode` в классе `DamageReporter`, подробнее рассматриваемом в следующем разделе.

```
BeanInfo info = Introspector.getBeanInfo(DamageReport.class);
for (PropertyDescriptor desc : info.getPropertyDescriptors())
  if (desc.getName().equals("removeMode"))
    desc.setValue("transient", Boolean.TRUE);
```

В примере программы из листинга 8.9 демонстрируется применение различных делегатов сохраняемости. Следует, однако, иметь в виду, что в данной программе демонстрируется наихудший вариант. В реальных приложениях многие классы могут быть записаны в архив без применения делегатов сохраняемости.

Листинг 8.9. Исходный код из файла `persistenceDelegate/PersistenceDelegateTest.java`

```
1   package persistenceDelegate;
2
3   import java.awt.geom.*;
4   import java.beans.*;
5   import java.net.*;
6   import java.util.*;
7
8   /**
9    * В этой программе демонстрируется применение различных
10   * делегатов сохраняемости
11   * @version 1.01 2007-10-03
12   * @author Cay Horstmann
13  */
14  public class PersistenceDelegateTest
15  {
16    public static class Point
17    {
18      private final int x, y;
19
20      @ConstructorProperties( { "x", "y" })
21      public Point(int x, int y)
22      {
23        this.x = x;
24        this.y = y;
25      }
26
```

```
27      public int getX()
28      {
29        return x;
30      }
31
32      public int getY()
33      {
34        return y;
35      }
36
37    }
38
39    public static void main(String[] args) throws Exception
40    {
41      PersistenceDelegate delegate = new PersistenceDelegate()
42        {
43          protected Expression instantiate(Object oldInstance, Encoder out)
44          {
45            Employee e = (Employee) oldInstance;
46            GregorianCalendar c = new GregorianCalendar();
47            c.setTime(e.getHireDay());
48            return new Expression(oldInstance, Employee.class, "new",
49               new Object[] {e.getName(), e.getSalary(),
50               c.get(Calendar.YEAR), c.get(Calendar.MONTH),
51               c.get(Calendar.DATE) });
52          }
53        };
54      BeanInfo info = Introspector.getBeanInfo(Employee.class);
55      info.getBeanDescriptor().setValue("persistenceDelegate", delegate);
56
57      XMLEncoder out = new XMLEncoder(System.out);
58      out.setExceptionListener(new ExceptionListener()
59        {
60          public void exceptionThrown(Exception e)
61          {
62            e.printStackTrace();
63          }
64        });
65
66      out.setPersistenceDelegate(Rectangle2D.Double.class,
67         new DefaultPersistenceDelegate(
68        new String[] { "x", "y", "width", "height" }));
69
70      out.setPersistenceDelegate(Inet4Address.class,
71          new DefaultPersistenceDelegate()
72        {
73          protected Expression instantiate(Object oldInstance, Encoder out)
74          {
75            return new Expression(oldInstance, InetAddress.class,
76               "getByAddress", new Object[]
77               { ((InetAddress) oldInstance).getAddress() });
78          }
79        });
80
81      out.setPersistenceDelegate(BitSet.class,
82          new DefaultPersistenceDelegate()
83        {
84          protected void initialize(Class<?> type, Object oldInstance,
85             Object newInstance, Encoder out)
86          {
```

```
87              super.initialize(type, oldInstance, newInstance, out);
88              BitSet bs = (BitSet) oldInstance;
89              for (int i = bs.nextSetBit(0); i >= 0; i = bs.nextSetBit(i + 1))
90                out.writeStatement(new Statement(bs, "set", new Object[]
91                   { i, i + 1, true }));
92            }
93          });
94
95        out.writeObject(new Employee("Harry Hacker", 50000, 1989, 10, 1));
96        out.writeObject(new Point(17, 29));
97        out.writeObject(new java.awt.geom.Rectangle2D.Double(5, 10, 20, 30));
98        out.writeObject(InetAddress.getLocalHost());
99        BitSet bs = new BitSet();
100       bs.set(1, 4);
101       bs.clear(2, 3);
102       out.writeObject(bs);
103       out.close();
104   }
105 }
```

Исчерпывающий пример сохраняемости компонентов JavaBeans

В завершение обсуждаемой темы рассмотрим исчерпывающий пример сохраняемости компонентов JavaBeans (рис. 8.15). В данном примере создается прикладная программа, записывающая отчет о повреждениях арендованной автомашины. Агент по аренде автомобилей вводит запись об аренде, выбирает тип автомашины, отмечает мышью поврежденные части и сохраняет отчет. Эта программа позволяет также загрузить существующий отчет о повреждениях. Весь ее исходный код представлен в листинге 8.10.

Для сохранения и загрузки объектов типа `DamageReport` в рассматриваемой здесь прикладной программе используется сохраняемость компонентов JavaBeans (листинг 8.11). Данный пример иллюстрирует ряд следующих особенностей сохраняемости компонентов JavaBeans

- Свойства сохраняются и восстанавливаются автоматически. Для сохранения и восстановления свойств `rentalRecord` и `carType` не требуется принимать никаких специальных мер.
- Для восстановления данных о поврежденных частях автомашины необходимо выполнить ряд действий после конструирования объектов. С этой целью делегат сохраняемости формирует операторы, в которых вызывается метод `click()`.
- Для класса `Point2D.Double` требуется делегат типа `DefaultPersisetnce Delegate`, который создает точку, исходя из значений свойств `x` и `y`.
- Свойство `removeMode`, которое определяет, удаляется или добавляется отметка о повреждении, если щелкнуть на ней кнопкой мыши, является переходным, поскольку оно не должно сохраняться в отчете.

Ниже приведен пример отчета о повреждениях, сохраняемого в формате XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0" class="java.beans.XMLDecoder">
  <object class="DamageReport">
    <object class="java.lang.Enum" method="valueOf">
      <class>DamageReport$CarType</class>
      <string>SEDAN</string>
    </object>
```

```
    <void property="rentalRecord">
      <string>12443-19</string>
    </void>
    <void method="click">
      <object class="java.awt.geom.Point2D$Double">
        <double>181.0</double>
        <double>84.0</double>
      </object>
    </void>
    <void method="click">
      <object class="java.awt.geom.Point2D$Double">
        <double>162.0</double>
        <double>66.0</double>
      </object>
    </void>
  </object>
</java>
```

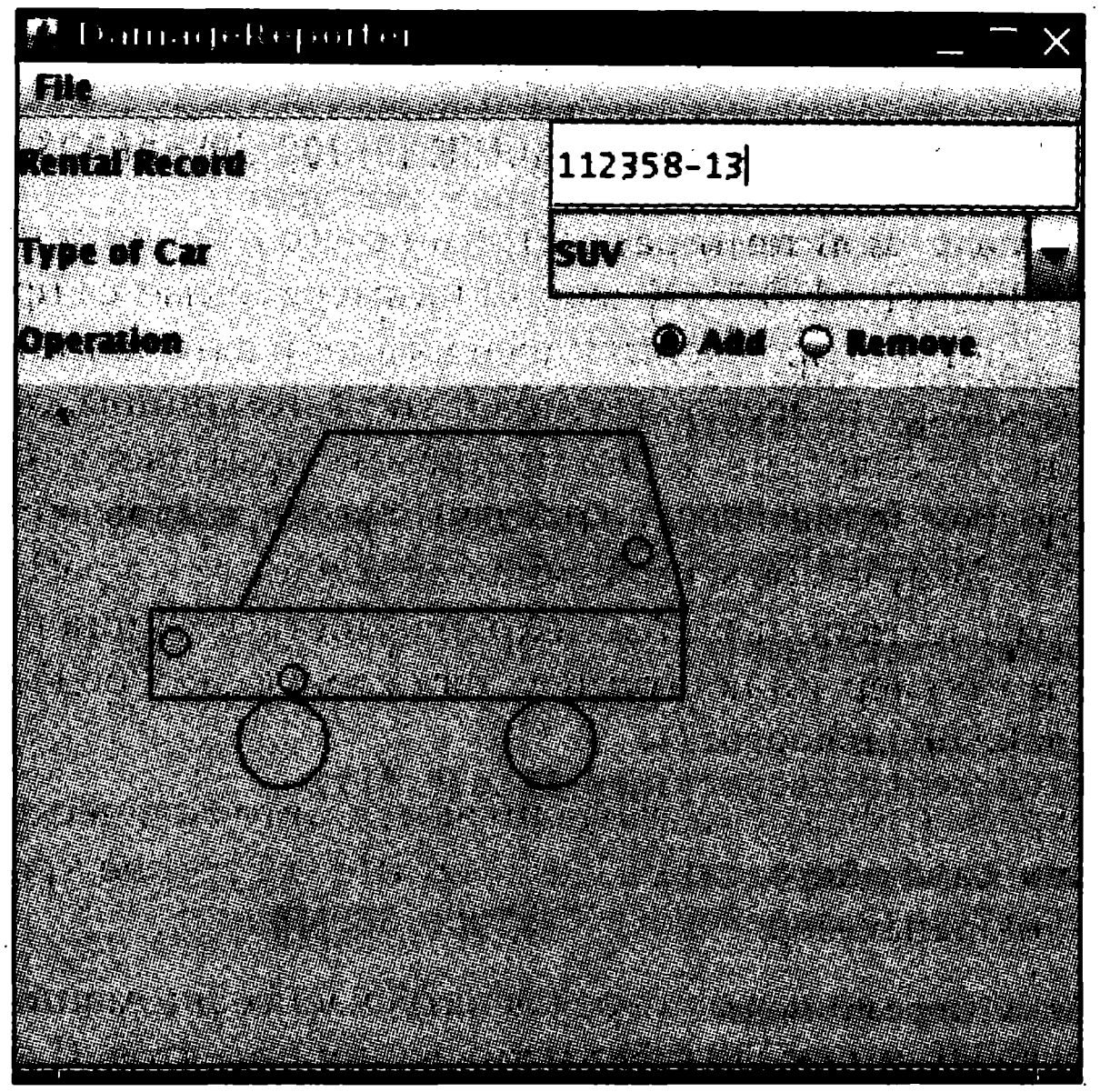


Рис. 8.15. Прикладная программа `DamageReporter`

НА ЗАМЕТКУ! В рассматриваемом здесь примере прикладной программы сохраняемость компонентов JavaBeans *не* используется для сохранения элементов ГПИ. Этот вопрос может представлять интерес для создателей ИСР. Здесь же основное внимание уделяется применению механизма сохраняемости для сохранения *прикладных данных*.

Данным примером завершается обсуждение сохраняемости компонентов JavaBeans. Ниже кратко подытожены характеристики архивов, создаваемых благодаря сохраняемости компонентов JavaBeans.

- Пригодны для долговременного хранения.
- Быстры и компактны.

- Просты в употреблении.
- Допускают редактирование вручную.
- Входят в состав стандартного набора средств Java.

Листинг 8.10. Исходный код из файла `damageReporter/DamageReporterFrame.java`

```
1   package damageReporter;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.awt.geom.*;
6   import java.beans.*;
7   import java.io.*;
8   import java.util.*;
9   import javax.swing.*;
10
11  public class DamageReporterFrame extends JFrame
12  {
13    private JTextField rentalRecord;
14    private JComboBox<DamageReport.CarType> carType;
15    private JComponent carComponent;
16    private JRadioButton addButton;
17    private JRadioButton removeButton;
18    private DamageReport report;
19    private JFileChooser chooser;
20
21    private static Map<DamageReport.CarType, Shape> shapes =
22       new EnumMap<>(DamageReport.CarType.class);
23
24    public DamageReporterFrame()
25    {
26      chooser = new JFileChooser();
27      chooser.setCurrentDirectory(new File("."));
28
29      report = new DamageReport();
30      report.setCarType(DamageReport.CarType.SEDAN);
31
32      // установить строку меню
33      JMenuBar menuBar = new JMenuBar();
34      setJMenuBar(menuBar);
35
36      JMenu menu = new JMenu("File");
37      menuBar.add(menu);
38
39      JMenuItem openItem = new JMenuItem("Open");
40      menu.add(openItem);
41      openItem.addActionListener(new ActionListener()
42        {
43          public void actionPerformed(ActionEvent evt)
44          {
45            // показать диалоговое окно для выбора файлов
46            int r = chooser.showOpenDialog(null);
47            // если файл выбран, открыть его
48            if (r == JFileChooser.APPROVE_OPTION)
49            {
50              try
51              {
```

```
52                File file = chooser.getSelectedFile();
53                XMLDecoder decoder = new XMLDecoder(
54                   new FileInputStream(file));
55                report = (DamageReport) decoder.readObject();
56                decoder.close();
57                rentalRecord.setText(report.getRentalRecord());
58                carType.setSelectedItem(report.getCarType());
59                repaint();
60             }
61             catch (IOException e)
62             {
63                JOptionPane.showMessageDialog(null, e);
64             }
65          }
66       }
67    });
68
69    JMenuItem saveItem = new JMenuItem("Save");
70    menu.add(saveItem);
71    saveItem.addActionListener(new ActionListener()
72       {
73          public void actionPerformed(ActionEvent evt)
74          {
75             report.setRentalRecord(rentalRecord.getText());
76             chooser.setSelectedFile(new File(rentalRecord.getText()
77                + ".xml"));
78
79             // показать диалоговое окно для выбора файлов
80             int r = chooser.showSaveDialog(null);
81
82             // если файл выбран, открыть его
83             if (r == JFileChooser.APPROVE_OPTION)
84             {
85                try
86                {
87                   File file = chooser.getSelectedFile();
88                   XMLEncoder encoder = new XMLEncoder(
89                      new FileOutputStream(file));
90                   report.configureEncoder(encoder);
91                   encoder.writeObject(report);
92                   encoder.close();
93                }
94                catch (IOException e)
95                {
96                   JOptionPane.showMessageDialog(null, e);
97                }
98             }
99          }
100      });
101
102   JMenuItem exitItem = new JMenuItem("Exit");
103   menu.add(exitItem);
104   exitItem.addActionListener(new ActionListener()
105      {
106         public void actionPerformed(ActionEvent event)
107         {
108            System.exit(0);
109         }
110      });
111
```

```
112     // комбинированный список для выбора типа автомашины
113     rentalRecord = new JTextField();
114     carType = new JComboBox<>();
115     carType.addItem(DamageReport.CarType.SEDAN);
116     carType.addItem(DamageReport.CarType.WAGON);
117     carType.addItem(DamageReport.CarType.SUV);
118
119     carType.addActionListener(new ActionListener()
120       {
121         public void actionPerformed(ActionEvent event)
122         {
122           DamageReport.CarType item = carType.getItemAt(
124             carType.getSelectedIndex());
125           report.setCarType(item);
126           repaint();
127         }
128       });
129
130     // компонент для показа формы автомашины и мест повреждения
131     carComponent = new JComponent()
132       {
133         private static final int DEFAULT_WIDTH = 400;
134         private static final int DEFAULT_HEIGHT = 200;
135
136         public void paintComponent(Graphics g)
137         {
138           Graphics2D g2 = (Graphics2D) g;
139           g2.setColor(new Color(0.9f, 0.9f, 0.45f));
140           g2.fillRect(0, 0, getWidth(), getHeight());
141           g2.setColor(Color.BLACK);
142           g2.draw(shapes.get(report.getCarType()));
143           report.drawDamage(g2);
144         }
145         public Dimension getPreferredSize()
146         {
147           return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
148         }
149       };
150     carComponent.addMouseListener(new MouseAdapter()
151       {
152         public void mousePressed(MouseEvent event)
153         {
154           report.click(new Point2D.Double(event.getX(), event.getY()));
155           repaint();
156         }
157       });
158
159     // кнопки-переключатели для ввода и удаления мест повреждений
160     addButton = new JRadioButton("Add");
161     removeButton = new JRadioButton("Remove");
162     ButtonGroup group = new ButtonGroup();
163     JPanel buttonPanel = new JPanel();
164     group.add(addButton);
165     buttonPanel.add(addButton);
166     group.add(removeButton);
167     buttonPanel.add(removeButton);
168     addButton.setSelected(!report.getRemoveMode());
169     removeButton.setSelected(report.getRemoveMode());
170     addButton.addActionListener(new ActionListener()
171       {
```

```
172         public void actionPerformed(ActionEvent event)
173         {
174           report.setRemoveMode(false);
175         }
176       });
177     removeButton.addActionListener(new ActionListener()
178       {
179         public void actionPerformed(ActionEvent event)
180         {
181           report.setRemoveMode(true);
182         }
183       });
184
185     // расположить компоненты на панели
186     JPanel gridPanel = new JPanel();
187     gridPanel.setLayout(new GridLayout(0, 2));
188     gridPanel.add(new JLabel("Rental Record"));
189     gridPanel.add(rentalRecord);
190     gridPanel.add(new JLabel("Type of Car"));
191     gridPanel.add(carType);
192     gridPanel.add(new JLabel("Operation"));
193     gridPanel.add(buttonPanel);
194
195     add(gridPanel, BorderLayout.NORTH);
196     add(carComponent, BorderLayout.CENTER);
197     pack();
198   }
199
200   static
201   {
202     int width = 200;
203     int x = 50;
204     int y = 50;
205     Rectangle2D.Double body = new Rectangle2D.Double(
206        x, y + width / 6, width - 1, width / 6);
207     Ellipse2D.Double frontTire = new Ellipse2D.Double(
208        x + width / 6, y + width / 3, width / 6, width / 6);
209     Ellipse2D.Double rearTire = new Ellipse2D.Double(
210        x + width * 2 / 3, y + width / 3, width / 6, width / 6);
211
212     Point2D.Double p1 = new Point2D.Double(x + width / 6, y + width / 6);
213     Point2D.Double p2 = new Point2D.Double(x + width / 3, y);
214     Point2D.Double p3 = new Point2D.Double(x + width * 2 / 3, y);
215     Point2D.Double p4 = new Point2D.Double(x + width * 5 / 6,
216        y + width / 6);
217
218     Line2D.Double frontWindshield = new Line2D.Double(p1, p2);
219     Line2D.Double roofTop = new Line2D.Double(p2, p3);
220     Line2D.Double rearWindshield = new Line2D.Double(p3, p4);
221
222     GeneralPath sedanPath = new GeneralPath();
223     sedanPath.append(frontTire, false);
224     sedanPath.append(rearTire, false);
225     sedanPath.append(body, false);
226     sedanPath.append(frontWindshield, false);
227     sedanPath.append(roofTop, false);
228     sedanPath.append(rearWindshield, false);
229     shapes.put(DamageReport.CarType.SEDAN, sedanPath);
230
231     Point2D.Double p5 = new Point2D.Double(x + width * 11 / 12, y);
```

```
232     Point2D.Double p6 = new Point2D.Double(x + width, y + width / 6);
233     roofTop = new Line2D.Double(p2, p5);
234     rearWindshield = new Line2D.Double(p5, p6);
235
236     GeneralPath wagonPath = new GeneralPath();
237     wagonPath.append(frontTire, false);
238     wagonPath.append(rearTire, false);
239     wagonPath.append(body, false);
240     wagonPath.append(frontWindshield, false);
241     wagonPath.append(roofTop, false);
242     wagonPath.append(rearWindshield, false);
243     shapes.put(DamageReport.CarType.WAGON, wagonPath);
244
245     Point2D.Double p7 = new Point2D.Double(x + width / 3, y - width / 6);
246     Point2D.Double p8 = new Point2D.Double(x + width * 11 / 12,
247        y - width / 6);
248     frontWindshield = new Line2D.Double(p1, p7);
249     roofTop = new Line2D.Double(p7, p8);
250     rearWindshield = new Line2D.Double(p8, p6);
251
252     GeneralPath suvPath = new GeneralPath();
253     suvPath.append(frontTire, false);
254     suvPath.append(rearTire, false);
255     suvPath.append(body, false);
256     suvPath.append(frontWindshield, false);
257     suvPath.append(roofTop, false);
258     suvPath.append(rearWindshield, false);
259     shapes.put(DamageReport.CarType.SUV, suvPath);
260   }
261 }
```

Листинг 8.11. Исходный код из файла damageReporter/DamageReport.java

```
1   package damageReporter;
2
3   import java.awt.*;
4   import java.awt.geom.*;
5   import java.beans.*;
6   import java.util.*;
7
8   /**
9    * Этот класс описывает отчет о повреждениях автомашины, сохраняемый
10   * и загружаемый с помощью механизма долговременной сохраняемости
11   * @version 1.22 2012-01-26
12   * @author Cay Horstmann
13  */
14  public class DamageReport
15  {
16    private String rentalRecord;
17    private CarType carType;
18    private boolean removeMode;
19    private java.util.List<Point2D> points = new ArrayList<>();
20
21    private static final int MARK_SIZE = 5;
22
23    public enum CarType
24    {
25      SEDAN, WAGON, SUV
26    }
```

```
27
28    // это свойство сохраняется автоматически
29    public void setRentalRecord(String newValue)
30    {
31       rentalRecord = newValue;
32    }
33
34    public String getRentalRecord()
35    {
36      return rentalRecord;
37    }
38
39    // это свойство сохраняется автоматически
40    public void setCarType(CarType newValue)
41    {
42      carType = newValue;
43    }
44
45    public CarType getCarType()
46    {
47      return carType;
48    }
49
50    // это свойство устанавливается как переходное
51    public void setRemoveMode(boolean newValue)
52    {
53      removeMode = newValue;
54    }
55
56    public boolean getRemoveMode()
57    {
58      return removeMode;
59    }
60
61    public void click(Point2D p)
62    {
63      if (removeMode)
64      {
65        for (Point2D center : points)
66        {
67          Ellipse2D circle = new Ellipse2D.Double(center.getX()
68             - MARK_SIZE, center.getY()
69             - MARK_SIZE, 2 * MARK_SIZE, 2 * MARK_SIZE);
70          if (circle.contains(p))
71          {
72            points.remove(center);
73            return;
74          }
75        }
76      }
77      else points.add(p);
78    }
79
80    public void drawDamage(Graphics2D g2)
81    {
82      g2.setPaint(Color.RED);
83      for (Point2D center : points)
84      {
85        Ellipse2D circle = new Ellipse2D.Double(center.getX()
86           - MARK_SIZE, center.getY()
```

```
87            - MARK_SIZE, 2 * MARK_SIZE, 2 * MARK_SIZE);
88          g2.draw(circle);
89       }
90    }
91
92    public void configureEncoder(XMLEncoder encoder)
93    {
94       // этот шаг требуется для сохранения объектов типа Point2D.Double
95       encoder.setPersistenceDelegate(Point2D.Double.class,
96          new DefaultPersistenceDelegate(new String[] { "x", "y" }));
97
98       // этот шаг требуется, так как списочный массив точек не доступен
99       // да и должен быть доступен как свойство
100      encoder.setPersistenceDelegate(DamageReport.class,
101         new DefaultPersistenceDelegate()
102      {
103         protected void initialize(Class<?> type, Object oldInstance,
104            Object newInstance, Encoder out)
105         {
106            super.initialize(type, oldInstance, newInstance, out);
107            DamageReport r = (DamageReport) oldInstance;
108
109            for (Point2D p : r.points)
110               out.writeStatement(new Statement(oldInstance, "click",
111                  new Object[] { p }));
112         }
113      });
114   }
115
116   // этот шаг требуется, чтобы сделать свойство removeMode переходным
117   static
118   {
119      try
120      {
121         BeanInfo info = Introspector.getBeanInfo(DamageReport.class);
122         for (PropertyDescriptor desc : info.getPropertyDescriptors())
123            if (desc.getName().equals("removeMode"))
124               desc.setValue("transient", Boolean.TRUE);
125      }
126      catch (IntrospectionException e)
127      {
128         e.printStackTrace();
129      }
130   }
131 }
```

java.beans.XMLEncoder 1.4

- `XMLEncoder(OutputStream out)`

  Создает объект типа `XMLEncoder`, направляющий выводимые данные в указанный поток вывода.

- `void writeObject(Object obj)`

  Записывает указанный объект в архив.

- `void writeStatement(Statement stat)`

  Записывает указанный оператор в архив. Этот метод вызывается только делегатом сохраняемости.

java.beans.Encoder 1.4

- `void setPersistenceDelegate(Class<?> type, PersistenceDelegate delegate)`
- `PersistenceDelegate getPersistenceDelegate(Class<?> type)`

  Устанавливают или получают делегат сохраняемости для архивирования объектов указанного типа.
- `void setExceptionListener(ExceptionListener listener)`
- `ExceptionListener getExceptionListener()`

  Устанавливают или получают приемник событий, который уведомляется о том, что в процессе кодирования возникло исключение.

java.beans.ExceptionListener 1.4

- `void exceptionThrown(Exception e)`

  Вызывается, если в процессе кодирования или декодирования возникло исключение.

java.beans.XMLDecoder 1.4

- `XMLDecoder(InputStream in)`

  Создает объект типа `XMLDecoder` для считывания архива из указанного потока ввода.
- `Object readObject()`

  Считывает очередной объект из архива.
- `void setExceptionListener(ExceptionListener listener)`
- `ExceptionListener getExceptionListener()`

  Устанавливают или получают приемник событий, который уведомляется о том, что в процессе кодирования возникло исключение.

java.beans.PersistenceDelegate 1.4

- `protected abstract Expression instantiate(Object oldInstance, Encoder out)`

  Возвращает выражение для получения экземпляра объекта, эквивалентного `oldInstance`.
- `protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)`

  Записывает в поток вывода `out` операторы, которые преобразуют объект `newInstance` в объект, эквивалентный `oldInstance`.

java.beans.DefaultPersistenceDelegate 1.4

- `DefaultPersistenceDelegate()`

  Создает делегат сохраняемости для класса с конструктором без аргументов.

- `DefaultPersistenceDelegate(String[] propertyNames)`

  Создает делегат сохраняемости для класса, параметры конструктора которого определяют значения указанных свойств.

- `protected Expression instantiate(Object oldInstance, Encoder out)`

  Возвращает выражение для вызова конструктора без аргументов или с параметрами, определяющими значения указанных свойств.

- `protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)`

  Записывает в поток вывода *out* операторы, которые применяют методы установки свойств к объекту *newInstance*, пытаясь сделать этот объект эквивалентным *oldInstance*.

java.beans.Expression 1.4

- `Expression(Object value, Object target, String methodName, Object[] parameters)`

  Создает выражение, которое вызывает указанный метод для целевого объекта `target` с заданными параметрами. Предполагается, что результатом выполнения выражения является значение `value`. Для вызова этого конструктора в качестве параметра `target` следует указать объект типа `Class`, а в качестве параметра `methodName` — символьную строку `"new"`.

java.beans.Statement 1.4

- `Statement(Object target, String methodName, Object[] parameters)`

  Создает оператор, который вызывает указанный метод для целевого объекта *target* с заданными параметрами.

На этом три длинных главы, посвященных программированию ГПИ средствами Swing, AWT и JavaBeans, завершаются. В следующей главе речь пойдет уже о совершенно другом предмете: безопасности. Безопасность всегда была отличительной особенностью платформы Java. В связи с постоянным увеличением степени риска в мире, где мы живем, трудимся и программируем, ясное представление о средствах безопасности Java становится все более актуальным для многих разработчиков.

ГЛАВА 9

Безопасность

В этой главе...

- Загрузчики классов
- Верификация байт-кода
- Диспетчеры защиты и полномочия
- Аутентификация пользователей
- Цифровые подписи
- Подписание кода
- Шифрование

С появлением технологии Java специалисты оценили по достоинству не только хорошо продуманные выразительные средства языка, но и механизмы обеспечения безопасности при выполнении аплетов, доставляемых через Интернет (более подробно аплеты рассматривались в главе 10 первого тома данной книги). Очевидно, что доставка исполняемых аплетов имеет смысл только тогда, когда получатели уверены, что код не может нанести ущерба работе их компьютеров. Поэтому вопросы безопасности были и остаются основной заботой как разработчиков, так и пользователей технологии Java. Это означает, что, в отличие от других языков и систем, где безопасность обеспечивалась в последнюю очередь, механизмы защиты изначально являются неотъемлемой частью технологии Java.

В технологии Java безопасность обеспечивают следующие три механизма.

- Структурные функциональные возможности языка (например, проверка границ массивов, запрет на преобразования непроверенных типов, отсутствие указателей и т.д.).
- Средства контроля доступа, определяющие действия, которые разрешается или запрещается выполнять в коде (например, может ли код получать доступ к файлам, передавать данные по сети и т.д.).

- Механизм цифровой подписи, предоставляющий авторам возможность применять стандартные алгоритмы для аутентификации своих программ, а пользователям — точно определять, кто создал код и изменился ли он с момента его подписания.

В этой главе сначала рассматриваются *загрузчики классов*, проверяющие файлы классов на предмет целостности при их загрузке в виртуальную машину Java. А затем будет показано, каким образом этот механизм может выявлять в файлах классов признаки злонамеренных действий.

Для обеспечения максимальной безопасности оба механизма загрузки классов (используемый по умолчанию и специальный) должны взаимодействовать с классом *диспетчера защиты*, определяющим действия, которые разрешено или запрещено выполнять в коде. Поэтому далее в этой главе подробно поясняется, каким образом настраивается безопасность на платформе Java.

И в завершение главы будут рассмотрены различные алгоритмы шифрования, доступные в пакете `java.security` и позволяющие подписывать код и обеспечивать аутентификацию пользователей. Как обычно, основное внимание уделяется только тем вопросам, которые представляют наибольший интерес для разработчиков прикладных программ на Java. Тем, кому требуется более углубленное изучение данной темы, рекомендуем книгу *Inside Java 2 Platform Security: Architecture, API Design, and Implementation, 2nd edition* Ли Гонга, Гэри Эллисона и Мэри Дейджфорд (Li Gong, Gary Ellison, Mary Dageforde; издательство Prentice Hall PTR, 2003 г.).

Загрузчики классов

Компилятор Java преобразует исходные операторы языка в понятный для виртуальной машины Java байт-код, который сохраняется в файле класса с расширением `.class`. В каждом файле класса содержится код определения и реализации только для одного класса или интерфейса. Далее все эти файлы классов интерпретируются программой, которая способна преобразовывать инструкции виртуальной машины в машинные команды целевой платформы.

Следует иметь в виду, что интерпретатор виртуальной машины загружает только те файлы классов, которые требуются для выполнения программы в данный момент. Допустим, выполнение программы начинается с файла `MyProgram.class`. Ниже описаны действия, которые выполняет виртуальная машина Java.

1. В виртуальной машине имеется механизм загрузки файлов классов, например, путем их считывания с диска или загрузки из Интернета. С помощью этого механизма загружается содержимое файла класса `MyProgram`.
2. Если в классе `MyProgram` встречаются поля или объекты, ссылающиеся на классы других типов, то дополнительно загружаются файлы этих классов. (Процесс загрузки всех классов, от которых зависит данный класс, называется *разрешением класса*.)
3. Далее виртуальная машина выполняет метод `main()` из класса `MyProgram`. (Этот метод является статическим, поэтому никаких экземпляров класса `MyProgram` создавать не требуется.)
4. Если для выполнения метода `main()` или вызываемого из него метода требуются дополнительные классы, то они загружаются далее из соответствующих файлов.

Но в механизме загрузки классов используется не один, а несколько загрузчиков. Каждой программе на Java сопутствует, по меньшей мере, три загрузчика классов.

- Загрузчик базовых классов.
- Загрузчик расширений классов.
- Загрузчик системных классов, иногда называемый загрузчиком прикладных классов.

Загрузчик базовых классов загружает базовые или системные классы, как правило, из файла `rt.jar`. Он является неотъемлемой частью виртуальной машины и обычно реализуется на языке C. Этому загрузчику классов не соответствует ни один из объектов типа `ClassLoader`. Например, при вызове следующего метода возвращается пустое значение `null`:

```
String.class.getClassLoader()
```

Загрузчик расширений классов загружает стандартные расширения из каталога *jre*`/lib/ext`. Если переместить файлы с расширением `.jar` в этот каталог, то загрузчик расширений классов найдет в нем нужные классы расширений даже без указания пути к ним. (Некоторые рекомендуют применять этот механизм, чтобы избежать путаницы при указании пути к классам, но такой способ связан с определенными сложностями, поясняемыми в приведенной ниже врезке "Внимание".)

Загрузчик системных классов загружает системные или прикладные классы, которые размещаются в каталогах и архивных файлах формата JAR/ZIP. Путь к классам должен быть указан в переменной окружения `CLASSPATH` или параметре командной строки **`-classpath`**.

В версии платформы Java, внедренной компанией Oracle, загрузчики расширений и системных классов реализуются на Java. Оба загрузчика представляют собой экземпляры класса `URLClassLoader`.

ВНИМАНИЕ! Если требуется извлечь из архива класс, который, в свою очередь, должен загружать класс, не являющийся системным или расширением, то размещать архивные JAR-файлы в каталоге *jre*`/lib/ext` нежелательно. Ведь загрузчик расширений классов не использует сведения о пути к классам, устанавливаемые в переменной окружения `CLASSPATH`. Это обстоятельство следует иметь в виду, принимая решение об использовании каталога *jre*`/lib/ext` для хранения своих классов.

НА ЗАМЕТКУ! Помимо всех перечисленных выше мест, классы могут загружаться из каталога *jre*`/lib/endorsed`. Такой механизм может применяться только для замены определенных стандартных библиотек Java (вроде тех, что предназначены для поддержки XML и CORBA) более новыми версиями. Подробнее об этом можно узнать по адресу `http://docs.oracle.com/javase/7/docs/technotes/guides/standards`.

Иерархия загрузчиков классов

Загрузчики классов связаны отношениями "родитель–потомок". У каждого загрузчика классов, за исключением базовых классов, имеется свой родительский загрузчик классов. Предполагается, что загрузчик классов дает возможность своему родителю загружать любой нужный класс и загружает его сам только в том случае, если этого не может сделать родитель. Так, если загрузчик системных классов получает запрос

на загрузку системного класса (например, `java.util.arrayList`), то он сначала предлагает загрузить его загрузчику расширений классов. А загрузчик расширений классов, в свою очередь, предлагает сделать это загрузчику базовых классов. В итоге загрузчик базовых классов находит и загружает класс из архивного файла `rt.jar`, а все остальные загрузчики классов прекращают дальнейший поиск.

Некоторые программы имеют модульную архитектуру, где некоторые части кода упаковываются в дополнительные, но необязательно подключаемые модули. Если подключаемые модули упаковываются в архивные JAR-файлы, то классы этих модулей могут просто загружаться с помощью экземпляра класса `URLClassLoader`, как показано ниже.

```
URL url = new URL("file:///path/to/plugin.jar");
URLClassLoader pluginLoader = new URLClassLoader(new URL[] { url });
Class<?> cl = pluginLoader.loadClass("mypackage.MyClass");
```

В конструкторе класса `URLClassLoader` вообще не указан родитель загрузчика подключаемых модулей типа `pluginLoader`, поэтому в роли его родителя будет выступать загрузчик системных классов. Схематическое представление иерархии загрузчиков классов приведено на рис. 9.1.

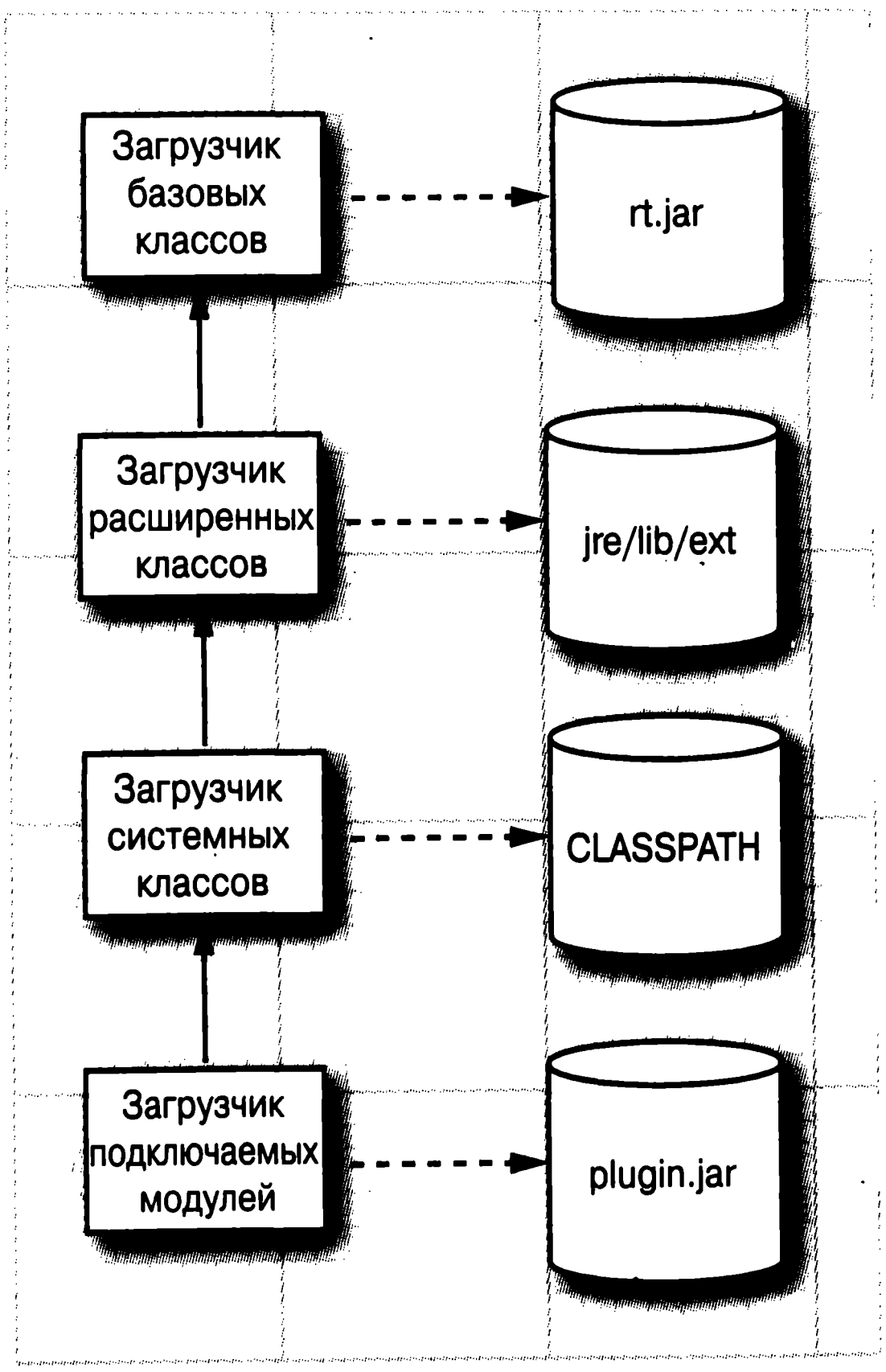


Рис. 9.1. *Иерархия загрузчиков классов*

Как правило, разработчиков прикладных программ мало интересует иерархия загрузчиков классов. Обычно одни классы загружаются потому, что этого требуют другие классы, и этот процесс является абсолютно прозрачным для разработчика. Но иногда разработчикам прикладных программ все-таки приходится вмешиваться в этот процесс, чтобы указывать нужный загрузчик классов. Рассмотрим следующий пример.

- В коде прикладной программы имеется вспомогательный метод, вызывающий метод `Class.forName(classNameString)`.
- Этот метод вызывается из класса подключаемого модуля.
- Параметр `classNameString` указывает на класс, содержащийся в архивном файле `plugin.jar`.

Автор подключаемого модуля имеет все основания предполагать, что его класс будет загружаться. Но загрузка класса вспомогательного метода выполнялась загрузчиком системных классов, т.е. именно тем загрузчиком, который использовался методом `Class.forName()`. Это означает, что классы, находящиеся в архивном файле `plugin.jar`, недоступны. Подобное явление называется *инверсией загрузчиков классов*.

Для устранения данного препятствия требуется, чтобы вспомогательный метод использовал нужный загрузчик классов. Для этого можно указать нужный загрузчик классов в виде параметра или установить его в качестве загрузчика контекста классов в текущем потоке исполнения. Вторая методика применяется во многих прикладных интерфейсах API (например, JAXP и JNDI, упоминавшихся в главах 2 и 4).

В каждом потоке исполнения имеется ссылка на загрузчик классов, называемый *загрузчиком контекста классов*. Таким загрузчиком для главного потока исполнения является загрузчик системных классов. При создании нового потока исполнения для него назначается загрузчик контекста классов из того потока исполнения, который его создает. Следовательно, если не вмешиваться в этот процесс, то для всех потоков исполнения в качестве загрузчика контекста классов назначается загрузчик системных классов. Тем не менее существует возможность указать любой другой загрузчик классов, сделав следующий вызов:

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

Вспомогательный метод может затем извлечь загрузчик контекста классов следующим образом:

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class cl = loader.loadClass(className);
```

Теперь остается разрешить следующий вопрос: когда в качестве этого загрузчика контекста классов должен быть назначен загрузчик класса подключаемого модуля? Решение зависит уже от самого разработчика прикладной программы. Обычно лучше сделать так, чтобы загрузчик контекста классов устанавливался при вызове метода из класса подключаемого модуля, который был загружен с помощью другого загрузчика классов. Но, с другой стороны, можно сделать и так, чтобы загрузчик контекста классов устанавливался в коде, вызывающем вспомогательный метод.

СОВЕТ. При написании метода, загружающего класс по имени, рекомендуется предоставлять вызывающему его коду возможность выбирать между передачей явно задаваемого загрузчика классов и применением загрузчика контекста классов. А просто использовать загрузчик класса этого мето-

Применение загрузчиков классов в качестве пространств имен

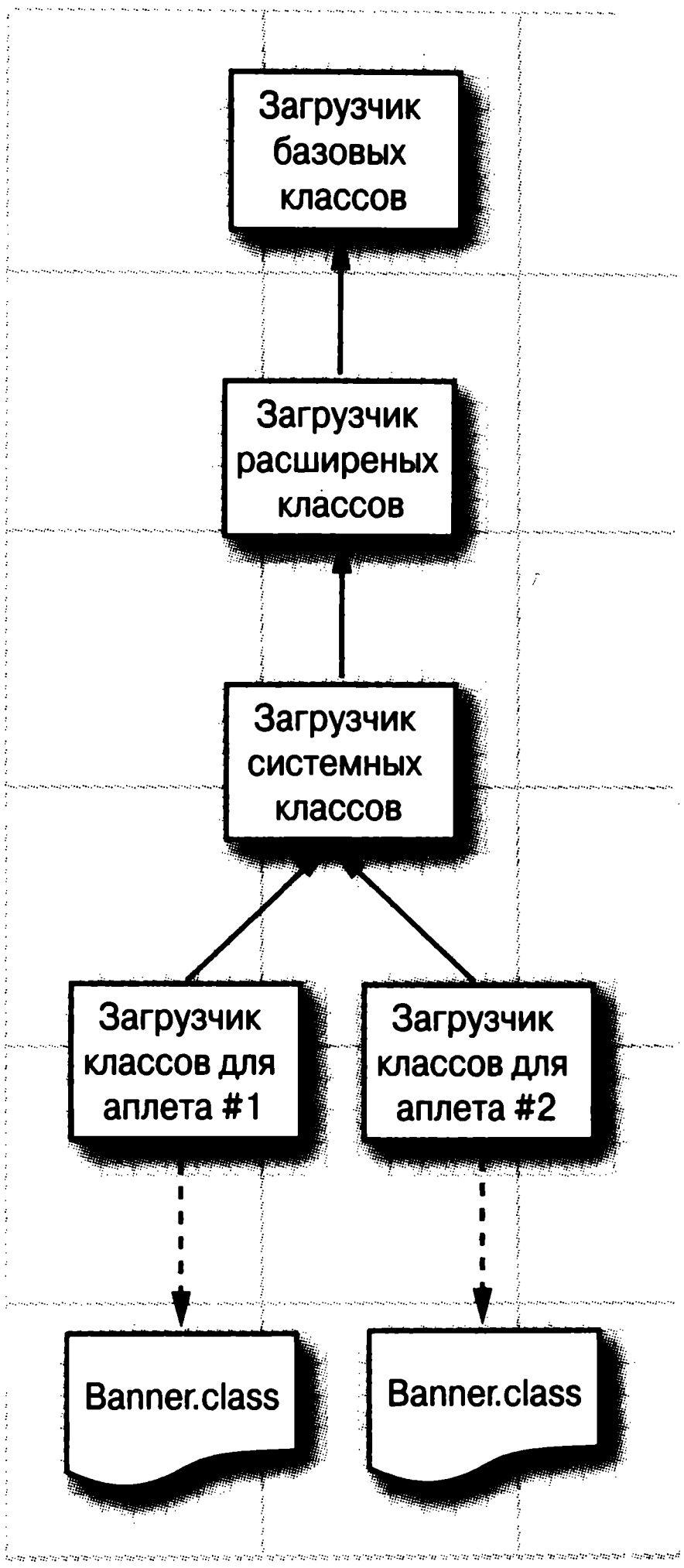


Рис. 9.2. Два загрузчика классов загружают разные классы аплетов с одинаковыми именами

Всякому программирующему на Java известно, что для устранения конфликтов имен применяются имена пакетов. Например, в стандартной библиотеке Java имеются два класса под именем `Date`, но полностью они именуются как `java.util.Date` и `java.sql.Date`. Короткие имена употребляются для удобства программирования, но требуют включения соответствующих операторов `import` в исходный код. А в выполняющейся программе все имена классов непременно содержат имена своих пакетов.

Как ни странно, но в пределах одной и той же виртуальной машины могут существовать два разных класса, имеющих одинаковое имя класса и пакета. Дело в том, что класс определяется по его полному имени и загрузчику класса. Такая технология очень удобна при загрузке кода из нескольких источников. Например, в веб-браузере для загрузки каждой веб-страницы используются отдельные экземпляры загрузчика классов аплетов. Это позволяет виртуальной машине различать классы разных веб-страниц независимо от их имен. Допустим, веб-страница содержит два аплета, предоставляемых двумя разными рекламодателями, и у каждого из этих аплетов имеется класс `Banner`. Поскольку каждый аплет загружается отдельным загрузчиком классов, эти классы являются совершенно отдельными и не конфликтуют друг с другом (рис. 9.2).

НА ЗАМЕТКУ! Существуют и другие способы использования данной технологии, например, "горячее развертывание" сервлетов и компонентов Enterprise JavaBeans. Подробнее об этом можно узнать по адресу `http://zeroturnaround.com/labs/rjc301`.

Создание собственного загрузчика классов

Для специальных целей можно создавать и свои собственные загрузчики классов. Такой подход позволяет выполнять перед передачей байт-кода виртуальной машине какие-нибудь специальные проверки. Например, можно создать загрузчик классов,

способный запрещать загрузку класса, который не был помечен как "оплаченный". Для того чтобы создать собственный загрузчик классов, достаточно расширить класс `ClassLoader` и переопределить метод `findClass(String className)`.

Метод `loadClass()` из суперкласса `ClassLoader` берет на себя все хлопоты по делегированию функций загрузки родительскому загрузчику классов и вызывает метод `findClass()` только в том случае, если класс еще не был загружен и если родительскому загрузчику классов не удалось его загрузить. Поэтому при самостоятельной реализации метода `findClass()` необходимо сделать так, чтобы он выполнял следующее.

1. Загружал байт-код класса из локальной файловой системы или какого-нибудь другого источника.
2. Вызывал метод `defineClass()` из суперкласса `ClassLoader` для представления байт-кода виртуальной машине.

В примере программы из листинга 9.1 реализуется загрузчик классов, способный загружать файлы зашифрованных классов. Эта программа сначала запрашивает у пользователя имя первого загружаемого класса (т.е. класса, содержащего метод `main()`) и ключ расшифровки. Затем она использует специальный загрузчик классов для загрузки указанного класса и вызова метода `main()`. Этот специальный загрузчик расшифровывает указанный класс и все несистемные классы, на которые он ссылается, после чего программа наконец-то вызывает метод `main()` из загруженного класса, как показано на рис. 9.3. Ради простоты для шифрования файлов классов в рассматриваемом здесь примере используется древний шифр Юлия Цезаря, пренебрегая более чем 2000-летним опытом в области криптографии.

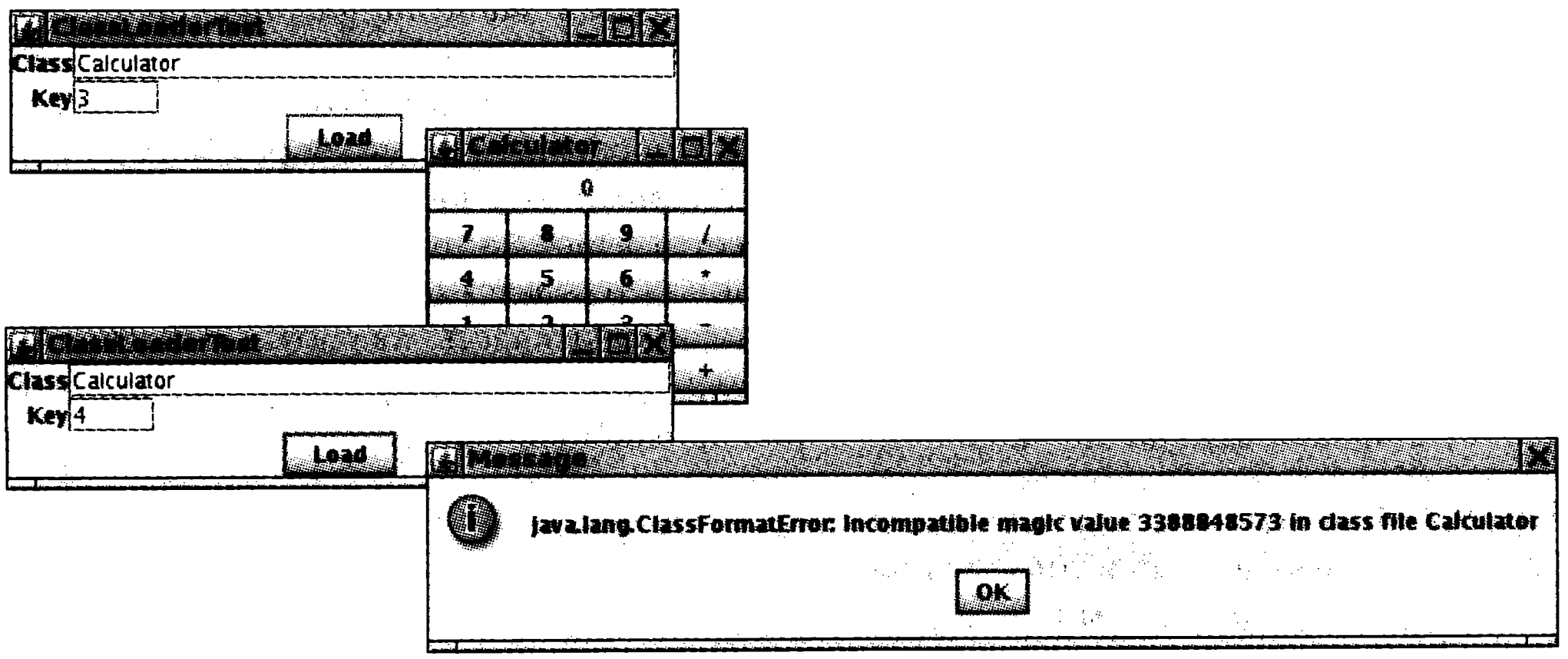


Рис. 9.3. Программа `ClassLoaderTest`

НА ЗАМЕТКУ! В прекрасной книге *The Codebreakers* Дэвида Кана (David Kahn; издательство Macmillan, 1967 г.; в русском переводе эта книга вышла под названием *Взломщики кодов* в издательстве "Центрполиграф", 2000 г.) говорится, что Юлий Цезарь кодировал 24 буквы латинского алфавита, сдвигая их на 3 буквы, чем повергал в недоумение всех своих противников. На момент написания первоначального варианта этой главы правительство США ограничивало экспорт наиболее сложных методов шифрования. Поэтому использование довольно простого метода шифрования Юлия Цезаря в рассматриваемом здесь примере никак не нарушит экспортных ограничений.

В рассматриваемом здесь варианте тайнописи Юлия Цезаря в качестве ключа используется число в пределах от **1** до **255**. Для шифрования байт просто суммируется с этим ключом по модулю **256**. Алгоритм шифрования реализован в программе из файла `Caesar.java`, исходный код которой приведен в листинге 9.2. Чтобы не поставить в тупик стандартный загрузчик классов, для зашифрованных файлов классов используется расширение `.caesar`.

При расшифровке загрузчик классов вычитает значение ключа из каждого байта. В загружаемом коде к данной книге предлагаются четыре файла классов, зашифрованных с помощью традиционного значения **3** ключа шифрования. Эти классы нельзя загрузить и запустить на выполнение стандартным интерпретатором байт-кода. Для этого понадобится специальный загрузчик классов, реализованный в рассматриваемой здесь программе `ClassLoaderTest`.

Зашифрованные файлы классов имеют самое разное практическое применение (если, конечно, используется более сложный шифр, чем тайнопись Юлия Цезаря). Без ключа шифрования эти файлы классов бесполезны, поскольку они не могут выполняться в стандартной виртуальной машине, а восстановить их исходный код нелегко.

Это означает, что для аутентификации пользователя класса или для проверки того, что программа оплачена до запуска, можно воспользоваться специальным загрузчиком классов. Естественно, что шифрование является только одним из возможных применений специального загрузчика классов. Для решения других задач, например, сохранения файлов классов в базе данных, можно создавать и применять иные разновидности загрузчиков классов.

Листинг 9.1. Исходный код из файла `classLoader/ClassLoaderTest.java`

```
1   package classLoader;
2
3   import java.io.*;
4   import java.lang.reflect.*;
5   import java.nio.file.*;
6   import java.awt.*;
7   import java.awt.event.*;
8   import javax.swing.*;
9   /**
10   * В этой программе демонстрируется специальный загрузчик классов,
11   * расшифровывающий файлы классов
12   * @version 1.23 2012-06-08
13   * @author Cay Horstmann
14  */
15  public class ClassLoaderTest
16  {
17    public static void main(String[] args)
18    {
19      EventQueue.invokeLater(new Runnable()
20        {
21          public void run()
22          {
23            JFrame frame = new ClassLoaderFrame();
24            frame.setTitle("ClassLoaderTest");
25            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26            frame.setVisible(true);
27          }
28        });
```

```
29    }
30 }
31
32 /**
33  * Этот фрейм содержит два текстовых поля для ввода имени
34  * загружаемого класса и ключ расшифровки
35 */
36 class ClassLoaderFrame extends JFrame
37 {
38   private JTextField keyField = new JTextField("3", 4);
39   private JTextField nameField = new JTextField("Calculator", 30);
40   private static final int DEFAULT_WIDTH = 300;
41   private static final int DEFAULT_HEIGHT = 200;
42
43   public ClassLoaderFrame()
44   {
45     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
46     setLayout(new GridBagLayout());
47     add(new JLabel("Class"), new GBC(0, 0).setAnchor(GBC.EAST));
48     add(nameField, new GBC(1, 0).setWeight(100, 0).setAnchor(GBC.WEST));
49     add(new JLabel("Key"), new GBC(0, 1).setAnchor(GBC.EAST));
50     add(keyField, new GBC(1, 1).setWeight(100, 0).setAnchor(GBC.WEST));
51     JButton loadButton = new JButton("Load");
52     add(loadButton, new GBC(0, 2, 2, 1));
53     loadButton.addActionListener(new ActionListener()
54       {
55         public void actionPerformed(ActionEvent event)
56         {
57           runClass(nameField.getText(), keyField.getText());
58         }
59       });
60     pack();
61   }
62
63   /**
64    * Выполняет основной метод указанного класса
65    * @param name Имя класса
66    * @param key Ключ расшифровки файлов классов
67   */
68   public void runClass(String name, String key)
69   {
70     try
71     {
72       ClassLoader loader = new CryptoClassLoader(Integer.parseInt(key));
73       Class<?> c = loader.loadClass(name);
74       Method m = c.getMethod("main", String[].class);
75       m.invoke(null, (Object) new String[] {});
76     }
77     catch (Throwable e)
78     {
79       JOptionPane.showMessageDialog(this, e);
80     }
81   }
82
83 }
84
85 /**
86  * Этот загрузчик клаосов загружает их из зашифрованных файлов
```

```
87  */
88  class CryptoClassLoader extends ClassLoader
89  {
90    private int key;
91
92    /**
93     * Конструирует загрузчик зашифрованных классов
94     * @param k Ключ расшифровки
95    */
96    public CryptoClassLoader(int k)
97    {
98      key = k;
99    }
100
101   protected Class<?> findClass(String name) throws ClassNotFoundException
102   {
103     try
104     {
105       byte[] classBytes = null;
106       classBytes = loadClassBytes(name);
107       Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
108       if (cl == null) throw new ClassNotFoundException(name);
109       return cl;
110     }
111     catch (IOException e)
112     {
113       throw new ClassNotFoundException(name);
114     }
115   }
116
117   /**
118    * Загружает и расшифровывает байты из файла класса
119     * @param name Имя класса
120     * @return Возвращает массив с байтами из файла класса
121   */
122   private byte[] loadClassBytes(String name) throws IOException
123   {
124     String cname = name.replace('.', '/') + ".caesar";
125     byte[] bytes = Files.readAllBytes(Paths.get(cname));
126     for (int i = 0; i < bytes.length; i++)
127       bytes[i] = (byte) (bytes[i] - key);
128     return bytes;
129   }
130 }
```

Листинг 9.2. Исходный код из файла `classLoader/Caesar.java`

```
1  package classLoader;
2
3  import java.io.*;
4
5  /**
6   * Шифрует файл, используя тайнопись Юлия Цезаря
7   * @version 1.01 2012-06-10
8   * @author Cay Horstmann
9  */
```

```
10 public class Caesar
11 {
12   public static void main(String[] args) throws Exception
13   {
14     if (args.length != 3)
15     {
16       System.out.println("USAGE: java classLoader.Caesar in out key");
17       return;
18     }
19
20     try(FileInputStream in = new FileInputStream(args[0]);
21        FileOutputStream out = new FileOutputStream(args[1]))
22     {
23       int key = Integer.parseInt(args[2]);
24       int ch;
25       while ((ch = in.read()) != -1)
26       {
27         byte c = (byte) (ch + key);
28         out.write(c);
29       }
30     }
31   }
32 }
```

java.lang.Class 1.0

- `ClassLoader getClassLoader()`

  Получает загрузчик для загрузки данного класса.

java.lang.ClassLoader 1.0

- `ClassLoader getParent()` **1.2**

  Получает родительский загрузчик классов или пустое значение `null`, если это загрузчик базовых классов.

- `static ClassLoader getSystemClassLoader()` **1.2**

  Получает загрузчик системных классов, т.е. тот, который применялся для загрузки первого прикладного класса.

- `protected Class findClass(String` *name*`)` **1.2**

  Должен быть переопределен в загрузчике классов для поиска байт-кода класса и его представления виртуальной машине благодаря вызову метода `defineClass()`. Для отделения имени пакета от имени класса следует использовать точку и не указывать суффикс **`.class`**.

- `Class defineClass(String` *name*`, byte[]` *byteCodeData*`, int` *offset*`, int` *length*`)`

  Передает виртуальной машине новый класс, предоставляя байт-код в определенном диапазоне данных.

java.net.URLClassLoader 1.2

- `URLClassLoader(URL[] urls)`
- `URLClassLoader(URL[] urls, ClassLoader parent)`

  Создает загрузчик классов по указанному URL. Если URL оканчивается знаком /, то подразумевается каталог, а иначе — архивный JAR-файл.

java.lang.Thread 1.0

- `ClassLoader getContextClassLoader()` **1.2**

  Получает загрузчик классов, обозначенный создателем данного потока исполнения как наиболее приемлемый для использования в этом потоке.

- `void setContextClassLoader(ClassLoader loader)` **1.2**

  Устанавливает загрузчик классов для кода в данном потоке исполнения. Если при запуске потока на исполнение загрузчик контекста классов не задан явно, то используется родительский загрузчик контекста классов.

Верификация байт-кода

Когда загрузчик классов представляет виртуальной машине байт-код класса, этот код сначала обследуется *верификатором*. Верификатор проверяет все классы за исключением системных и определяет те команды, которые могут нанести ущерб виртуальной машине.

Ниже приведены некоторые виды проверок, выполняемых верификатором.

- Инициализация переменных перед их использованием.
- Согласование типов ссылок при вызове метода.
- Соблюдение правил доступа к закрытым данным и методам.
- Доступ к локальным переменным в стеке во время выполнения.
- Отсутствие переполнения стека.

При невыполнении какой-нибудь из этих проверок класс считается поврежденным и загружаться не будет.

НА ЗАМЕТКУ! Теорема Гёделя утверждает, что невозможно разработать алгоритмы, способные обрабатывать программные файлы и принимать решение, обладают ли программы ввода данных конкретным свойством (например, независимостью от переполнения стека). В таком случае возникает вопрос: каким образом верификатор может определить, что в файле класса не соблюдается соответствие типов, используются неинициализированные переменные и переполняется стек? Нет ли здесь противоречия законам логики? Никакого противоречия нет, потому что верификатор *не* является алгоритмом принятия решений в смысле теоремы Гёделя. Если верификатор принимает программу, значит, она действительно безопасна. Но он может и отклонять команды виртуальной машины, даже если те выглядят вполне безопасными. (Вам, вероятно, уже доводилось сталкиваться с подобным затруднением и инициализировать переменную каким-нибудь фиктивным значением из-за того, что компилятор не в состоянии гарантировать ее должную инициализацию.)

Такая строгая верификация требуется по соображениям безопасности. Случайные ошибки, связанные с использованием неинициализированных переменных, могут легко нанести значительный ущерб программе, если не определить их вовремя. Еще важнее то обстоятельство, что при повсеместном доступе к Интернету необходимо обеспечить надежную защиту от злонамеренных попыток недобросовестных программистов, стремящихся проникнуть в чужую систему или нанести ей ущерб. Например, видоизменив значения в стеке выполняемой программы или в закрытых полях системных объектов, любая программа способна нарушить систему защиты веб-браузера.

Зачем же создавать специальный верификатор для проверки всех этих условий? Ведь компилятор все равно не позволит сгенерировать файл класса, в котором предполагается использовать неинициализированную переменную или область закрытых данных, доступную из другого класса. В самом деле, файл класса, сформированный компилятором Java, всегда проходит верификацию. Но используемый в этих файлах формат байт-кода хорошо документирован, и всякий, имеющий опыт программирования на ассемблере и работы в шестнадцатеричном редакторе, может без труда создать вручную файл класса с достоверными, но потенциально опасными командами для виртуальной машины Java. Таким образом, верификатор блокирует злонамеренно измененные файлы классов, а не только проверяет созданные компилятором файлы.

В листинге 9.3 приведен пример программы `VerifierTest.java`, демонстрирующий изменение файла класса вручную. В этой простой программе вызывается метод `fun()` и выводится результат его выполнения. Она может быть запущена из командной строки или в виде аплета. Метод `fun()` выполняет сложение чисел **1** и **2**, как показано ниже.

```
static int fun()
{
  int m;
  int n;
  m = 1;
  n = 2;
  int r = m + n;
  return r;
}
```

Листинг 9.3. Исходный код из файла `verifier/VerifierTest.java`

```
1  package verifier;
2
3  import java.applet.*;
4  import java.awt.*;
5  /**
6   * В этой прикладной программе демонстрируется верификатор байт-кода
7   * виртуальной машины Java. Если воспользоваться шестнадцатеричным
8   * редактором для видоизменения файла класса, то виртуальная машина
9   * должна обнаружить злонамеренное искажение содержимого файла класса
10  * @version 1.00 1997-09-10
11  * @author Cay Horstmann
12 */
13 public class VerifierTest extends Applet
14 {
15   public static void main(String[] args)
16   {
17     System.out.println("1 + 2 == " + fun());
18   }
```

```
19
20   /**
21     * функция, вычисляющая сумму чисел 1 + 2
22     * @return Возвращает сумму 3, если код не нарушен
23   */
24   public static int fun()
25   {
26     int m;
27     int n;
28     m = 1;
29     n = 2;
30     // воспользоваться шестнадцатеричным редактором, чтобы
31     // изменить значение переменной m на 2 в файле класса
32     int r = m + n;
33     return r;
34   }
35
36   public void paint(Graphics g)
37   {
38     g.drawString("1 + 2 == " + fun(), 20, 20);
39   }
40 }
```

В качестве эксперимента попробуйте скомпилировать приведенный ниже видоизмененный вариант метода `fun()`.

```
static int fun()
{
  int m = 1;
  int n;
  m = 1;
  m = 2;
  int r = m + n;
  return r;
}
```

В данном случае переменная **n** не инициализирована и может иметь произвольное значение. Естественно, что компилятор обнаружит эту ошибку и откажется компилировать программу. Для создания недопустимого файла класса запустите утилиту **javap**, чтобы выяснить, как компилятор транслирует метод `fun()`:

```
javap -c verifier.VerifierTest
```

Эта утилита показывает байт-код из файла класса в мнемоническом виде:

```
Method int fun()
   0 iconst_1
   1 istore_0
   2 iconst_2
   3 istore_1
   4 iload_0
   5 iload_1
   6 iadd
   7 istore_2
   8 iload_2
   9 ireturn
```

Для изменения команды **3**, т.е. замены `istore_1` на `istore_0`, воспользуйтесь шестнадцатеричным редактором. Таким образом, локальная переменная **0** (т.е. **m**) будет инициализирована дважды, а локальная переменная **1** (т.е. **n**) вообще не будет

инициализирована. Чтобы сделать это, необходимо знать шестнадцатеричные значения команд. Эти значения приводятся ниже и взяты из книги *The Java Virtual Machine Specification* Тима Линдхольма и Фрэнка Йеллина (Tim Lindholm & Frank Yellin; издательство Prentice Hall PTR, 1999 г.).

```
0 iconst_1 04
1 istore_0 3B
2 iconst_2 05
3 istore_1 3C
4 iload_0 1A
5 iload_1 1B
6 iadd 60
7 istore_2 3D
8 iload_2 1C
9 ireturn AC
```

Отредактировать любое их этих значений можно в шестнадцатеричном редакторе. На рис. 9.4 приведено рабочее окно шестнадцатеричного редактора Gnome с загруженным файлом `VerifierTest.class`, в котором светло-серым выделены байт-коды метода `fun()`.

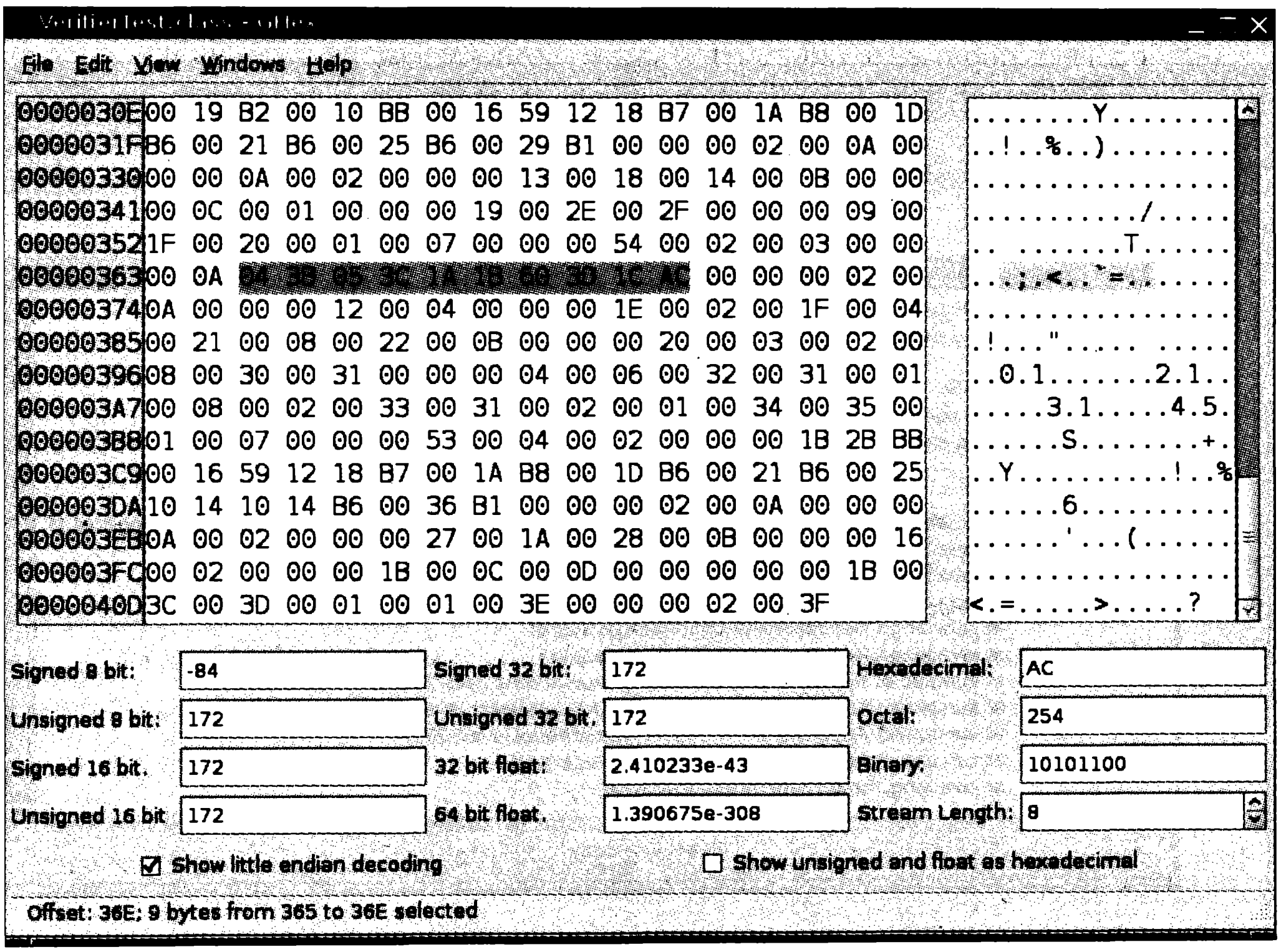


Рис. 9.4. Видоизменение байт-кода в шестнадцатеричном редакторе

Замените шестнадцатеричное значение **3C** на **3B**, сохраните файл, а затем попытайтесь запустить программу `VerifierTest` на выполнение. В итоге вы получите следующее сообщение об ошибке:

```
Exception in thread "main" java.lang.VerifyError: (class: VerifierTest,
    method:fun signature: ()I) Accessing value from uninitialized register 1
```

Таким образом, виртуальная машина обнаружила недопустимое видоизменение байт-кода. А теперь запустите программу с параметром **-noverify** (или **-Xverify:none**), т.е. без верификации, как показано ниже.

```
java -noverify verifier.VerifierTest
```

Метод `fun()` возвращает случайную сумму, получаемую в результате сложения числа **2** с произвольным значением, хранящимся в переменной **n**, которая не была инициализирована. Ниже приведен типичный результат выполнения рассматриваемой здесь программы без верификации байт-кода.

```
1 + 2 == 15102330
```

Для того чтобы продемонстрировать, каким образом верификация производится в веб-браузере, в рассматриваемой здесь программе предусмотрена возможность ее запуска в виде аплета. Загрузите эту программу как аплет в веб-браузер по следующему URL:

```
file:///C:/CoreJavaBook/v2ch9/verifier/VerifierTest.html
```

Как показано на рис. 9.5, в результате выполнения данной программы появится сообщение об ошибке, обнаруженной во время верификации байт-кода.

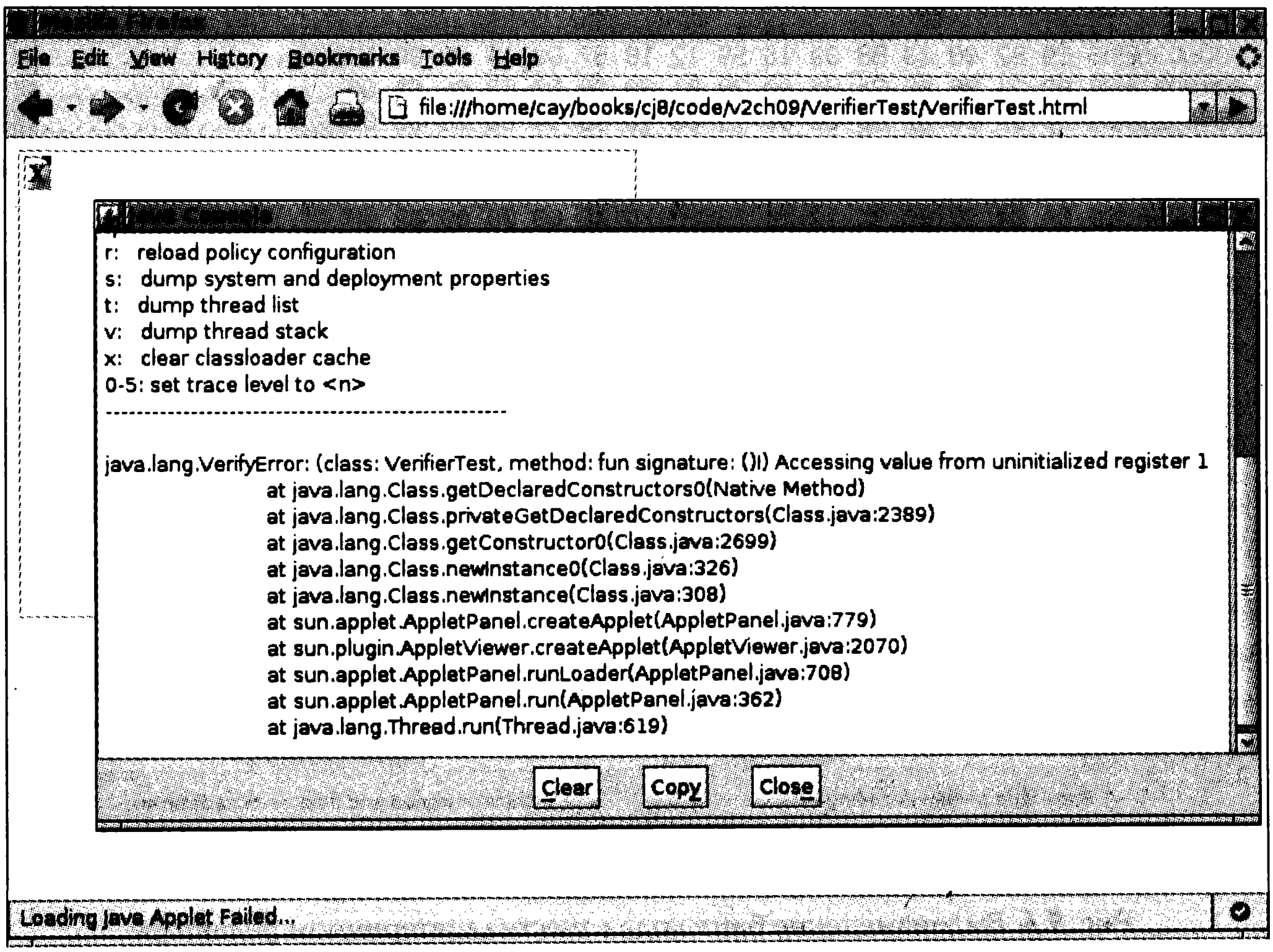


Рис. 9.5. Попытка загрузить преднамеренно видоизмененный файл класса приводит к ошибке верификации байт-кода

Диспетчеры защиты и полномочия

После загрузки класса в виртуальную машину и проверки верификатором в действие вступает второй механизм обеспечения безопасности на платформе Java: диспетчер защиты. В качестве *диспетчера защиты* служит класс, определяющий, разрешено ли коду выполнять ту или иную операцию. Ниже перечислены операции, подпадающие под контроль диспетчера защиты. Существует немало других проверок, выполняемых диспетчером защиты в библиотеке Java.

- Создание нового загрузчика классов.
- Выход из виртуальной машины.
- Получение доступа к члену другого класса с помощью рефлексии.
- Получение доступа к файлу.
- Установление соединения через сокет.
- Запуск задания на печать.
- Получение доступа к системному буферу обмена.
- Получение доступа к очереди событий в AWT.
- Обращение к окну верхнего уровня.

Поведение по умолчанию при запуске прикладных программ на Java *не* предусматривает никакой установки диспетчера защиты, поэтому все перечисленные выше операции оказываются разрешенными. Тем не менее программа просмотра аплетов принудительно применяет правила защиты, которые в значительной степени ограничивают количество разрешенных операций.

Например, аплетам не разрешается завершать работу виртуальной машины. При попытке аплета вызвать метод `exit()` выдается исключение системы безопасности. В частности, метод `exit()` из класса `Runtime` вызывает метод `checkExit()` диспетчера защиты. Ниже приведен исходный код метода `exit()`.

```
public void exit(int status)
{
  SecurityManager security = System.getSecurityManager();
  if (security != null)
    security.checkExit(status);
  exitInternal(status);
}
```

Далее диспетчер защиты проверяет, откуда поступил запрос на выход (т.е. на завершение работы виртуальной машины): из браузера или из отдельного аплета. Если диспетчер защиты дает разрешение на выполнение этого запроса, метод `checkExit()` просто возвращает управление, и далее процесс обработки продолжается обычным образом. Но если диспетчер защиты не запрещает выполнение запроса, метод `checkExit()` генерирует исключение типа `SecurityException`.

Метод `exit()` продолжает выполняться только в том случае, если никакого исключения не было. Затем он вызывает *закрытый собственный* метод `exitInternal()`, который и завершает работу виртуальной машины. Другого способа для завершения работы виртуальной машины не существует, а поскольку метод `exitInternal()` закрытый, то он не может вызываться из какого-нибудь другого класса. Следовательно, всякий код, делающий попытку выйти из виртуальной машины, должен непременно

вызывать метод `exit()`, а следовательно, проходить проверку защиты, выполняемую методом `checkExit()`, но без генерирования соответствующего исключения.

Очевидно, что целостность правил защиты зависит от тщательности программирования. Поставщики системных служб в стандартной библиотеке должны всегда обращаться к диспетчеру защиты перед попытками выполнить любые серьезные операции.

Диспетчер защиты на платформе Java позволяет как программистам, так и системным администраторам организовать тщательно продуманное управление отдельными полномочиями. Более подробно об этом речь пойдет в следующем разделе, где сначала приводится краткий обзор предлагаемой в Java модели защиты, а затем показывается, как организуется управление правами доступа с помощью *файлов правил защиты*, и, наконец, объясняется, как определять свои собственные виды прав доступа.

НА ЗАМЕТКУ! Допускается реализовывать и устанавливать свой собственный диспетчер защиты, но это лучше делать только тем, у кого имеется достаточный опыт работы в области компьютерной безопасности. Намного безопаснее просто настраивать нужным образом стандартный механизм диспетчера защиты.

Организация защиты на платформе Java

Модель защиты в версии JDK версии 1.0 была очень проста: локальные классы имели все полномочия, а возможности удаленных классов были ограничены так называемой "песочницей", т.е. диспетчер защиты аплета отказывал в любом доступе к локальным ресурсам. Подобно тому, как ребенку разрешается играть в песочнице, строго ограничивая его действия ее пределами, удаленные классы имели возможность лишь выводить сведения на экран и взаимодействовать с пользователем. В версии JDK 1.1 эта модель была немного видоизменена, т.е. доверенные удаленные объекты имели те же права доступа, что и локальные классы. Тем не менее обе версии модели защиты были организованы по принципу "все или ничего", т.е. программы получали полный доступ к ресурсам, или же их действие ограничивалось пределами "песочницы".

Начиная с версии Java SE 1.2 на платформе Java предоставляется намного более гибкий механизм защиты. *Правила защиты* преобразуют *источники кода* в *наборы полномочий*, как показано на рис. 9.6.

Каждый *источник кода* характеризуется *кодовой базой* и *набором сертификатов*. Кодовая база указывает место происхождения кода. Например, кодовой базой кода удаленного аплета является URL типа HTTP, откуда этот аплет загружен, а кодовой базой кода в архивном JAR-файле — URL, по которому загружен этот файл. Сертификаты, если таковые имеются, предоставляются соответствующей организацией и служат гарантией того, что данный код не является подделкой. Подробнее о сертификатах речь пойдет далее в этой главе.

Под *полномочиями* (иначе — правами доступа или привилегиями) подразумеваются любые свойства, которые проверяются диспетчером защиты. На платформе Java поддерживается целый ряд представляющих полномочия классов, каждый из которых инкапсулирует подробности конкретных полномочий. В качестве примера ниже приведено получение экземпляра класса `FilePermission`, дающего разрешение на чтение и запись любого файла в каталоге `/tmp`.

```
FilePermission p = new FilePermission("/tmp/*", "read,write");
```

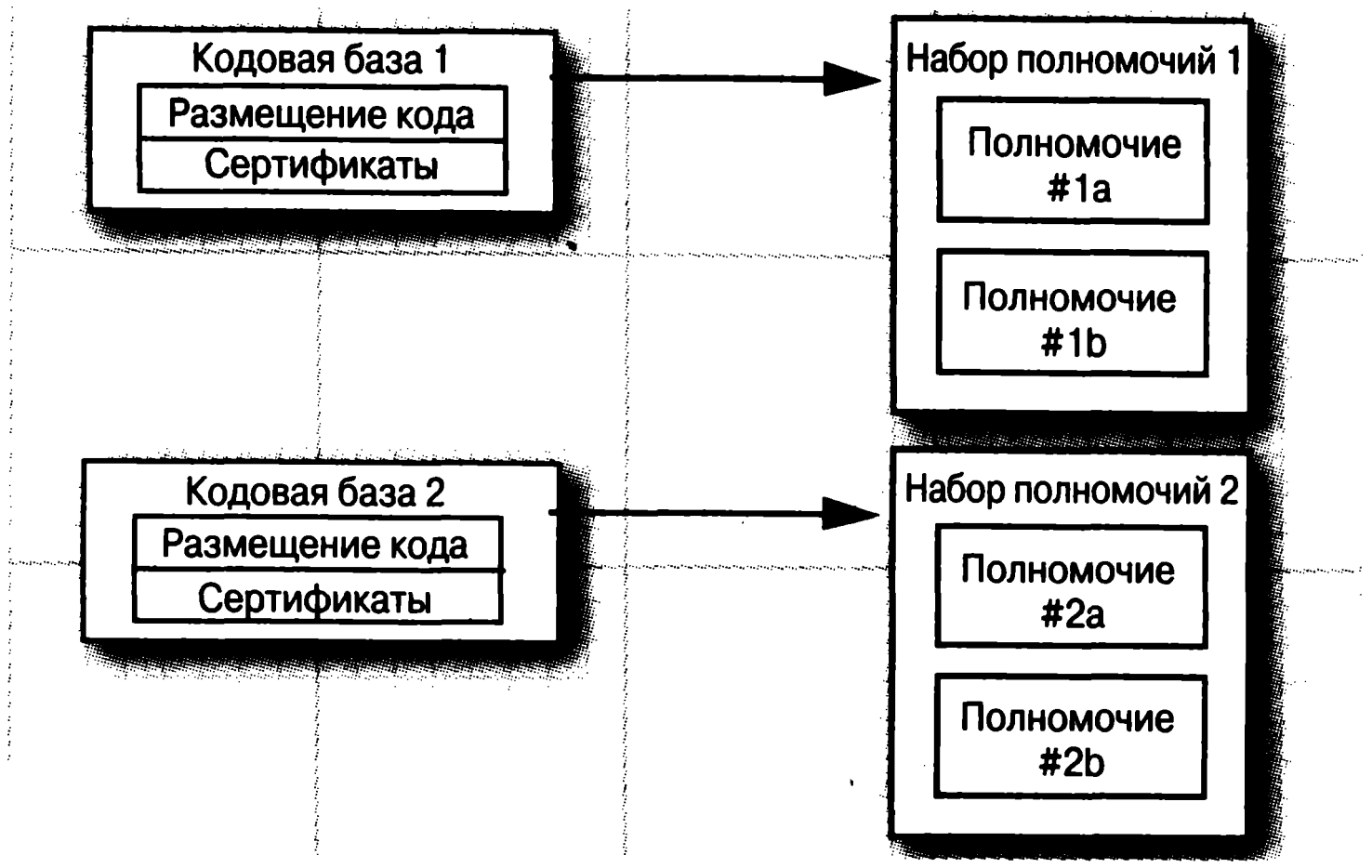


Рис. 9.6. Правила защиты

Но еще важнее, что в стандартной реализации класс `Policy` считывает полномочия из *файла прав доступа*. В этом файле те же самые полномочия на чтение и запись, что и выше, будут выражены следующим образом:

```
permission java.io.FilePermission "/tmp*", "read,write";
```

Более подробно файлы прав доступа рассматриваются в следующем разделе. На рис. 9.7 представлена иерархия классов полномочий, которые предоставлялись в версии Java SE 1.2. А в последующих версиях Java было внедрено немало других классов полномочий.

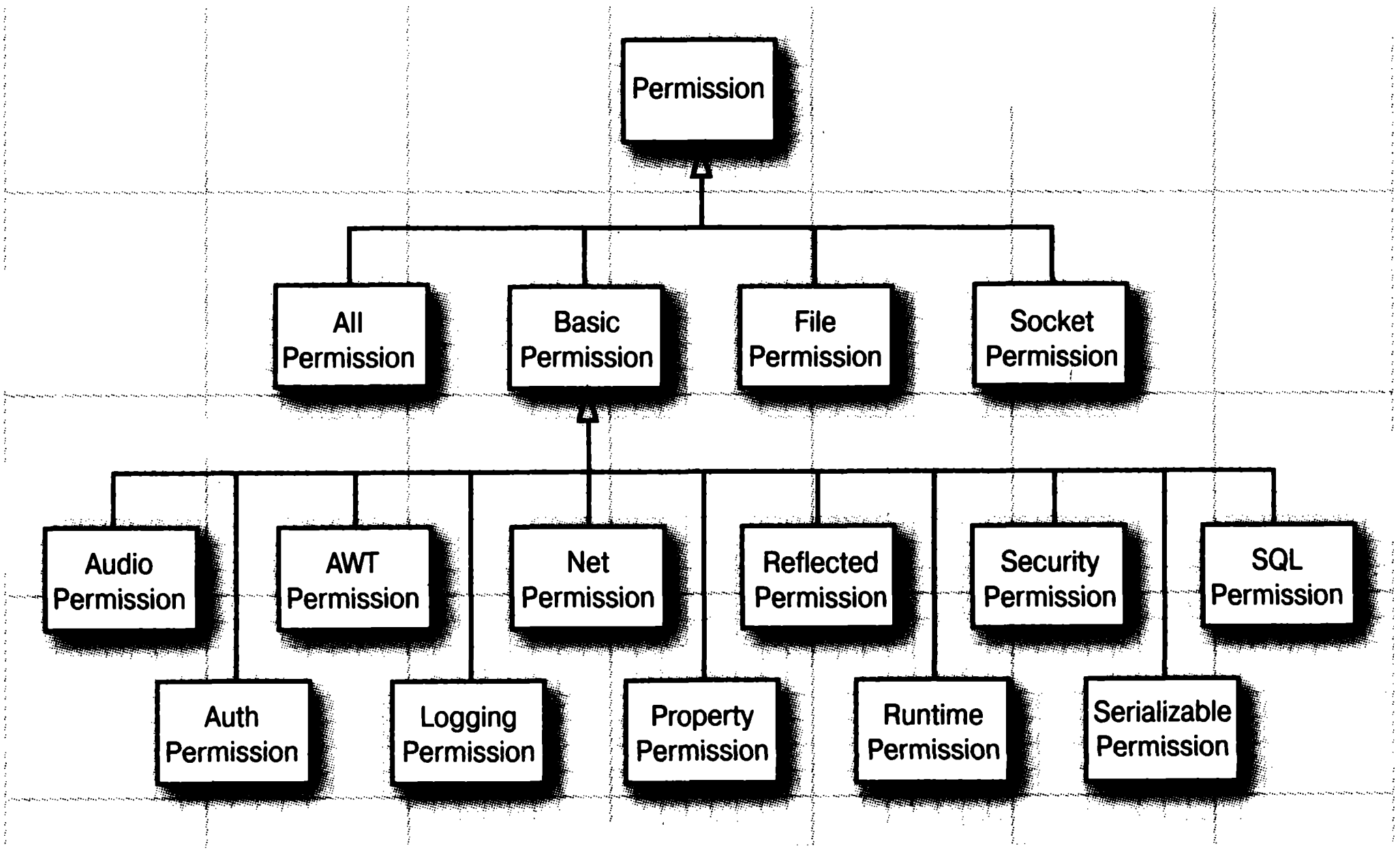


Рис. 9.7. Часть существующей иерархии классов полномочий

Как было показано в предыдущем разделе, в классе SecurityManager имеются методы проверки защиты типа checkExit(). Эти методы существуют только для удобства программирования и обратной совместимости. Все они отображаются в стандартные проверки полномочий. В качестве примера ниже приведен исходный код метода checkExit().

```
public void checkExit()
{
  checkPermission(new RuntimePermission("exitVM"));
}
```

У каждого класса имеется *домен защиты*, т.е. объект, который инкапсулирует как источник кода, так и набор прав доступа этого класса. При необходимости проверить то или иное полномочие диспетчер защиты типа SecurityManager сначала выясняет, к каким классам относятся все методы, находящиеся в настоящий момент в стеке вызовов. Затем он получает доступ к доменам защиты всех этих классов и выясняет, допускает ли их набор полномочий выполнение проверяемой в данный момент операции. Если все домены дают разрешение на выполнение данной операции, то проверка завершается успешно, в противном случае генерируется исключение типа SecurityException.

А почему разрешение на выполнение операции должны давать все методы в стеке вызовов? Для ответа на этот вопрос обратимся к конкретному примеру. Допустим, в методе init() из аплета требуется открыть файл с помощью следующего вызова:

```
Reader in = new FileReader(name);
```

Конструктор класса FileReader вызывает конструктор класса FileInputStream, который, в свою очередь, обращается к методу checkRead() диспетчера защиты, а тот вызывает метод checkPermission() с объектом, получаемым из конструктора FilePermission(name, "read"). Внешний вид стека вызовов в этом случае приведен в табл. 9.1.

Таблица 9.1. Стек вызовов во время проверки полномочий

Класс	Метод	Источник кода	Полномочия
SecurityManager	checkPermission	null	AllPermission
SecurityManager	checkRead	null	AllPermission
FileInputStream	Constructor	null	AllPermission
FileReader	Constructor	null	AllPermission
Applet	init	Источник кода аплета	Полномочия аплета

Классы FileInputStream и SecurityManager являются системными, их источник кода — пустым значением null, а набор полномочий —экземпляром класса AllPermission, который разрешает выполнять все операции. Очевидно, что полномочия только этих классов не могут определить исход проверки. Помимо них, метод checkPermission() должен непременно принять во внимание и ограниченные полномочия класса аплета. Благодаря проверке всего стека вызовов механизм защиты исключает вероятность выполнения одним классом важных операций от имени какого-нибудь другого класса.

НА ЗАМЕТКУ! Приведенное выше краткое описание проверки полномочий дает лишь самое общее представление о данном процессе. Многие технические детали в этом описании были опущены. Учитывая тот факт, что в вопросах безопасности вся суть скрывается именно в деталях, для ознакомления с ними рекомендуется книга *Securing Java: Getting Down to Business with Mobile Code* Гэри Макгроу и Эда Фельтена (Gary McGraw & Ed Felten; издательство Wiley, 1999 г.). Электронная версия книги доступна по адресу `www.securingjava.com`.

java.lang.SecurityManager 1.0

- `void checkPermission(Permission p)` **1.2**

  Проверяет, предоставляет ли диспетчер защиты указанные полномочия. Если не предоставляет, то генерируется исключение типа `SecurityException`.

java.lang.Class 1.0

- `ProtectionDomain getProtectionDomain()` **1.2**

  Получает доступ к представляющему данный класс домену защиты или возвращает пустое значение `null`, если этот класс загружен без домена защиты.

java.security.ProtectionDomain 1.2

- `ProtectionDomain(CodeSource source, PermissionCollection permissions)`

  Создает домен защиты, исходя из указанного источника кода и прав доступа.

- `CodeSource getCodeSource()`

  Получает источник кода из данного домена защиты.

- `boolean implies(Permission p)`

  Возвращает логическое значение `true`, если указанные полномочия разрешены в данном домене защиты.

java.security.CodeSource 1.2

- `Certificate[] getCertificates()`

  Получает цепочку сертификатов для подписей файлов классов, связанных с данным источником кода.

- `URL getLocation()`

  Получает кодовую базу файлов классов, связанных с данным источником кода.

Файлы правил защиты

Диспетчер правил защиты считывает содержимое *файлов правил защиты*, т.е. инструкции для преобразования источников кода в полномочия. Ниже приведен

пример типичного файла правил защиты. Этот файл предоставляет полномочия на чтение и запись файлов в каталог /tmp любому коду, загруженному по адресу http://www.horstmann.com/classes.

```
grant codeBase "http://www.horstmann.com/classes"
{
  permission java.io.FilePermission "/tmp/*", "read,write";
};
```

Устанавливаться файлы правил защиты могут только в стандартных местах. По умолчанию такими местами являются:

- файл java.policy в основном каталоге платформы Java;
- файл .java.policy в рабочем каталоге пользователя (обратите внимание на точку, стоящую в начале его имени).

НА ЗАМЕТКУ! Стандартные пути к этим файлам можно изменить в конфигурационном файле java.security, находящемся в каталоге *jre*/lib/security. По умолчанию пути к этим файлам указываются в конфигурационном файле и выглядят следующим образом:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

Системный администратор может редактировать файл java.security и указывать для правил защиты URL, находящиеся на каком-нибудь другом сервере, исключая всякую возможность для пользователей вносить изменения в эти правила. В файле правил защиты можно указать сколько угодно URL (по порядку следования номеров). Полномочия во всех подобных файлах все равно объединяются.

Если требуется, чтобы правила защиты хранились вне файловой системы, можно сначала реализовать подкласс, производный от класса Policy и собирающий сведения обо всех полномочиях, а затем изменить строку в конфигурационном файле java.security следующим образом:

```
policy.provider=sun.security.provider.PolicyFile
```

Во время тестирования не имеет смысла постоянно вносить изменения в стандартные файлы правил защиты, поэтому рекомендуется разместить сначала все полномочия в отдельном файле, например MyApp.policy, а затем указать явным образом имя этого файла для каждой прикладной программы. А применить правила защиты можно двумя способами. Во-первых, установить свойства системы в главном методе прикладной программы:

```
System.setProperty("java.security.policy", "MyApp.policy");
```

И во-вторых, запустить виртуальную машину, введя следующую команду:

```
java -Djava.security.policy=MyApp.policy MyApp
```

Для аплетов вместо этого следует ввести такую команду:

```
appletviewer -J-Djava.security.policy=MyApplet.policy MyApplet.html
```

(С помощью параметра **-J** утилиты appletviewer можно передать виртуальной машине любой аргумент из командной строки.)

В приведенных выше примерах файл MyApp.policy добавляется к остальным действующим правилам защиты. Но если добавить еще один знак равенства, т.е. ввести команду **java -Djava.security.policy==MyApp.policy MyApp**, то в прикладной

программе будет использоваться только указанный файл правил защиты, а все стандартные файлы правил защиты — игнорироваться.

ВНИМАНИЕ! Во время тестирования можно легко допустить ошибку, случайно оставив в текущем каталоге файл `.java.policy`, который может предоставлять многие, а то и все возможные полномочия (`AllPermission`). Поэтому, заметив, что приложение как будто не замечает ограничений, указанных в добавленном файле правил защиты, нужно прежде всего проверить, не остался ли в текущем каталоге файл `java.policy`. Вероятность допустить подобную ошибку наиболее велика на системах UNIX, где файлы, имена которых начинаются с точки, по умолчанию не отображаются.

Как упоминалось ранее, диспетчер защиты в прикладных программах на Java по умолчанию не устанавливается. Это означает, что до тех пор, пока он не будет установлен, увидеть действие файлов правил защиты не удастся. А установить диспетчер защиты можно двумя способами. Во-первых, добавить в тело метода `main()` следующую строку кода:

```
System.setSecurityManager(new SecurityManager());
```

И во-вторых, указать при запуске виртуальной машины параметр командной строки **`-Djava.security.manager`**, как показано ниже.

```
java -Djava.security.manager -Djava.security.policy=MyApp.policy MyApp
```

Далее в этом разделе подробно рассматриваются способы описания полномочий в файле правил защиты и его формата, кроме способов указания сертификатов, речь о которых пойдет ниже.

Итак, в файле правил защиты содержится последовательность записей `grant`, каждая из которых имеет следующий вид:

```
grant источник_кода
{
  полномочие_1;
  полномочие_2;
  ...
}
```

В источнике кода содержатся сведения о кодовой базе (которая может опускаться, если данная запись `grant` распространяется на код из всех источников), а также имена доверенных уполномоченных и подписавших сертификаты лиц (которые могут опускаться, если подписи не являются обязательными для данной записи). Сведения о кодовой базе указываются следующим образом:

```
codeBase "url"
```

Наличие косой черты (/) в конце URL означает, что ссылка делается на каталог, а отсутствие — на архивный JAR-файл, как показано в приведенных ниже примерах.

```
grant codeBase "www.horstmann.com/classes/" { . . . };
grant codeBase "www.horstmann.com/classes/MyApp.jar" { . . . };
```

В качестве разделителя файлов в URL кодовой базы должны всегда использоваться знаки косой черты, даже если речь идет об URL со ссылками на файлы в операционной системе Windows, как в следующем примере:

```
grant codeBase "file:C:/myapps/classes/" { . . . };
```

НА ЗАМЕТКУ! Как известно, URL типа HTTP всегда начинаются двумя знаками косой черты (`http://`). Но в отношении URL типа `file`, по-видимому, возникает путаница из-за того, что средство чтения файлов правил защиты разрешает использовать для них два формата: `file://`*`локальный_файл`* и `file:`*`локальный_файл`*. Более того, последняя косая черта перед именем диска в Windows не является обязательной. Это означает, что приемлемыми являются все следующие варианты:

```
file:C:/dir/filename.ext
file:/C:/dir/filename.ext
file://C:/dir/filename.ext
file:///C:/dir/filename.ext
```

На самом деле в рассматриваемых здесь примерах тестовых программ вариант `file:////C:/dir/filename.ext` также оказывается приемлемым, но найти этому разумное объяснение нам так и не удалось.

Полномочия задаются в следующей синтаксической форме:

```
permission ИмяКласса ИмяЦелевогоОбъекта, СписокДействий;
```

Вместо *ИмяКласса* указывается полностью уточненное имя класса, представляющего конкретные полномочия (например, `java.io.FilePermission`), а вместо *ИмяЦелевогоОбъекта* — конкретный целевой объект, на который должно распространяться действие указываемых полномочий. Так, для полномочий доступа к файлам это может быть имя файла и каталога, а для полномочий доступа к сокетам — название и номер порта хоста. И наконец, вместо параметра *СписокДействий* указывается перечень охватываемых данным полномочием допустимых действий (например, чтения или установления соединения), разделяемых запятой. В некоторых классах полномочий не требуется указывать ни имена целевых объектов, ни списки действий. В табл. 9.2 перечислены некоторые наиболее употребительные классы полномочий и охватываемые ими действия.

Таблица 9.2. Классы полномочий и связанные с ними целевые объекты и действия

Полномочия	Целевые объекты	Действия
`java.io.FilePermission`	Файлы	`read, write, execute, delete`
`java.net.SocketPermission`	Сокеты	`accept, connect, listen, resolve`
`ava.util.PropertyPermission`	Свойства	`read, write`
`java.lang.RuntimePermission`	`createClassLoader, getClassLoader, setContextClassLoader, enableContextClassLoaderOverride, createSecurityManager, setSecurityManager, exitVM, getenv.variableName, shutdownHooks, setFactory, setIO, modifyThread, stopThread, modifyThreadGroup, getProtectionDomain, readFileDescriptor, writeFileDescriptor, loadLibrary.libraryName, accessClassInPackage.packageName, defineClassInPackage.packageName, accessDeclaredMembers.className, queuePrintJob, getStackTrace, setDefaultUncaughtExceptionHandler, preferences, usePolicy`	Отсутствуют

Окончание табл. 9.2

Полномочия	Целевые объекты	Действия
java.awt.AWTPermission	showWindowWithoutWarningBanner, accessClipboard, accessEventQueue, createRobot, fullScreenExclusive, listenToAllAWTEvents, readDisplayPixels, replaceKeyboardFocusManager, watchMousePointer, setWindowAlwaysOnTop, setAppletStub	Отсутствуют
java.net.NetPermission	setDefaultAuthenticator, specifyStreamHandler, requestPasswordAuthentication, setProxySelector, getProxySelector, setCookieHandler, getCookieHandler, setResponseCache, getResponseCache	Отсутствуют
java.lang.reflect.ReflectPermission	suppressAccessChecks	Отсутствуют
java.io.SerializablePermission	enableSubclassImlementation, enableSubstitution	Отсутствуют
java.security.SecurityPermission	createAccessControlContext, getDomainCombiner, getPolicy, setPolicy, getProperty.keyName, setProperty.keyName, insertProvider.providerName, removeProvider.providerName, setSystemScope, setIdentityPublicKey, setIdentityInfo, addIdentityCertificate, removeIdentityCertificate, printIdentity, clearProviderProperties.providerName, putProviderProperty.providerName, removeProviderProperty.providerName, getSignerPrivateKey, setSignerKeyPair	Отсутствуют
java.security.AllPermission	Отсутствуют	Отсутствуют
javax.audio.AudioPermission	Воспроизведение записи	Отсутствуют
javax.security.auth.AuthPermission	doAs, doAsPrivileged, getSubject, getSubjectFromDomainCombiner, setReadOnly, modifyPrincipals, modifyPublicCredentials, modifyPrivateCredentials, refreshCredential, destroyCredential, createLoginContext.contextName, getLoginConfiguration, setLoginConfiguration, refreshLoginConfiguration, javax.security.auth.AuthPermission	Отсутствуют
java.util.logging.LoggingPermission	control	Отсутствуют
java.sql.SQLPermission	setLog	Отсутствуют

Как следует из табл. 9.2, большинство полномочий просто позволяют выполнять определенные операции. К операции можно относиться как к целевому объекту с подразумеваемым действием "permit" (разрешить). Все классы полномочий, перечисленные в табл. 9.2, являются производными от класса BasicPermission (см. рис. 9.7). Но классы с такими целевыми объектами, как файл, сокет и свойство, оказываются более сложными и поэтому заслуживают более подробного рассмотрения.

Целевые объекты прав доступа к файлам могут иметь следующий вид:

файл	Файл
каталог/	Каталог
*каталог/**	Все файлы из данного каталога
*	Все файлы из текущего каталога
каталог/-	Все файлы из данного каталога и всех его подкаталогов
-	Все файлы из текущего каталога и всех его подкаталогов
<<ALL FILES>>	Все файлы из файловой системы

Например, приведенная ниже запись полномочий означает, что доступ разрешается ко всем файлам в каталоге /myapp и любом из его подкаталогов.

```
permission java.io.FilePermission "/myapp/-", "read,write,delete";
```

Для обозначения обратной косой черты в пути к файлам в ОС Windows этот знак нужно повторить дважды, как показано ниже.

```
permission java.io.FilePermission "c:\\myapp\\-", "read,write,delete";
```

Целевые объекты прав доступа через сокет требуют указания узла (хоста) и диапазона портов. Синтаксическая форма для указания хоста выглядит следующим образом:

имя_хоста или *IP-адрес*	Одиночный узел
localhost или пустая строка	Локальный узел
*.*суффикс_домена*	Любой узел, принадлежащий домену, имя которого оканчивается указанным суффиксом
*	Все узлы

Номера портов являются необязательными и указываются в приведенной ниже форме.

:n	Единственный порт
:n-	Все порты с номерами *n* и выше
:-n	Все порты с номерами *n* и ниже
:n1-n2	Все порты в пределах от *n1* до *n2*

Ниже приведен пример записи прав доступа через сокет.

```
permission java.net.SocketPermission "*.horstmann.com:8000-8999", "connect";
```

И наконец, целевые объекты прав доступа к свойствам могут принимать одну из двух следующих форм:

свойство	Отдельное свойство
префикс_свойства.*	Все свойства с указанным суффиксом

Таким образом, права доступа к свойствам могут выглядеть как "java.home" и как "java.vm.*". Например, следующая запись полномочий означает, что программе разрешается считывать все свойства, начинающиеся с java.vm:

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

В файлах правил защиты допускается использовать и системные свойства. Лексема ${*свойство*} в этом случае заменяется значением свойства. Например, лексема ${user.home} заменяется основным каталогом пользователя. Ниже приведен типичный пример применения этого системного свойства в записи полномочий.

```
permission java.io.FilePermission "${user.home}", "read,write";
```

Для составления не зависящих от используемой платформы файлов правил защиты вместо явных разделителей / или \\ рекомендуется использовать свойство file.separator или даже его сокращенный вариант ${/}, чтобы упростить дело. Например, для предоставления прав на чтение и запись в основном каталоге пользователя и всех его подкаталогах можно использовать следующую запись, удобную в отношении переносимости:

```
permission java.io.FilePermission "${user.home}${/}-", "read,write";
```

НА ЗАМЕТКУ! В состав JDK входит примитивная утилита policytool, предназначенная для редактирования файлов правил защиты (рис. 9.8). Очевидно, что для конечных пользователей эта утилита не подходит, поскольку большинство ее параметров будет для них совершенно непонятным. На наш взгляд, она больше подходит для системных администраторов, которые предпочитают указывать нужный параметр и щелкать, а не вводить сложную синтаксическую конструкцию вручную. Тем не менее этой утилите недостает понятного набора уровней защиты (например, низкого, среднего и высокого), что может быть очень важно для неспециалистов. В общем, мы считаем, что платформа Java содержит все компоненты, требующиеся для создания подробной модели защиты, но в то же время на этой платформе могли бы быть предусмотрены и более удобные средства для работы с подобными моделями как для конечных пользователей, так и для системных администраторов.

Пользовательские полномочия

В этом разделе будет показано, каким образом создаются собственные классы полномочий, на которые пользователи могут ссылаться в файлах правил защиты. Для реализации классов полномочий следует расширить класс Permission и реализовать перечисленные ниже методы.

- Конструктор с двумя строковыми параметрами для указания объекта и списка действий.
- Метод String getActions().
- Метод boolean equals().
- Метод int hashCode().
- Метод boolean implies(Permission *other*).

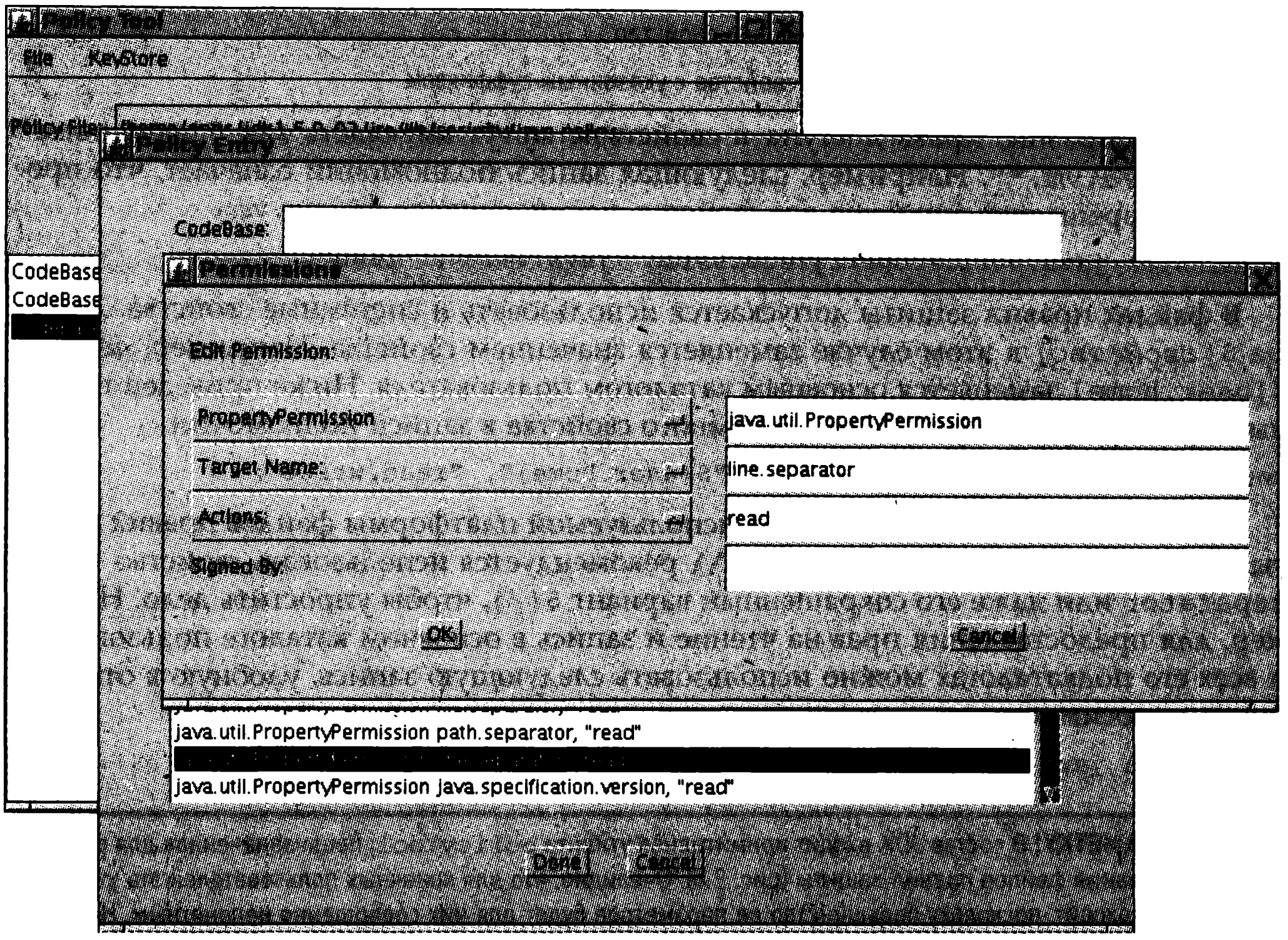


Рис. 9.8. Утилита `policytool`

Последний метод самый важный. Для полномочий предусмотрен определенный *порядок*, в соответствии с которым наиболее общие полномочия *предполагают* использование специальных прав доступа. Например, приведенное ниже право доступа к файлу разрешает чтение и запись любого файла из каталога `/tmp` и любых его подкаталогов.

```
p1 = new FilePermission("/tmp/-", "read, write");
```

Это общее право доступа предполагает наличие других, специальных прав доступа:

```
p2 = new FilePermission("/tmp/-", "read");
p3 = new FilePermission("/tmp/aFile", "read, write");
p4 = new FilePermission("/tmp/aDirectory/-", "write");
```

Иными словами, право доступа к файлу `p1` предполагает наличие другого права доступа `p2`, если выполняются два условия.

1. Набор целевых файлов в `p1` содержит набор целевых файлов в `p2`.
2. Набор действий в `p1` содержит набор действий в `p2`.

Рассмотрим, например, применение метода `implies()`. Если конструктор класса `FileInputStream()` открывает файл для чтения, то он проверяет наличие прав такого доступа. Для выполнения этой проверки *специальный* объект прав доступа передается методу `checkPermission()`, как показано ниже. После этого диспетчер защиты запрашивает все имеющиеся права доступа, предполагается ли специальное право доступа.

```
checkPermission(new FilePermission(fileName, "read"));
```

В частности, полномочия типа `AllPermission` предполагают все права доступа. При определении собственных классов полномочий необходимо определить предполагаемые права доступа. Например, для определения прав доступа пользователя `Tommy` к телевизору под управлением технологии Java в некотором промежутке времени можно определить следующий класс `TVPermission`:

```
new TVPermission("Tommy:2-12:1900-2200", "watch,record")
```

Этот класс разрешает пользователю `Tommy` смотреть и записывать телевизионные передачи на каналах **2–12** в период времени от **19:00** до **22:00**. Для применения приведенного ниже специального права доступа придется реализовать метод `implies()`.

```
new TVPermission("Tommy:4:2000-2100", "watch")
```

Реализация класса полномочий

В этом разделе на примере конкретной программы демонстрируется реализация новых полномочий для контроля над текстом, вставляемым в текстовую область. В обязанности этой программы входит предотвращение попыток ввода в текстовую область всевозможных "непристойных слов", вроде `sex`, `drugs` и `C++`, но не `rock-n-roll`! Для того чтобы предоставить возможность передавать список всех подобных непристойных слов в файл правил защиты, применяется специальный класс полномочий. Он является производным от класса `JTextArea` и всегда запрашивает диспетчер защиты, можно ли вводить новый текст в текстовую область, как показано ниже. Если диспетчер защиты предоставляет полномочия типа `WordCheckPermission`, то новый текст вводится, а иначе метод `checkPermission()` генерирует исключение.

```
class WordCheckTextArea extends JTextArea
{
  public void append(String text)
  {
    WordCheckPermission p = new WordCheckPermission(text, "insert");
    SecurityManager manager = System.getSecurityManager();
    if (manager != null) manager.checkPermission(p);
    super.append(text);
  }
}
```

Полномочия на проверку слов (`WordCheckPermission`) допускают выполнение двух следующих действий: `insert` (вставка указанного текста) и `avoid` (ввод текста без указанных непристойных слов). Запускать рассматриваемую здесь программу следует с помощью приведенного ниже файла правил защиты. В этом файле предоставляется разрешение на вставку любого текста, который не содержит такие непристойные слова, как `sex`, `drugs` и `C++`.

```
grant
{
  permission permissions.WordCheckPermission "sex,drugs,C++", "avoid";
};
```

При разработке класса `WordCheckPermission` особое внимание следует уделить методу `implies()`. Ниже перечислены правила, определяющие, должны ли полномочия `p1` предполагать полномочия `p2`.

- Если полномочия p1 допускают действие avoid, а полномочия p2 — действие insert, то целевой объект с полномочиями p2 должен исключать все слова из полномочий p1. Например, полномочия WordCheckPermission "sex,drugs,C++", "avoid" предполагают полномочия permissions.WordCheckPermission "Mary had a little lamb", "insert".
- Если оба вида полномочий p1 и p2 допускают действие avoid, то набор слов с полномочиями p2 должен содержать все слова из набора слов с полномочиями p1. Например, полномочия permissions.WordCheckPermission "sex,drugs", "avoid" предполагают полномочия permissions.WordCheckPermission "sex,drugs,C++", "avoid".
- Если оба вида полномочий p1 и p2 допускают действие insert, то текст с полномочиями p1 должен содержать текст с полномочиями p2. Например, полномочия permissions.WordCheckPermission "Mary had a little lamb", "insert" предполагают полномочия permissions.WordCheckPermission "a little lamb", "insert".

Исходный код, реализующий класс WordCheckPermission, представлен в листинге 9.4. Следует заметить, что целевой объект полномочий извлекается методом с не совсем подходящим именем getName() из класса Permission.

Полномочия описываются в файлах правил защиты с помощью пары символьных строк, поэтому классы прав доступа должны быть подготовлены к синтаксическому анализу этих строк. Так, в рассматриваемом здесь примере для преобразования списка разделяемых запятыми непристойных слов с полномочиями avoid в подлинное множество типа Set используется следующий метод:

```
public Set<String> badWordSet()
{
  Set<String> set = new HashSet<String>();
  set.addAll(Arrays.asList(getName().split(",")));
  return set;
}
```

Этот код позволяет использовать для сравнения множеств (в данном случае — наборов слов) методы equals() и containsAll(). Как было показано в главе 2, метод equals() из класса Set признает два множества равными в том случае, если в них содержатся одинаковые элементы, независимо от порядка их расположения. Например, наборы слов "sex,drugs,C++" и "C++,drugs,sex" будут признаны равными.

ВНИМАНИЕ! Класс полномочий должен обязательно быть открытым (public). Загрузчик файлов правил защиты не может загружать классы с уровнем доступности пакета, выходящим за пределы доступности пути к классам начальной загрузки, и поэтому он просто игнорирует любые классы, которые ему не удается обнаружить.

В исходном коде, приведенном в листинге 9.5, показано, каким образом действует класс WordCheckPermission. Попробуйте ввести в текстовой области какой-нибудь текст и щелкнуть на кнопке Insert (Вставить). Если проверка на безопасность пройдет успешно, текст будет вставлен в текстовую область, в противном случае появится сообщение об ошибке (рис. 9.9).

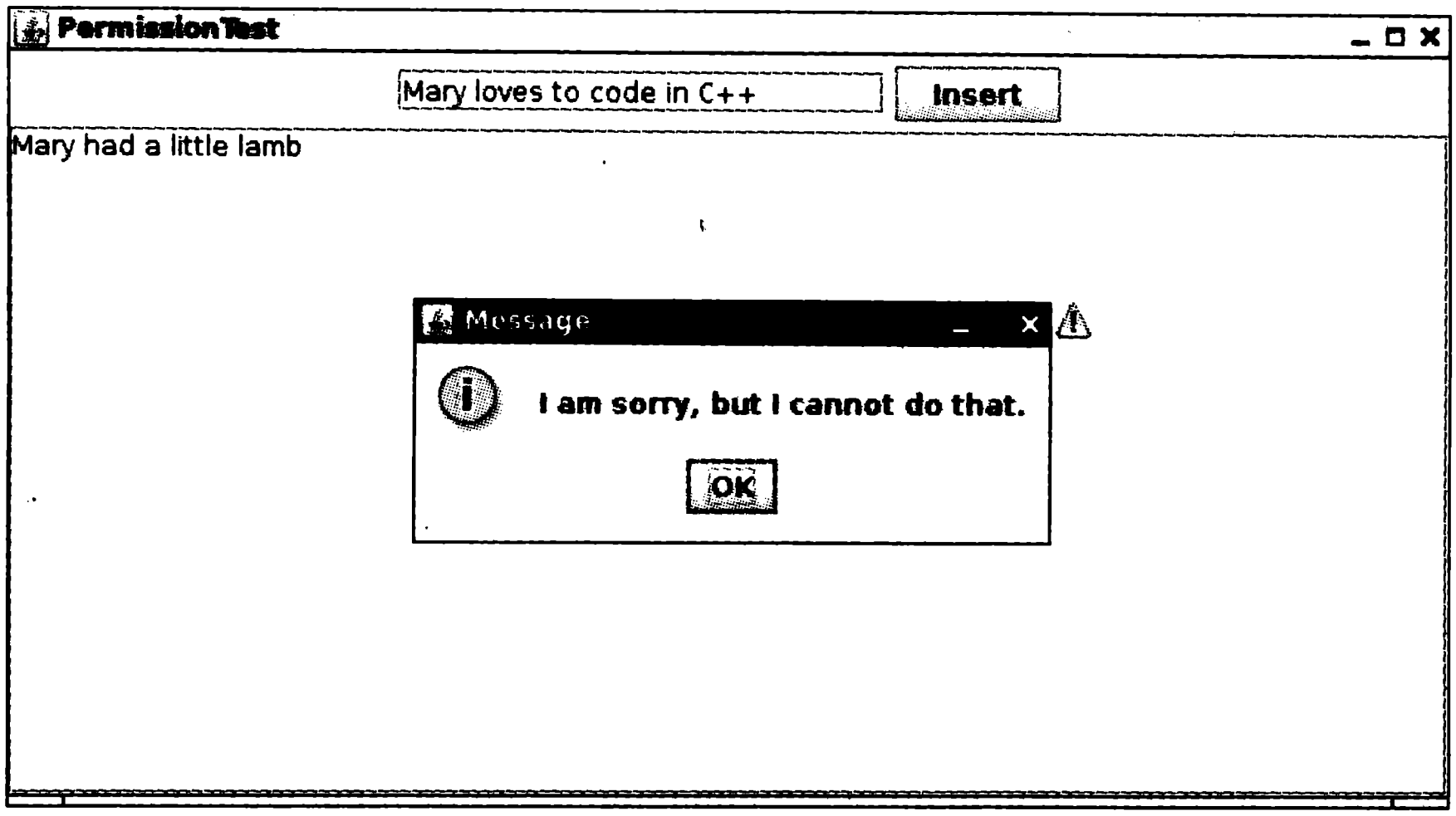


Рис. 9.9. Программа `PermissionTest` в действии

ВНИМАНИЕ! Глядя на рис. 9.9, можно заметить маленький треугольник в окне сообщения, предупреждающий о том, что данное окно могло появиться без разрешения. Некогда грозное сообщение "`Untrusted Java Applet Window`" (Окно не заслуживающего доверия аплета) в последующих версиях JDK было смягчено и стало, по существу, бесполезным для предупреждения пользователей. Это предупреждение можно отключить с помощью целевого объекта `showWindowWithoutWarningBanner` полномочий типа `java.awt.AWTPermission`. Для того чтобы предоставить эти полномочия, достаточно отредактировать файл правил защиты.

Листинг 9.4. Исходный код из файла `permissions/WordCheckPermission.java`

```
1  package permissions;
2
3  import java.security.*;
4  import java.util.*;
5
6  /**
7   * Полномочия на проверку непристойных слов
8  */
9  public class WordCheckPermission extends Permission
10 {
11   private String action;
12
13   /**
14    * Конструирует полномочия на проверку непристойных слов
15    * @param target Список разделяемых запятыми слов
16    * @param anAction Действие "вставить" или "исключить"
17   */
18   public WordCheckPermission(String target, String anAction)
19   {
20     super(target);
21     action = anAction;
22   }
23   public String getActions()
```

```
24    {
25       return action;
26    }
27
28    public boolean equals(Object other)
29    {
30       if (other == null) return false;
31       if (!getClass().equals(other.getClass())) return false;
32       WordCheckPermission b = (WordCheckPermission) other;
33       if (!Objects.equals(action, b.action)) return false;
34       if ("insert".equals(action)) return
35          Objects.equals(getName(), b.getName());
36       else if ("avoid".equals(action)) return
37          badWordSet().equals(b.badWordSet());
38       else return false;
39    }
40
41    public int hashCode()
42    {
43       return Objects.hash(getName(), action);
44    }
45
46    public boolean implies(Permission other)
47    {
48       if (!(other instanceof WordCheckPermission)) return false;
49       WordCheckPermission b = (WordCheckPermission) other;
50       if (action.equals("insert"))
51       {
52          return b.action.equals("insert") &&
53             getName().indexOf(b.getName()) >= 0;
54       }
55       else if (action.equals("avoid"))
56       {
57          if (b.action.equals("avoid")) return
58             b.badWordSet().containsAll(badWordSet());
59          else if (b.action.equals("insert"))
60          {
61             for (String badWord : badWordSet())
62                if (b.getName().indexOf(badWord) >= 0) return false;
63             return true;
64          }
65          else return false;
66       }
67       else return false;
68    }
69
70    /**
71     * Получает непристойные слова, описываемые данным правилом прав доступа
72     * @return Возвращает непристойные слова
73    */
74    public Set<String> badWordSet()
75    {
76       Set<String> set = new HashSet<>();
77       set.addAll(Arrays.asList(getName().split(",")));
78       return set;
79    }
80 }
```

Листинг 9.5. Исходный код из файла `permissions/PermissionTest.java`

```
1   package permissions;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6
7   /**
8    * Этот класс демонстрирует применение специальных полномочий
9    * типа WordCheckPermission
10   * @version 1.03 2007-10-06
11   * @author Cay Horstmann
12  */
13  public class PermissionTest
14  {
15    public static void main(String[] args)
16    {
17      System.setProperty("java.security.policy",
18         "permissions/PermissionTest.policy");
19      System.setSecurityManager(new SecurityManager());
20      EventQueue.invokeLater(new Runnable()
21        {
22          public void run()
23          {
24            JFrame frame = new PermissionTestFrame();
25            frame.setTitle("PermissionTest");
26            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27            frame.setVisible(true);
28          }
29        });
30    }
31  }
32
33  /**
34   * Этот фрейм содержит текстовое поле для ввода слов в текстовую
35   * область, защищенную от вставки непристойных слов
36  */
37  class PermissionTestFrame extends JFrame
38  {
39    private JTextField textField;
40    private WordCheckTextArea textArea;
41    private static final int TEXT_ROWS = 20;
42    private static final int TEXT_COLUMNS = 60;
43
44    public PermissionTestFrame()
45    {
46      textField = new JTextField(20);
47      JPanel panel = new JPanel();
48      panel.add(textField);
49      JButton openButton = new JButton("Insert");
50      panel.add(openButton);
51      openButton.addActionListener(new ActionListener()
52        {
53          public void actionPerformed(ActionEvent event)
54          {
```

```
55            insertWords(textField.getText());
56          }
57        });
58
59      add(panel, BorderLayout.NORTH);
60
61      textArea = new WordCheckTextArea();
62      textArea.setRows(TEXT_ROWS);
63      textArea.setColumns(TEXT_COLUMNS);
64      add(new JScrollPane(textArea), BorderLayout.CENTER);
65      pack();
66    }
67
68    /**
69     * Пытается вставить слова в текстовую область. Отображает
70     * диалоговое окно, если попытка окажется неудачной
72     * @param words Вставляемые слова
73     */
74    public void insertWords(String words)
75    {
76      try
77      {
78        textArea.append(words + "\n");
79      }
80      catch (SecurityException ex)
81      {
82        JOptionPane.showMessageDialog(this,
83           "I am sorry, but I cannot do that.");
84        ex.printStackTrace();
85      }
86    }
87  }
88
89  /**
90   * Текстовая область, в которой метод ввода текста
91   * проверяет в целях безопасности, чтобы в нее не были
92   * вставлены непристойные слова
95  */
96  class WordCheckTextArea extends JTextArea
97  {
98    public void append(String text)
99    {
100     WordCheckPermission p = new WordCheckPermission(text, "insert");
101     SecurityManager manager = System.getSecurityManager();
102     if (manager != null) manager.checkPermission(p);
103     super.append(text);
104   }
105 }
```

На этом рассмотрение способов и средств настройки безопасности на платформе Java завершается. Как правило, вам придется лишь подкорректировать должным образом стандартные полномочия. Но если потребуются дополнительные средства для контроля безопасности, вы сможете теперь определить специальные классы полномичий и настроить их таким же образом, как и стандартные полномочия.

java.security.Permission 1.2

- `Permission(String` *name*`)`

  Создает полномочия с указанным именем целевого объекта.

- `String getName()`

  Возвращает имя целевого объекта для данных полномочий.

- `boolean implies(Permission` *other*`)`

  Проверяет, предполагают ли данные полномочия другие полномочия. Это имеет место в том случае, если в других полномочиях описывается более конкретное условие, вытекающее из условия, указанного в данных полномочиях.

Аутентификация пользователей

Начиная с версии Java SE 1.4 в состав платформы Java входит JAAS (Java Authentication and Authorization Service — служба аутентификации и авторизации Java). Слово "Authentication" в названии этой службы указывает на возможность опознавания пользователей программ, а слово "Authorization" — на возможность проверки их полномочий.

Служба JAAS, по существу, представляет собой встраиваемый прикладной интерфейс API, отделяющий прикладные программы на Java от конкретной технологии, применяемой для реализации средств аутентификации. Помимо прочего, эта служба поддерживает механизмы регистрации в UNIX и NT, механизм аутентификации Kerberos и механизмы аутентификации по сертификатам.

После аутентификации за пользователем может быть закреплен определенный набор полномочий. Ниже приведен пример, в котором пользователю `harry` предоставляется особый ряд полномочий, которых нет ни у кого из других пользователей.

```
grant principal com.sun.security.auth.UnixPrincipal "harry"
{
  permission java.util.PropertyPermission "user.*", "read";
  . . .
};
```

В данном пример класс `com.sun.security.auth.UnixPrincipal` выполняет проверку имени запускающего данную программу пользователя UNIX. Метод `getName()` из этого класса возвращает имя пользователя, зарегистрированное в UNIX, которое затем сравнивается на равенство с именем `harry`.

Для того чтобы предоставить диспетчеру защиты возможность проверить правильность предоставления таких полномочий, служит класс `LoginContext`. Ниже приведена общая структура кода регистрации.

```
try
{
  System.setSecurityManager(new SecurityManager());
  LoginContext context = new LoginContext("Login1"); // определяется в
                                    // конфигурационном файле службы JAAS
  context.login();
  // получить аутентифицированный объект типа Subject
  Subject subject = context.getSubject();
```

```
  context.logout();
}
catch (LoginException exception) // это исключение генерируется, если
                                 // попытка регистрации оказалась неудачной
{
  exception.printStackTrace();
}
```

Теперь объект `subject` обозначает прошедшего аутентификацию пользователя. Строковый параметр `"Login1"` в конструкторе класса `LoginContext` обозначает запись с аналогичным именем в конфигурационном файле службы JAAS. Ниже приведен пример такого конфигурационного файла.

```
Login1
{
  com.sun.security.auth.module.UnixLoginModule required;
  com.whizzbang.auth.module.RetinaScanModule sufficient;
};

Login2
{
  . . .
};
```

Разумеется, в JDK не предусмотрено никаких модулей для биометрической регистрации. В состав пакета `com.sun.security.auth.module` входят только следующие модули:

```
UnixLoginModule
NTLoginModule
Krb5LoginModule
JndiLoginModule
KeyStoreLoginModule
```

Правила регистрации состоят из ряда модулей, каждый из которых обозначен как `required`, `sufficient`, `requisite` или `optional`. Значение этих ключевых слов можно понять из приведенного ниже описания алгоритма регистрации.

1. Модули выполняются по очереди до тех пор, пока выполнение модуля `sufficient` не завершится успешно, а выполнение модуля `requisite` — неудачно, или же до тех пор, пока не будет достигнут конец списка модулей.
2. Аутентификация проходит успешно в том случае, если выполнение всех модулей `required` и `requisite` завершается успешно, ни один из них вообще не выполняется или хотя бы один из модулей `sufficient` или `optional` завершается успешно.

При регистрации выполняется аутентификация *субъекта*, который может иметь несколько *принципалов*. Каждый принципал просто описывает какое-то свойство субъекта, например, имя пользователя, идентификатор группы или роль. Как было показано ранее в операторе `grant`, принципалы управляют правами доступа. Объект типа `com.sun.security.auth.UnixPrincipal` описывает имя пользователя, зарегистрированное в UNIX, а объект типа `UnixNumeric Group Principal` может выполнять проверку на принадлежность пользователя к группе в UNIX. Оператор `grant` позволяет проверить наличие принципала с помощью следующей синтаксической конструкции:

```
grant КлассПринципала "ИмяПринципала"
```

Следовательно, в рассматриваемом здесь примере этот оператор принимает следующий вид:

```
grant com.sun.security.auth.UnixPrincipal "harry"
```

Когда пользователь регистрируется, в отдельном контексте управления доступом должен быть выполнен код, требующий проверки принципалов. Для инициирования нового привилегированного действия типа `PriviligedAction` вызывается статический метод `doAs()` или `doAsPrivileged()`. Оба метода выполняют действие, вызывая метод `run()` для объекта, класс которого реализует интерфейс `PrivilegedAction`, а также используя полномочия принципалов субъекта, как показано ниже. Если же при выполнении действий генерируются проверяемые исключения, то вместо упомянутого выше интерфейса лучше реализовать интерфейс `PriviligedExceptionAction`.

```
PrivilegedAction<T> action = new PrivilegedAction()
  {
    public T run()
    {
      // выполнить с полномочиями принципалов субъекта
      . . .
    }
  };
T result = Subject.doAs(subject, action);
    // или Subject.doAsPrivileged(subject, action, null)
```

Методы `doAs()` и `doAsPrivileged()` отличаются лишь незначительно. Так, метод `doAs()` запускается в текущем контексте управления доступом, а метод `doAsPrivileged()` — в новом контексте. Кроме того, метод `doAsPrivileged()` позволяет разделять права доступа для кода регистрации и бизнес-логики. В рассматриваемом здесь примере прикладной программы код регистрации имеет следующие полномочия:

```
permission javax.security.auth.AuthPermission "createLoginContext.Login1";
permission javax.security.auth.AuthPermission "doAsPrivileged";
```

Пользователь, прошедший аутентификацию, получает приведенные ниже полномочия. Если бы вместо метода `doAsPrivileged()` использовался метод `doAs()`, такие же полномочия требовались бы и коду регистрации!

```
permission java.util.PropertyPermission "user.*", "read";
```

В рассматриваемом здесь примере программы `AuthTest` (ее исходный код приведен в листингах 9.6 и 9.7) демонстрируется, как ограничивать права доступа определенных пользователей. Эта программа аутентифицирует пользователя, а затем выполняет действие, извлекающее системное свойство. Для того чтобы данный пример программы оказался работоспособным, коды регистрации и действия должны находиться в двух отдельных JAR-файлах:

```
javac auth/*.java
jar cvf login.jar auth/AuthTest.class
jar cvf action.jar auth/SysPropAction.class
```

Как следует из файла правил защиты, содержимое которого приведено в листинге 9.8, пользователь ОС UNIX с именем `harry` обладает правами на чтение всех

файлов. Замените сначала имя `harry` своим учетным именем, а затем выполните приведенную ниже команду. Настройка регистрации представлена в листинге 9.9.

```
java -classpath login.jar:action.jar
    -Djava.security.policy=auth/AuthTest.policy
    -Djava.security.auth.login.config=auth/jaas.config
     auth.AuthTest
```

В ОС Windows нужно заменить `UnixPrincipal` на `NTUserPrincipal` в файлах `AuthTest.policy` и `jaas.configUnix` и использовать точку с запятой для разделения JAR-файлов, как показано в приведенной ниже команде.

```
java -classpath login.jar;action.jar . . .
```

После этого программа `AuthTest` должна отобразить значение свойства `user.home`. Но если зарегистрироваться под другим учетным именем, то должно возникнуть исключение, поскольку у данного пользователя больше не будет требуемых полномочий.

ВНИМАНИЕ! Очень важно выполнить *все* приведенные выше инструкции точно и аккуратно, поскольку даже незначительные отклонения могут привести к неверной настройке аутентификации пользователей.

Листинг 9.6. Исходный код из файла `auth/AuthTest.java`

```
1  package auth;
2
3  import java.security.*;
4  import javax.security.auth.*;
5  import javax.security.auth.login.*;
6
7  /**
8   * В этой программе демонстрируется аутентификация пользователей
9   * через специальную регистрацию и последующее выполнение действия
10  * типа SysPropAction с привилегиями зарегистрированного пользователя
11  * @version 1.01 2007-10-06
12  * @author Cay Horstmann
13 */
14 public class AuthTest
15 {
16   public static void main(final String[] args)
17   {
18     System.setSecurityManager(new SecurityManager());
19     try
20     {
21       LoginContext context = new LoginContext("Login1");
22       context.login();
23       System.out.println("Authentication successful.");
24       Subject subject = context.getSubject();
25       System.out.println("subject=" + subject);
26       PrivilegedAction<String> action = new SysPropAction("user.home");
27       String result = Subject.doAsPrivileged(subject, action, null);
28       System.out.println(result);
29       context.logout();
30     }
31     catch (LoginException e)
32     {
33       e.printStackTrace();
```

```
34      }
35    }
36 }
```

Листинг 9.7. Исходный код из файла auth/SysPropAction.java

```
1  package auth;
2
3  import java.security.*;
4
5  /**
6   * Это действие осуществляет поиск системного свойства
7   * @version 1.01 2007-10-06
8   * @author Cay Horstmann
9  */
10 public class SysPropAction implements PrivilegedAction<String>
11 {
12   private String propertyName;
13
14   /**
15    * Конструирует действие для поиска заданного свойства
16    * @param propertyName Имя свойства (например, "user.home")
17   */
18   public SysPropAction(String propertyName)
19      { this.propertyName = propertyName; }
20
21   public String run()
22   {
23     return System.getProperty(propertyName);
24   }
25 }
```

Листинг 9.8. Исходный код из файла auth/AuthTest.policy

```
1  grant codebase "file:login.jar"
2  {
3    permission javax.security.auth.AuthPermission "createLoginContext.Login1";
4    permission javax.security.auth.AuthPermission "doAsPrivileged";
5  };
6
7  grant principal com.sun.security.auth.UnixPrincipal "harry"
8  {
9    permission java.util.PropertyPermission "user.*", "read";
10 };
```

Листинг 9.9. Исходный код из файла auth/jaas.config

```
1 Login1
2 {
3   com.sun.security.auth.module.UnixLoginModule required;
4 };
```

javax.security.auth.login.LoginContext 1.4

- `LoginContext(String name)`

  Создает контекст регистрации. Параметр `name` соответствует дескриптору регистрации в конфигурационном файле службы JAAS.

- `void login()`

  Регистрирует субъект, а при неудачном исходе регистрации генерирует исключение типа `LoginException`. Вызывает метод `login()` для диспетчеров, указанных в конфигурационном файле службы JAAS.

- `void logout()`

  Отменяет регистрацию субъекта. Вызывает метод `logout()` для диспетчеров, указанных в конфигурационном файле службы JAAS.

- `Subject getSubject()`

  Возвращает аутентифицированный субъект.

javax.security.auth.Subject 1.4

- `Set<Principal> getPrincipals()`

  Возвращает принципалы данного субъекта.

- `static Object doAs(Subject subject, PrivilegedAction action)`
- `static Object doAs(Subject subject, PrivilegedExceptionAction action)`
- `static Object doAsPrivileged(Subject subject, PrivilegedAction action, AccessControlContext context)`
- `static Object doAsPrivileged(Subject subject, PrivilegedExceptionAction action, AccessControlContext context)`

  Выполняют привилегированное действие от имени субъекта. Возвращают объект, сформированный методом `run()` из класса `PrivilegedAction`. Методы `doAsPrivileged()` выполняют действие в указанном контексте управления доступом. Для них можно указать контекст, полученный ранее в результате вызова метода `AccessController.getContext()`. Если задано пустое значение `null`, код будет выполняться в новом контексте.

java.security.PrivilegedAction 1.4

- `Object run()`

  Этот метод нужно определить самостоятельно для выполнения кода от имени субъекта.

java.security.PrivilegedExceptionAction 1.4

- `Object run()`

  Этот метод нужно определить самостоятельно для выполнения кода от имени субъекта. Он может генерировать любые проверяемые исключения.

java.security.Principal 1.1

- `String getName()`

  Возвращает имя данного принципала.

Модули регистрации в службе JAAS

В этом разделе рассматривается пример применения службы JAAS, демонстрирующий следующее:

- как реализовывать свой собственный модуль регистрации;
- как реализовывать *ролевую* аутентификацию.

Предоставление собственного модуля регистрации бывает полезным в случаях, когда учетные данные сохраняются в базе данных. Даже если вас вполне устраивает стандартный модуль регистрации, изучение процесса реализации специального модуля поможет вам лучше разобраться с предназначением параметров в конфигурационном файле службы JAAS.

Реализация механизма ролевой аутентификации играет важную роль в тех случаях, когда требуется управлять большим количеством пользователей. Размещать имена всех допустимых пользователей в файле правил регистрации непрактично. Вместо этого лучше поступить таким образом, чтобы модуль регистрации сопоставлял пользователей с ролями вроде «`admin`" или "`HR`" и предоставлял им права, исходя из их ролей.

Одна из задач модуля регистрации состоит в заполнении множества принципалов аутентифицируемого субъекта. Если в модуле регистрации поддерживаются роли, он должен также вводить в это множество объекты типа `Principal`, описывающие соответствующие роли. В библиотеке Java отсутствует класс для осуществления этой задачи, поэтому нам пришлось создать собственный класс, исходный код которого приведен в листинге 9.10. Этот класс просто сохраняет пары "описание–значение" типа `role=admin`. А его метод `getName()` возвращает их, что дает возможность вводить ролевые полномочия в файл правил регистрации, как показано ниже.

Рассматриваемый здесь модуль регистрации предусматривает поиск имен, паролей и ролей пользователей в текстовом файле, содержащем приблизительно такие строки:

```
harry|secret|admin
carl|guessme|HR
```

Разумеется, при разработке реального модуля регистрации эти сведения следовало бы хранить в какой-нибудь базе данных или каталоге. Исходный код рассматриваемого здесь примера модуля регистрации типа `SimpleLoginModule` представлен в листинге 9.11. Метод `checkLogin()` проверяет, соответствуют ли имя пользователя

и пароль какой-нибудь записи в файле паролей. Если они соответствуют, то во множество принципалов данного субъекта вводятся два объекта типа `SimplePrincipal`, как показано ниже.

```
Set<Principal> principals = subject.getPrincipals();
principals.add(new SimplePrincipal("username", username));
principals.add(new SimplePrincipal("role", role));
```

Остальная часть модуля типа `SimpleLoginModule` довольно проста. В частности, метод `initialize()` получает в качестве параметров следующее.

- Аутентифицируемый объект типа `Subject`.
- Обработчик для извлечения учетных данных.
- Отображение `sharedState`, которое можно использовать для обмена данными между модулями регистрации.
- Отображение `options`, которое содержит пары "имя–значение", задаваемые при настройке модуля регистрации.

Допустим, рассматриваемый здесь модуль регистрации настраивается приведенным ниже образом. В таком случае этот модуль будет извлекать из отображения `options` параметры настройки `pwfile`.

```
SimpleLoginModule required pwfile="password.txt";
```

Сбором имени пользователя и пароля данный модуль регистрации не занимается, поскольку эта задача поручена отдельному обработчику. Такое разделение ответственности позволяет использовать один и тоже же модуль, не особенно беспокоясь, откуда именно поступают учетные данные: из диалогового окна ГПИ, командной строки консоли или конфигурационного файла. Требующийся обработчик задается при создании экземпляра класса `LoginContext`, как показано в приведенном ниже примере.

```
LoginContext context = new LoginContext("Login1",
    new com.sun.security.auth.callback.DialogCallbackHandler());
```

Обработчик типа `DialogCallbackHandler` открывает простое диалоговое окно ГПИ для запрашивания имени пользователя и пароля, а обработчик типа `com.sun.security.auth.callback.TextCallbackHandler` извлекает эти учетные данные с консоли.

Но в данном примере для получения имени пользователя и пароля используется собственный ГПИ (рис. 9.10). Для этой цели создается специальный обработчик, способный сохранять и возвращать учетные данные, как показано в листинге 9.12.

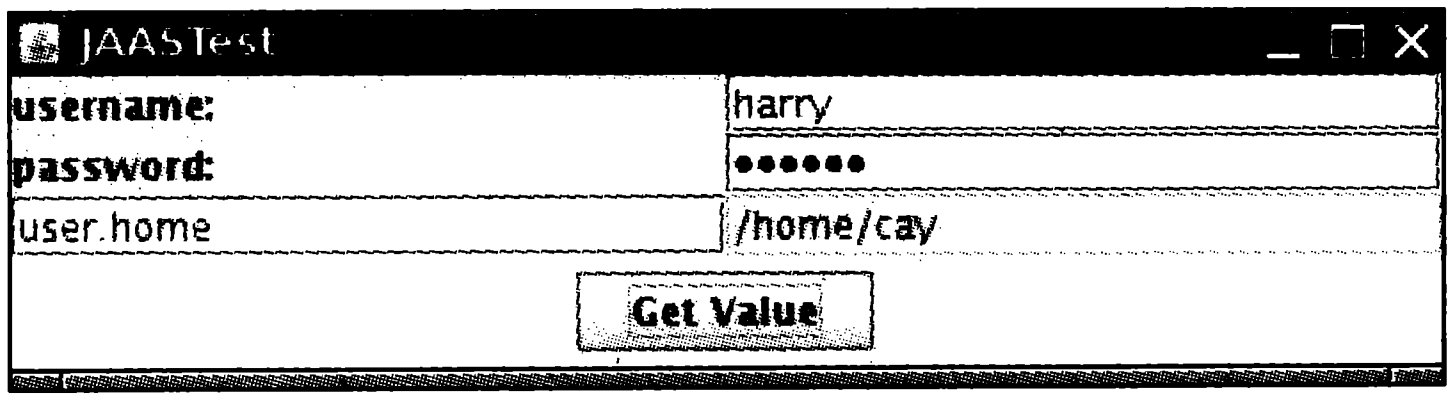


Рис. 9.10. Специальный модуль регистрации

Этот обработчик состоит из единственного метода `handle()`, который обрабатывает массив объектов типа `Callback`. Интерфейс `Callback` реализуется рядом предопределенных классов вроде `NameCallback` и `PasswordCallback`. При желании

можно добавить к ним собственный класс, например RetinaScanCallback. Исходный код этого обработчика выглядит не очень изящно, поскольку нуждается в анализе типа объектов обратного вызова, как показано ниже.

```
public void handle(Callback[] callbacks)
{
  for (Callback callback : callbacks)
  {
    if (callback instanceof NameCallback) . . .
    else if (callback instanceof PasswordCallback) . . .
    else . . .
  }
}
```

Модуль регистрации сначала подготавливает массив объектов типа Callback, которые требуются ему для аутентификации, как следует из приведенного ниже фрагмента кода, а затем просто извлекает из этих объектов всю необходимую информацию.

```
NameCallback nameCall = new NameCallback("username: ");
PasswordCallback passCall = new PasswordCallback("password: ", false);
callbackHandler.handle(new Callback[] { nameCall, passCall });
```

При выполнении рассматриваемого здесь примера программы из листинга 9.13 отображается форма для ввода учетных данных и имени системного свойства. При удачном исходе аутентификации пользователя значение этого свойства извлекается в объект типа PrivilegedAction. Как следует из файла правил регистрации, содержимое которого приведено в листинге 9.14, правами на чтение свойств обладают только пользователи с ролью admin.

Как и в примере программы из предыдущего раздела, код регистрации и код действия требуется разделить. Для этого создаются два JAR-файла следующим образом:

```
javac *.java
jar cvf login.jar JAAS*.class Simple*.class
jar cvf action.jar SysPropAction.class
```

После этого рассматриваемая здесь программа запускается на выполнение по команде

```
java -classpath login.jar:action.jar
   -Djava.security.policy=JAASTest.policy
   -Djava.security.auth.login.config=jaas.config
   JAASTest
```

Содержимое конфигурационного файла приведено в листинге 9.15.

НА ЗАМЕТКУ! Можно также обеспечить поддержку более сложного двухэтапного протокола и с его помощью сделать так, чтобы регистрация считалась *завершенной* только при успешном выполнении всех модулей, предусмотренных в настройке регистрации. Подробнее об этом можно узнать из руководства для разработчиков модулей регистрации, доступного по адресу http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASLMDevGuide.html.

Листинг 9.10. Исходный код из файла jaas/SimplePrincipal.java

```
1  package jaas;
2
3  import java.security.*;
4  import java.util.*;
```

```
5
6  /**
7   * Принципал с именованным значением (например,
8   * "role=HR" или "username=harry")
9  */
10 public class SimplePrincipal implements Principal
11 {
12   private String descr;
13   private String value;
14
15   /**
16    * Конструирует принципал типа SimplePrincipal для
17    * хранения отдельного описания и значения
16    * @param descr Описание
18    * @param value Связанное с ним значение
19   */
20   public SimplePrincipal(String descr, String value)
21   {
22     this.descr = descr;
23     this.value = value;
24   }
25
26   /**
27    * Возвращает имя роли данного принципала
28    * @return Имя роли
29   */
30   public String getName()
31   {
32     return descr + "=" + value;
33   }
34
35   public boolean equals(Object otherObject)
36   {
37     if (this == otherObject) return true;
38     if (otherObject == null) return false;
39     if (getClass() != otherObject.getClass()) return false;
40     SimplePrincipal other = (SimplePrincipal) otherObject;
41     return Objects.equals(getName(), other.getName());
42   }
43
44   public int hashCode()
45   {
46     return Objects.hashCode(getName());
47   }
48 }
```

Листинг 9.11. Исходный код из файла `jaas/SimpleLoginModule.java`

```
1  package jaas;
2
3  import java.io.*;
4  import java.nio.file.*;
5  import java.security.*;
6  import java.util.*;
7  import javax.security.auth.*;
8  import javax.security.auth.callback.*;
9  import javax.security.auth.login.*;
```

```
10  import javax.security.auth.spi.*;
11
12  /**
13   * Этот модуль регистрации аутентифицирует пользователей,
14   * считывая их имена, пароли и роли из текстового файла
15  */
16  public class SimpleLoginModule implements LoginModule
17  {
18    private Subject subject;
19    private CallbackHandler callbackHandler;
20    private Map<String, ?> options;
21
22    public void initialize(Subject subject, CallbackHandler callbackHandler,
23       Map<String, ?> sharedState, Map<String, ?> options)
24    {
25      this.subject = subject;
26      this.callbackHandler = callbackHandler;
27      this.options = options;
28    }
29
30    public boolean login() throws LoginException
31    {
32      if (callbackHandler == null) throw new LoginException("no handler");
33
34      NameCallback nameCall = new NameCallback("username: ");
35      PasswordCallback passCall = new PasswordCallback("password: ", false);
36      try
37      {
38        callbackHandler.handle(new Callback[] { nameCall, passCall });
39      }
40      catch (UnsupportedCallbackException e)
41      {
42        LoginException e2 = new LoginException("Unsupported callback");
43        e2.initCause(e);
44        throw e2;
45      }
46      catch (IOException e)
47      {
48        LoginException e2 = new LoginException("I/O exception in callback");
49        e2.initCause(e);
50        throw e2;
51      }
52
53      try
54      {
55        return checkLogin(nameCall.getName(), passCall.getPassword());
56      }
57      catch (IOException ex)
58      {
59        LoginException ex2 = new LoginException();
60        ex2.initCause(ex);
61        throw ex2;
62      }
63    }
64
65    /**
66     * Проверяет достоверность данных аутентификации. Если они достоверны,
67     * то субъекту требуются принципалы для имени пользователя и его роли
```

```
68      * @param username Имя пользователя
69      * @param password Символьный массив, содержащий пароль
70      * @return Возвращает логическое значение true, если
71      *         данные достоверны
72     */
73    private boolean checkLogin(String username, char[] password)
74       throws LoginException, IOException
75    {
76      try (Scanner in = new Scanner(Paths.get("" + options.get("pwfile"))))
75      {
76        while (in.hasNextLine())
77        {
78          String[] inputs = in.nextLine().split("\\|");
79          if (inputs[0].equals(username) &&
80             Arrays.equals(inputs[1].toCharArray(), password))
81          {
82            String role = inputs[2];
83            Set<Principal> principals = subject.getPrincipals();
84            principals.add(new SimplePrincipal("username", username));
85            principals.add(new SimplePrincipal("role", role));
86            return true;
87          }
88        }
89        return false;
90      }
91    }
92
93    public boolean logout()
94    {
95      return true;
96    }
97
98    public boolean abort()
99    {
100     return true;
101   }
102
103   public boolean commit()
104   {
105     return true;
106   }
107 }
```

Листинг 9.12. Исходный код из файла `aas/SimpleCallbackHandler.java`

```
1  package jaas;
2
3  import javax.security.auth.callback.*;
4
5  /**
6   * Этот простой обработчик обратных вызовов предоставляет
7   * заданные имя пользователя и пароль
8  */
9  public class SimpleCallbackHandler implements CallbackHandler
10 {
11   private String username;
12   private char[] password;
```

```
13
14    /**
15     * Конструирует обработчик обратных вызовов
16     * @param username Имя пользователя
17     * @param password Символьный массив, содержащий пароль
18    */
19    public SimpleCallbackHandler(String username, char[] password)
20    {
21      this.username = username;
22      this.password = password;
23    }
24
25    public void handle(Callback[] callbacks)
26    {
27      for (Callback callback : callbacks)
28      {
29        if (callback instanceof NameCallback)
30        {
31          ((NameCallback) callback).setName(username);
32        }
33        else if (callback instanceof PasswordCallback)
34        {
35          ((PasswordCallback) callback).setPassword(password);
36        }
37      }
38    }
39  }
```

Листинг 9.13. Исходный код из файла `jaas/JAASTest.java`

```
1  package jaas;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  /**
7   * В этой программе сначала производится аутентификация пользователя
8   * через специальную регистрацию, а затем поиск системного свойства
9   * с назначенными для пользователя привилегиями
10  * @version 1.01 2012-06-10
11  * @author Cay Horstmann
12 */
13 public class JAASTest
14 {
15   public static void main(final String[] args)
16   {
17     System.setSecurityManager(new SecurityManager());
18     EventQueue.invokeLater(new Runnable()
19     {
20       public void run()
21       {
22         JFrame frame = new JAASFrame();
23         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24         frame.setTitle("JAASTest");
25         frame.setVisible(true);
26       }
27     });
```

```
28    }
29 }
```

Листинг 9.14. Исходный код из файла `jaas/JAASTest.policy`

```
1  grant codebase "file:login.jar"
2  {
3    permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
4    permission java.awt.AWTPermission "accessEventQueue";
5    permission javax.security.auth.AuthPermission "createLoginContext.Login1";
6    permission javax.security.auth.AuthPermission "doAsPrivileged";
7    permission javax.security.auth.AuthPermission "modifyPrincipals";
8    permission java.io.FilePermission "jaas/password.txt", "read";
9  };
10
11 grant principal jaas.SimplePrincipal "role=admin"
12 {
13   permission java.util.PropertyPermission "*", "read";
14 };
```

Листинг 9.15. Исходный код из файла `jaas/jaas.config`

```
1 Login1
2 {
3   jaas.SimpleLoginModule required pwfile="jaas/password.txt" debug=true;
4 };
```

javax.security.auth.callback.CallbackHandler 1.4

- `void handle(Callback[] callbacks)`

  Обрабатывает указанные объекты типа `Callback`, взаимодействуя при необходимости с пользователем, а затем сохраняет в них информацию, требующуюся для безопасности.

javax.security.auth.callback.NameCallback 1.4

- `NameCallback(String prompt)`
- `NameCallback(String prompt, String defaultName)`

  Создают объект типа `NameCallback` с указанным приглашением и именем по умолчанию.

- `String getName()`
- `void setName(String name)`

  Получают или устанавливают имя, получаемое с помощью текущего объекта типа `Callback`.

- `String getPrompt()`

  Получает приглашение, которое должно использоваться при запрашивании данного имени.

- `String getDefaultName()`

  Получает имя по умолчанию, которое должно использоваться при запрашивании данного имени.

javax.security.auth.callback.PasswordCallback 1.4

- `PasswordCallback(String prompt, boolean echoOn)`

  Создает объект типа `PasswordCallback` с указанным приглашением и признаком режима отображения эхо-символов.

- `char[] getPassword()`
- `void setPassword(char[] password)`

  Получают или устанавливают пароль, получаемый с помощью текущего объекта типа `Callback`.

- `String getPrompt()`

  Получает приглашение, которое должно использоваться при запрашивании данного пароля.

- `boolean isEchoOn()`

  Получает признак режима отображения эхо-символов, который должен использоваться при запрашивании данного пароля.

javax.security.auth.spi.LoginModule 1.4

- `void initialize(Subject subject, CallbackHandler handler, Map<String,?> sharedState, Map<String,?> options)`

  Инициализирует объект типа `LoginModule` для аутентификации указанного субъекта. Для получения учетных данных использует указанный обработчик. Отображение *sharedState* применяется для взаимодействия с другими модулями регистрации, а отображение *options* — для хранения имен и значений, указанных при настройке данного экземпляра модуля регистрации.

- `boolean login()`

  Обеспечивает выполнение процесса регистрации и заполняет принципалы субъекта. Возвращает логическое значение `true`, если регистрация прошла успешно.

- `boolean commit()`

  Если сценарий регистрации предполагает завершение в два этапа, то данный метод вызывается после успешного завершения всех модулей регистрации. Возвращает логическое значение `true`, если операция выполнена успешно.

- `boolean abort()`

  Вызывается, если неудачный исход выполнения другого модуля предполагает отказ от регистрации. Возвращает логическое значение `true`, если операция выполнена успешно.

- `boolean logout()`

  Отменяет регистрацию субъекта. Возвращает логическое значение `true`, если операция выполнена успешно.

Цифровые подписи

Как упоминалось ранее, интерес к платформе Java зародился с аплетов. Программисты сразу поняли, что с практической точки зрения, несмотря на широкие анимационные возможности, аплеты не полностью поддерживают модель безопасности. В версии JDK 1.0 аплеты очень строго контролировались.

С другой стороны, в защищенной корпоративной сети компании риск атаки через аплеты минимален, поэтому логично было бы предоставить аплетам, выполняющимся в такой сети, дополнительные права. Разработчики из компании Sun Microsystems быстро осознали преимущества, которые дает применение аплетов, и поэтому решили предоставить пользователям возможность присваивать аплетам *разные* уровни защиты в зависимости от их происхождения. Так, если аплет поступает от надежного, заслуживающего доверия поставщика, то ему можно предоставить более обширные права доступа.

Для предоставления аплету дополнительных полномочий необходимо знать ответы на следующие два вопроса.

1. Откуда поступил аплет?
2. Не был ли его код поврежден во время передачи?

За последние 50 лет специалисты в области математики и информатики разработали немало сложных алгоритмов поддержки целостности данных и цифровых подписей. Многие из них реализованы в пакете `java.security`. Для применения таких алгоритмов совсем не обязательно понимать математические принципы, положенные в их основу. В последующих разделах описывается механизм свертки сообщений, позволяющий обнаружить факт изменения в документе, а также показывается, каким образом цифровая подпись способна подтверждать личность подписавшегося.

Свертки сообщений

Свертка сообщения — это цифровой "отпечаток" блока данных. Например, алгоритм безопасного хеширования SHA-1 (Secure Hash Algorithm #1) уплотняет любой блок данных в последовательность из 160 бит (20 байт). По аналогии с отпечатками пальцев, считается, что не существует двух одинаковых цифровых отпечатков по алгоритму SHA-1. На самом деле это не так, поскольку алгоритм SHA-1 поддерживает только 2^{160} отпечатков. Следовательно, теоретически они могут совпасть. Но число 2^{160} настолько велико, что вероятность дублирования цифровых отпечатков очень мала, но насколько? Как утверждается в книге *True Odds: How risks Affect Your Everyday Life* Джеймса Уолша (James Walsh; издательство Merritt Publishing, 1996 г.), вероятность смерти от удара молнии составляет **1/30000**. Если подсчитать вероятность такого же исхода для 10 человек (выбранных, например, злобным врагом), она окажется гораздо выше, чем вероятность наличия двух одинаковых цифровых отпечатков по алгоритму SHA-1. (Конечно, от удара молнии погибло гораздо больше, чем 10 человек, но здесь речь идет о специально выбранной группе людей.)

Свертка сообщения обладает двумя важными свойствами.

- Если изменяется один или несколько битов данных, то изменяется и свертка сообщения.

- Исходное сообщение нельзя изменить таким образом, чтобы полученное поддельное сообщение имело такую же свертку, как у исходного сообщения.

Второе свойство, конечно, соблюдается с определенной степенью вероятности. Допустим, некий миллиардер составил следующее завещание:

"После смерти мое имущество должно быть разделено между моими детьми, но мой сын Джордж ничего не получит".

Отпечаток данного сообщения по алгоритму SHA-1 имеет такой вид:

```
12 5F 09 03 E7 31 30 19 2E A6 E7 E4 90 43 84 B4 38 99 8F 67
```

Предположим, недоверчивый миллиардер отдал текст завещания одному адвокату, а его отпечаток — другому. Допустим далее, что Джордж подкупил адвоката, у которого хранится завещание, чтобы заменить слово **George** на слово **Bill**. В этом случае цифровой отпечаток по алгоритму SHA-1 поддельного завещания будет отличаться от аналогичного отпечатка исходного завещания:

```
7D F6 AB 08 EB 40 EC CD AB 74 ED E9 86 F9 ED 99 D1 45 B1 57
```

Может ли Джордж составить такое поддельное завещание, которое имело бы отпечаток оригинального завещания? Не может, потому что для перебора всех вариантов ему не хватило бы времени существования Земли, даже если бы он был счастливым обладателем миллиарда компьютеров, которые способны перебирать миллион вариантов завещания в секунду.

Для вычисления таких сверток сообщений был разработан целый ряд алгоритмов. Двумя наиболее известными из них являются: алгоритм SHA-1, разработанный Национальным институтом стандартов и технологий (National Institute of Standards and Technology — NIST), и алгоритм MD5, изобретенный Рональдом Райвестом (Ronald Rivest) из Массачусетского технологического института (Massachusetts Institute of Technologies — MIT). Оба эти алгоритма способны шифровать фрагменты сообщений своим оригинальным способом. Подробнее с ними можно ознакомиться, например, в книге *Cryptography and Network Security, 5th Edition* Вильяма Столлингса (William Stallings; издательство Prentice Hall, 2011 г.). Следует, однако, иметь в виду, что недавно в обоих алгоритмах были обнаружены незначительные изъяны. В настоящее время многие специалисты по шифрованию рекомендуют избегать алгоритма MD5 и пользоваться алгоритмом SHA-1 до тех пор, пока не появится какой-нибудь более надежный альтернативный вариант. (Подробнее об этом можно узнать по адресу `www.rsa.com/rsalabs/node.asp?id=2834`.)

В языке программирования Java реализованы оба упомянутых выше алгоритма. В частности, класс `MessageDigest` представляет собой *фабрику* для создания объектов, инкапсулирующих алгоритмы получения цифровых отпечатков. Этот класс содержит статический метод `getInstance()`, возвращающий экземпляр подкласса, производного от класса `MessageDigest`. Поэтому класс `MessageDigest` может выступать в роли фабричного класса или суперкласса для всех алгоритмов получения свертки сообщения. В приведенном ниже примере показано, каким образом получается объект для вычисления цифровых отпечатков по алгоритму SHA-1.

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

После создания объекта типа `MessageDigest` ему нужно передать все байты сообщения, повторно вызывая метод `update()`. Так, в приведенном ниже фрагменте

кода содержимое файла передается полученному выше объекту `alg`, формирующему цифровой отпечаток.

```
InputStream in = . . .
int ch;
while ((ch = in.read()) != -1)
  alg.update((byte) ch);
```

С другой стороны, если байты размещаются в массиве, метод `update()` можно применить ко всему массиву следующим образом:

```
byte[] bytes = . . .;
alg.update(bytes);
```

Далее следует вызвать метод `digest()`, который дополняет вводимые данные недостающими битами (необходимое условие для алгоритма получения цифровых отпечатков), вычисляет цифровой отпечаток и возвращает свертку сообщения в виде следующего байтового массива:

```
byte[] hash = alg.digest();
```

В примере программы из листинга 9.16 свертка сообщения вычисляется по алгоритму SHA или MD5. Для того чтобы запустить ее на выполнение, введите команду

```
java hash.Digest hash/input.txt
```

или

```
java hash.Digest hash/input.txt MD5
```

Листинг 9.16. Исходный код из файла `hash/Digest.java`

```
1  package hash;
2
3  import java.io.*;
4  import java.nio.file.*;
5  import java.security.*;
6
7  /**
8   * В этой программе вычисляется свертка сообщения из файла
9   * @version 1.20 2012-06-16
10  * @author Cay Horstmann
11 */
12 public class Digest
13 {
14   /**
15    * @param args args[0] - имя файла, args[1] - дополнительно
16    * алгоритм (SHA-1 или MD5)
17   */
18   public static void main(String[] args)
19      throws IOException, GeneralSecurityException
20   {
21     String algname = args.length >= 2 ? args[1] : "SHA-1";
22     MessageDigest alg = MessageDigest.getInstance(algname);
23     byte[] input = Files.readAllBytes(Paths.get(args[0]));
24     byte[] hash = alg.digest(input);
25     String d = "";
26     for (int i = 0; i < hash.length; i++)
27     {
28       int v = hash[i] & 0xFF;
```

```
29         if (v < 16) d += "0";
30         d += Integer.toString(v, 16).toUpperCase() + " ";
31       }
32       System.out.println(d);
33     }
34 }
```

java.security.MessageDigest 1.1

- `static MessageDigest getInstance(String` *algorithmName*`)`

  Возвращает объект типа `MessageDigest`, реализующий указанный алгоритм. Если такой алгоритм не поддерживается, генерирует исключение типа `NoSuchAlgorithmExcetion`.

- `void update(byte` *input*`)`
- `void update(byte[]` *input*`)`
- `void update(byte[]` *input*`, int` *offset*`, int` *len*`)`

  Обновляют свертку сообщения, используя указанные байты.

- `byte[] digest()`

  Вычисляет свертку сообщения по алгоритму хеширования, возвращает вычисленную свертку и устанавливает объект алгоритма в исходное состояние.

- `void reset()`

  Устанавливает объект свертки сообщения в исходное состояние.

Подписание сообщений

В предыдущем разделе было показано, как создавать свертку сообщения, т.е. делать своего рода дактилоскопический отпечаток исходного сообщения. При изменении сообщения цифровой отпечаток измененного сообщения не будет совпадать с отпечатком исходного. Это означает, что при доставке сообщения вместе с его цифровым отпечатком получатель сможет проверить, не было ли оно подделано. Но если злоумышленнику удастся перехватить как сообщение, так и его исходный отпечаток, то он сможет легко изменить сообщение и переделать этот отпечаток так, как ему нужно. В конце концов, алгоритмы получения сверток сообщений всем известны и не требуют использования секретных ключей. В таком случае получатель поддельного сообщения и переделанного цифрового отпечатка так и не узнает, что его сообщение было изменено. Этот недостаток позволяют устранить цифровые подписи.

Для того чтобы стал понятнее принцип действия цифровых подписей, нужно сначала ввести некоторые понятия из области *шифрования открытым ключом*. Основными в этой области являются такие понятия, как *открытый ключ* и *секретный ключ*. Открытый ключ сообщается всем, а секретный держится в строгом секрете. Соответствие между этими ключами устанавливается на основании математических отношений, которые здесь не рассматриваются. (Тем, кому интересно узнать о них более подробно, рекомендуется книга *The Handbook of Applied Cryptography*, которая доступна по адресу `http://www.cacr.math.uwaterloo.ca/hac/`.)

Эти ключи довольно длинные и сложные. В качестве примера ниже приведена пара открытого и секретного ключей, полученных с помощью алгоритма DSA (Digital Signature Algorithm — алгоритм создания цифровых подписей).

Открытый ключ:

```
p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617a
e01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e
35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4

y: c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a8123
ce5a8018b8161a760480fadd040b927281ddb22cb9bc4df596d7de4d1b977d50
```

Соответствующий ему секретный ключ:

```
p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01
f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b
71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4

x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a
```

Считается, что вывести один ключ из другого практически невозможно. Несмотря на то что открытый ключ известен всем, узнать с его помощью секретный ключ злоумышленникам никогда не удастся, сколько бы вычислительных ресурсов ни было в их распоряжении.

В то, что никому не удастся вычислить секретный ключ на основе открытого ключа, трудно поверить, но, по крайней мере, пока еще никто не изобрел алгоритм, способный делать нечто подобное с ключами, генерируемыми с помощью самых распространенных в настоящее время алгоритмов шифрования. Из-за того что эти ключи достаточно длинные, для их расшифровки методом "грубой силы", т.е. путем простого перебора всех возможных вариантов, потребуется больше компьютеров, чем может быть вообще создано из всех атомов солнечной системы, и не одна тысяча лет. Разумеется, не исключено, что кому-нибудь удастся придумать для разгадывания ключей шифрования более эффективный алгоритм, чем простой перебор возможных вариантов.

Рассмотрим в качестве примера алгоритм шифрования RSA, изобретенный Райвестом (Rivest), Шамиром (Shamir) и Адлеманом (Adleman). Принцип действия этого алгоритма основывается на сложности разложения больших чисел на простые множители. За последние двадцать лет многие известные математики пытались разработать удачные алгоритмы разложения чисел на простые множители, но добиться этого пока еще никому не удалось. Из-за этого большинство специалистов по шифрованию считают, что в настоящее время ключи длиной 2000 битов являются абсолютно защищенными от любых попыток взлома. Алгоритм шифрования DSA также считается довольно надежным. На рис. 9.11 показано, как выглядит процесс шифрования по алгоритму DSA на практике.

Допустим, Алиса хочет послать сообщение Бобу, а Боб желает быть уверенным в том, что сообщение поступило именно от Алисы, а не от какого-то самозванца. Для этого Алиса составляет сообщение и *подписывает* свертку этого сообщения с помощью своего секретного ключа. Далее Боб получает копию ее открытого ключа и применяет его для *верификации* подписи. При удачном завершении процесса верификации Боб может быть уверен в следующем:

- исходное сообщение не было изменено;
- сообщение было подписано Алисой, обладающей секретным ключом, соответствующим тому открытому ключу, который Боб использовал для верификации.

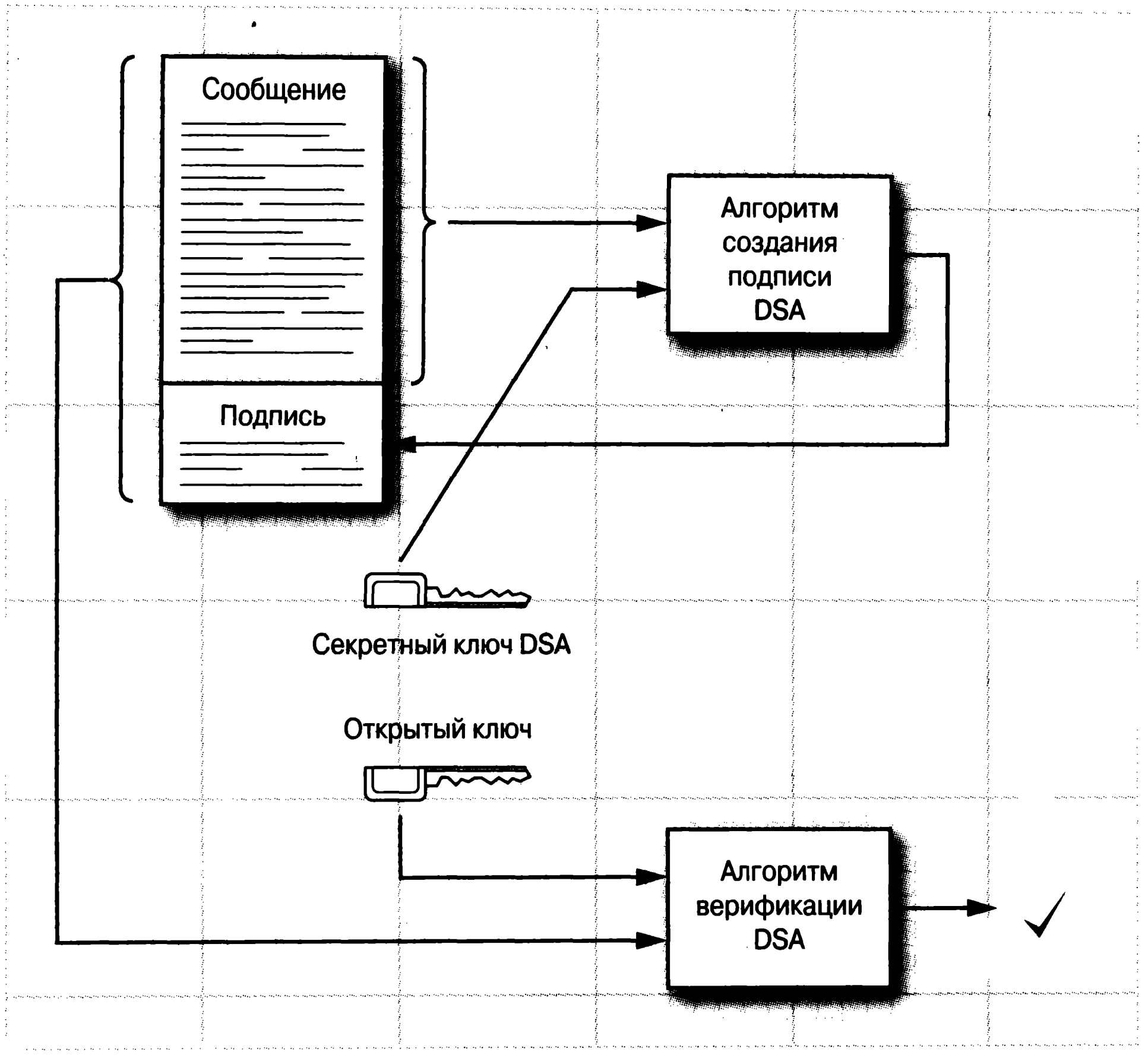


Рис. 9.11. Обмен сообщениями с использованием цифровой подписи, открытых ключей и алгоритма шифрования DSA

Из этого примера ясно видно, почему так важны секретные ключи. Если кто-нибудь выкрадет секретный ключ Алисы или если правительство попросит ее расшифровать его, ей грозят неприятности, потому что выкравший ключ злоумышленник или представитель правительства сможет легко подделывать ее сообщения, заявки на пересылку денежных средств и выполнять другие подобные операции, а все будут считать, что все это делает сама Алиса.

Верификация подписи

В состав JDK входит утилита `keytool` командной строки, предназначенная для генерирования сертификатов и управления ими. Пока что она способна работать

только в режиме командной строки, но можно надеяться, что в последующих выпусках JDK ее функциональные возможности будут доступны и в виде других, более дружественных к пользователю версиях. В этом разделе она используется для того, чтобы показать, каким образом Алиса может подписывать документ и отправлять его Бобу, а Боб — проверять и удостоверяться в том, что документ был действительно подписан Алисой, а не каким-нибудь самозванцем.

Утилита `keytool` позволяет управлять *хранилищами ключей*, базами данных сертификатов и парами секретных и открытых ключей. У каждой записи в хранилище ключей имеется свой *псевдоним*. Ниже показано, каким образом Алиса может создать хранилище `alice.store` и сгенерировать пару ключей с псевдонимом `alice`, воспользовавшись утилитой `keytool`.

```
keytool -genkeypair -keystore alice.certs -alias alice
```

При создании или открытии хранилища появляется приглашение ввести пароль. В данном примере вводится простое слово **`secret`**. Но при создании хранилища ключей с помощью утилиты `keytool` для каких-нибудь более серьезных целей рекомендуется выбрать надежный пароль и хранить его в полном секрете. При генерировании ключа появляется приглашение ввести следующую информацию:

```
Enter keystore password: secret
(Введите пароль для хранилища ключей)
Reenter new password: secret
(Еще раз введите новый пароль)
What is your first and last name?
(Ваше имя и фамилия [неизвестно])
[Unknown]: Alice Lee
What is the name of your organizational unit?
(Название подразделения вашей организации [неизвестно])
[Unknown]: Engineering Department
What is the name of your organization?
(Название вашей организации [неизвестно])
[Unknown]: ACME Software
What is the name of your City or Locality?
(Название вашего города или местности [неизвестно])
[Unknown]: San Francisco
What is the name of your State or Province?
(Название ваше штата или провинции [неизвестно])
[Unknown]: CA
What is the two-letter country code for this unit?
(Введите двухбуквенный код страны для данного подразделения [неизвестно])
[Unknown]: US
Is <CN=Alice Lee, OU=Engineering Department, O=ACME Software, L=San
Francisco, ST=CA, C=US> correct?
(Все правильно?)
[no]: yes
```

В утилите `keytool` для идентификации владельцев ключей и создателей сертификатов используются имена по стандарту X.500 с составляющими для указания общего имени (Common Name — CM), организационного подразделения (Organizational Unit — OU), организации (Organization — O), местонахождения (Location — L), штата (State — ST) и страны (Country — C). И в завершение требуется указать пароль для ключа или нажать клавишу <Enter>, чтобы использовать в его качестве пароль хранилища ключей.

Допустим, Алисе требуется предоставить Бобу копию своего открытого ключа. В таком случае ей нужно сначала экспортировать файл сертификата следующим образом:

```
keytool -exportcert -keystore alice.certs -alias alice -file alice.cer
```

После этого она может отправить этот сертификат Бобу. Боб же, получив этот сертификат, может распечатать его по следующей команде:

```
keytool -printcert -file alice.cer
```

Ниже приведен пример такой распечатки. Если Боб захочет проверить, тот ли сертификат он получил, он может позвонить Алисе и сверить свою копию с исходным цифровым отпечатком сертификата по телефону.

```
Owner: CN=Alice Lee, OU=Engineering Department, O=ACME Software,
       L=San Francisco, ST=CA, C=US
Issuer: CN=Alice Lee, OU=Engineering Department, O=ACME Software,
        L=San Francisco, ST=CA, C=US
Serial number: 470835ce
Valid from: Sat Oct 06 18:26:38 PDT 2007 until: Fri Jan 04 17:26:38 PST 2008
Certificate fingerprints:
         MD5: BC:18:15:27:85:69:48:B1:5A:C3:0B:1C:C6:11:B7:81
         SHA1: 31:0A:A0:B8:C2:8B:3B:B6:85:7C:EF:C0:57:E5:94:95:61:47:6D:34
         Signature algorithm name: SHA1withDSA
         Version: 3
```

НА ЗАМЕТКУ! Некоторые создатели сертификатов публикуют цифровые отпечатки своих сертификатов на веб-сайтах. Например, чтобы проверить сертификат компании VeriSign из каталога хранилища ключей *jre*/lib/security/cacerts, можно сначала отобразить содержимое этого каталога по следующей команде с параметром **-list**:

```
keytool -list -v -keystore jre/lib/security/cacerts
```

Паролем для доступа к данному хранилищу является слово **changeit**. А один из сертификатов в нем будет выглядеть так:

```
Owner: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. -
       For authorized use only",
OU=Class 1 Public Primary Certification Authority - G2,
       O="VeriSign, Inc.", C=US
Issuer: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. -
       For authorized
use only", OU=Class 1 Public Primary Certification Authority - G2,
       O="VeriSign, Inc.", C=US
Serial number: 4cc7eaaa983e71d39310f83d3a899192
Valid from: Sun May 17 17:00:00 PDT 1998 until: Tue Aug 01 16:59:59 PDT
2028
Certificate fingerprints:
          MD5: DB:23:3D:F9:69:FA:4B:B9:95:80:44:73:5E:7D:41:83
          SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
```

Чтобы удостовериться в том, что данный сертификат является действительным, достаточно посетить веб-сайт компании VeriSign по адресу `http://www.verisign.com/repository/root.html`.

Убедившись в правильности сертификата, Боб может далее импортировать его в свое хранилище ключей по следующей команде:

```
keytool -importcert -keystore bob.certs -alias alice -file alice.cer
```

ВНИМАНИЕ! Ни в коем случае не следует импортировать непроверенный сертификат в хранилище ключей. Ведь после ввода сертификата в хранилище ключей любая программа, пользующаяся хранилищем ключей, предполагает, что данный сертификат можно использовать для проверки цифровых подписей.

После этого Алиса может приступить к отправке Бобу подписанных документов. Подписывать и проверять JAR-файлы можно с помощью утилиты `jarsigner`. Сначала Алисе следует просто ввести подписываемый документ в JAR-файл, как показано ниже.

```
jar cvf document.jar document.txt
```

Далее с помощью утилиты `jarsigner` она может добавить к этому файлу свою цифровую подпись. Для этого ей достаточно указать хранилище ключей, JAR-файл и псевдоним используемого ключа в следующей команде:

```
jarsigner -keystore alice.certs document.jar alice
```

Получив этот файл, Боб может использовать для его верификации утилиту `jarsigner` с параметром **`-verify`**:

```
jarsigner -verify -keystore bob.certs document.jar
```

Указывать псевдоним ключа при этом Бобу не нужно. Утилита `jarsigner` сама отыщет в цифровой подписи указанное в формате X.509 имя владельца ключа и произведет поиск соответствующих ему сертификатов в хранилище ключей.

Если JAR-файл не поврежден и цифровая подпись совпадает, то после этого утилита `jarsigner` отобразит приведенное ниже сообщение. В противном случае вместо него появится сообщение об ошибке:

```
jar verified.
```

Проблема аутентификации

Допустим, вы получаете сообщение от своей подруги Алисы, которая подписала его описанным выше способом, используя свой секретный ключ. У вас уже может иметься копия ее открытого ключа, а иначе вам нетрудно будет получить ее у самой Алисы или на ее веб-странице. Наличие этого ключа позволяет легко удостовериться, что данное сообщение действительно написала Алиса и что оно не было кем-то подделано. А теперь допустим, что вы получаете сообщение от какого-то незнакомца, который представляется сотрудником известной компании по разработке программного обеспечения и предлагает вам воспользоваться прикрепленной им к данному сообщению программой. Этот незнакомец даже присылает вам копию своего открытого ключа, чтобы вы могли удостовериться, что именно он является автором данного сообщения. Вы проверяете его и убеждаетесь, что подпись действительна. Это подтверждает лишь то, что сообщение было подписано с помощью соответствующего секретного ключа и что оно не было подделано.

Но имейте в виду, что *о самом авторе сообщения вам все равно ничего не известно*! Сгенерировать секретный и открытый ключи, подписать сообщение с помощью секретного ключа и отправить его вместе с подходящим открытым ключом мог кто угодно. Выяснение личности отправителя называется *проблемой аутентификации*.

Обычно проблема аутентификации разрешается очень просто. Например, у вас и у отправителя может быть общий знакомый, которому вы оба доверяете. Тогда отправитель может встретиться с вашим знакомым лично и передать ему диск

с открытым ключом. Дальше ваш знакомый может встретиться с вами, подтвердить, что он действительно знает отправителя и что тот действительно работает на известную компанию по разработке программного обеспечения, после чего передать вам диск с открытым ключом (рис. 9.12). В таком случае получается, что гарантом аутентичности отправителя будет ваш знакомый.

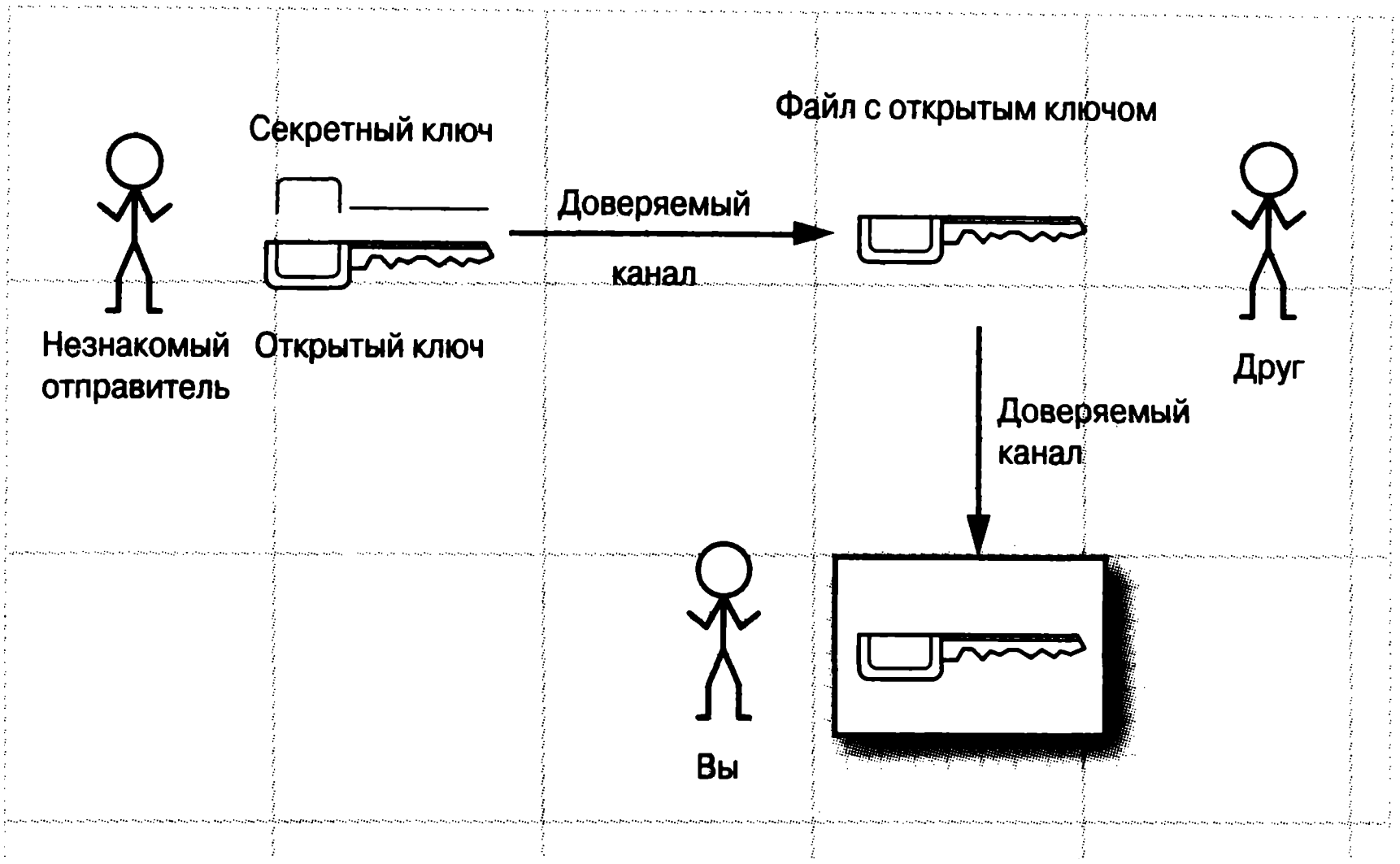


Рис. 9.12. Аутентификация через доверенного посредника

На самом деле вашему знакомому совсем не обязательно встречаться с вами. Вместо этого он может просто подписать файл с открытым ключом незнакомца с помощью своего секретного ключа и послать его вам (рис. 9.13). При получении файла с открытым ключом вы можете проверить подпись своего друга. А поскольку вы ему доверяете, то у вас не сомнений, что он проверил учетные данные незнакомца, прежде чем добавлять свою цифровую подпись.

Но у вас может и не быть такого общего знакомого. В некоторых моделях доверительных отношений предполагается, что всегда существует некая "цепочка доверия", т.е. цепочка общих знакомых, всем звеньям которой можно доверять. Но на практике так бывает далеко не всегда. Вы можете доверять своей подруге Алисе и знать, что она доверяет Бобу, но сами-то вы Боба не знаете, а следовательно, не до конца уверены, стоит ли ему доверять. Другая модель доверительных отношений предполагает наличие какого-то одного хорошо зарекомендовавшего себя человека или организации, которой все могут безоговорочно доверять. Наиболее известным примером такой организации служит компания Verisign, Inc. (`www.verisign.com`).

Вам часто будут встречаться цифровые подписи, заверенные одной или несколькими ручающимися за аутентичность организациями, и поэтому придется самостоятельно оценивать, насколько им стоит доверять. Вы можете доверять компании VeriSign, например, потому, что часто встречаете ее логотип на многих веб-страницах, или потому, что слышали, будто бы в ней применяются невероятные меры безопасности.

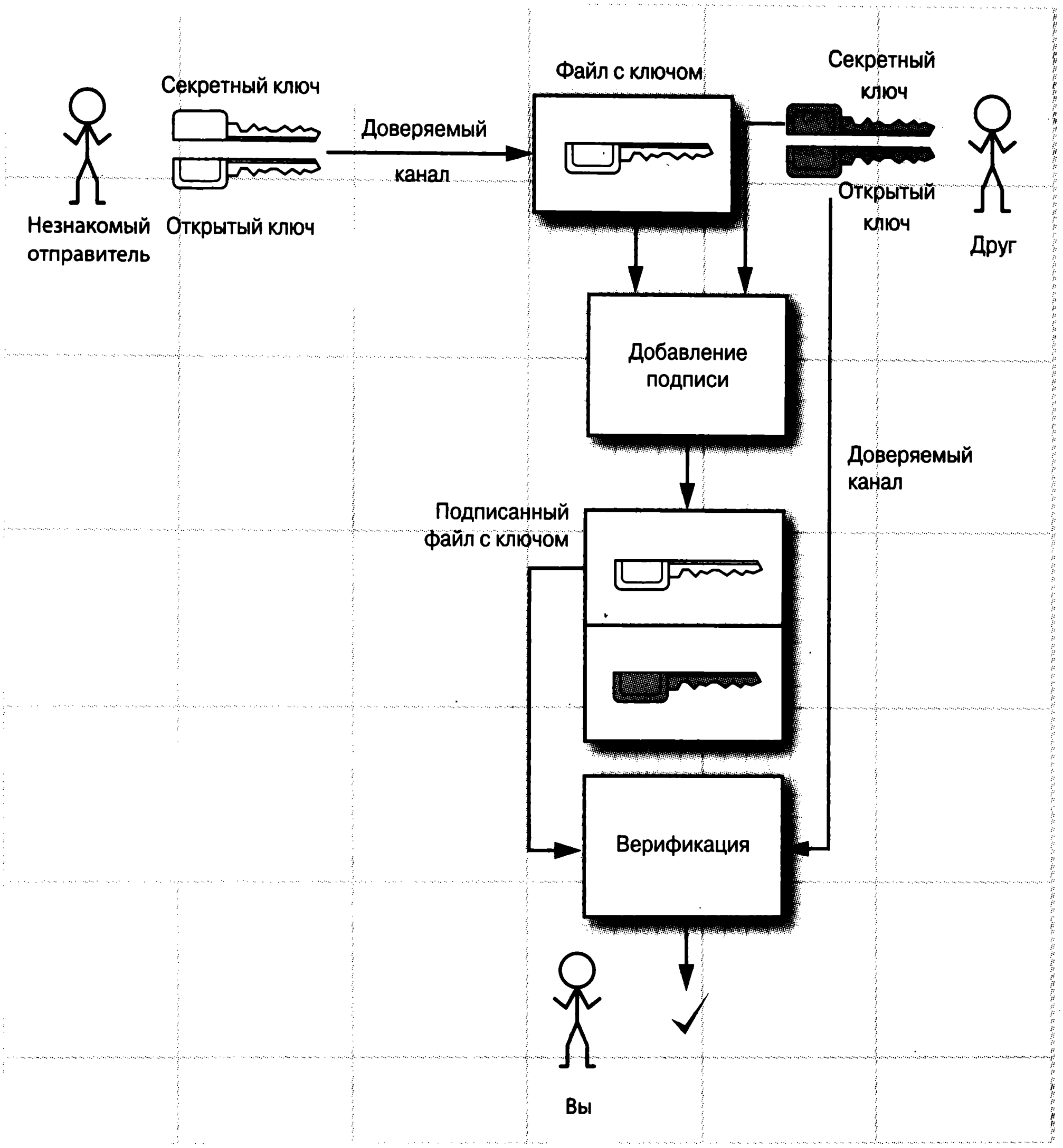


Рис. 9.13. Аутентификация через подпись доверенного посредника

Но вы все равно должны иметь ясное представление, каким образом в подобных организациях производится аутентификация и что именно они проверяют. Что касается компании VeriSign, то ее исполнительный директор, естественно, не встречается лично с каждым человеком или представителем компании при аутентификации ключа. Получить идентификатор "класса 1" от этой компании можно, заполнив веб-форму и заплатив небольшой взнос. В этой форме нужно указать имя, организацию, страну и адрес электронной почты. После заполнения формы по указанному адресу отсылается ключ или инструкции, как его получить. Таким образом, уверенным можно быть только в подлинности электронного адреса, потому что имя и название организации не проверяются и могут быть любыми. Существуют и более строгие классы идентификаторов, например, при желании получить идентификатор

"класса 3" компания VeriSign потребует предоставить отчет о финансовом положении, официально заверенный у нотариуса. У других организаций, занимающихся аутентификацией, будут другие требования. Поэтому очень важно, чтобы при получении заверенного цифровой подписью сообщения вы ясно понимали, каким образом проверялась его подлинность.

Подписание сертификатов

Ранее в разделе "Верификация подписи" было показано, как Алиса использовала самостоятельно подписываемый сертификат для передачи открытого ключа Бобу. Но Бобу пришлось проверить подлинность этого сертификата путем его сверки с исходным цифровым отпечатком от Алисы.

Допустим теперь, что Алисе требуется послать подписанное сообщение своей коллеге Синди, а та не любит тратить время на сверку цифровых подписей с исходными отпечатками. В этом случае Синди может помочь посредническая организация, которой она могла бы доверить проверку всех цифровых подписей. В рассматриваемом здесь примере предполагается, что такой организацией является отдел информационных ресурсов компании ACME Software.

Этот отдел занимается выдачей сертификатов, т.е. представляет собой своего рода *центр сертификации*. У каждого сотрудника компании ACME Software имеется полученный из этого отдела открытый ключ, который находится в хранилище ключей и перед установкой был тщательно сверен с цифровым отпечатком системным администратором. Этот центр сертификации занимается также подписанием ключей всех сотрудников компании ACME Software. Благодаря этому хранилище ключей автоматически доверяет устанавливаемым ключам сотрудников данной компании, поскольку они подписаны с помощью доверяемого ключа.

Ниже поясняется, как сымитировать этот процесс. Итак, создайте хранилище ключей `acmesoft.certs`. Затем сгенерируйте пару ключей и экспортируйте открытый ключ по следующим командам:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer
```

Открытый ключ экспортируется в "самозаверенный" сертификат. Далее введите его в хранилище ключей каждого сотрудника следующим образом:

```
keytool -importcert -keystore cindy.certs -alias acmeroot -file acmeroot.cer
```

Чтобы иметь возможность отправлять подписанные сообщения Синди и любым другим сотрудникам компании ACME Software, Алисе нужно принести свой сертификат в отдел информационных ресурсов данной компании и подписать его. Утилита `keytool`, к сожалению, не предоставляет никаких средств для решения подобной задачи. Для восполнения этого пробела в загружаемом коде к данной книге предлагается класс `CertificateSigner`. В этом случае уполномоченному сотруднику компании ACME Software остается только проверить личность Алисы и сформировать подписанный сертификат по следующей команде:

```
java CertificateSigner -keystore acmesoft.certs -alias acmeroot
     -infile alice.cer -outfile alice_signedby_acmeroot.cer
```

Очевидно, что выполнение такой команды связано с известным риском и требует, чтобы у применяемой для подписания сертификата программы был доступ к хранилищу ключей компании ACME Software, а у уполномоченного сотрудника — необходимый для этого пароль.

После этого Алиса может передать файл `alice_signedby_acmeroot.cer` Синди и любому другому сотруднику ACME Software. В качестве альтернативного варианта этот файл может быть также сохранен в каталоге компании. Напомним, что в этом файле содержится открытый ключ Алисы и подтверждение компании ACME Software, что данный ключ действительно принадлежит Алисе. Далее Синди может импортировать подписанный сертификат Алисы в свое хранилище ключей по следующей команде:

```
keytool -importcert -keystore cindy.certs -alias alice -file alice_signedby_
acmeroot.cer
```

При выполнении этой команды в хранилище ключей будет автоматически проверено, был ли данный сертификат подписан с помощью уже присутствующего в нем доверяемого корневого ключа. Благодаря этому Синди *не* придется сверять его с исходным цифровым отпечатком данного сертификата. Введя один раз корневой сертификат и сертификаты всех остальных лиц, часто присылающих ей документы, Синди избавляется от необходимости беспокоиться впредь о хранилище ключей.

Запросы сертификатов

В предыдущем разделе был рассмотрен пример имитации центра сертификации с хранилищем ключей и классом `CertificateSigner`. Но в большинстве центров сертификации для управления сертификатами применяется более сложное программное обеспечение и несколько другие форматы сертификатов. В этом разделе обсуждаются дополнительные действия, которые требуется выполнить для организации нормального взаимодействия с подобным программным обеспечением.

Выберем для примера пакет программного обеспечения OpenSSL. Это программное обеспечение, как правило, предварительно устанавливается во многих системах Linux и Mac OS X вместе с портом Cygwin. Если оно не установлено, его можно без особого труда загрузить по адресу `http://www.openssl.org`.

Чтобы создать центр сертификации, следует запустить сценарий `CA`. Точное месторасположение этого сценария зависит от конкретной операционной системы. Так, в ОС Ubuntu его можно запустить по следующей команде:

```
/usr/lib/ssl/misc/CA.pl -newca
```

Этот сценарий создаст в текущем каталоге подкаталог `demoCA`, содержащий пару корневых ключей и хранилище для сертификатов и списков аннулирования сертификатов.

Далее необходимо импортировать открытый ключ в хранилище ключей, созданное на Java для всех сотрудников. Но из-за того что этот ключ имеет формат PEM (Privacy Enhanced Mail — почта повышенной секретности), а не легко распознаваемый хранилищем ключей формат DER, сначала придется скопировать файл `demoCA/cacert.pem` в файл `acmeroot.pem`, открыть его в текстовом редакторе и удалить все, что находится перед строкой `-----BEGIN CERTIFICATE-----` и после строки `-----END CERTIFICATE-----`.

После этого файл `acmeroot.pem` можно скопировать в каждое хранилище ключей обычным способом, как показано ниже. Невероятно, но факт: утилита `keytool` не способна выполнить сама эту операцию редактирования.

```
keytool -importcert -keystore cindy.certs -alias alice -file acmeroot.pem
```

Чтобы подписать открытый ключ Алисы, нужно сначала сформировать *запрос сертификата* в формате PEM по следующей команде:

```
keytool -certreq -keystore alice.store -alias alice -file alice.pem
```

Далее для подписания этого сертификата нужно выполнить следующую команду:

```
openssl ca -in alice.pem -out alice_signedby_acmeroot.pem
```

После этого, как и прежде, нужно удалить из файла `alice_signedby_acmeroot.pem` все строки, которые находятся за пределами маркеров `BEGIN CERTIFICATE/END CERTIFICATE`, и затем импортировать этот файл в хранилище ключей по приведенной ниже команде. Аналогичным образом подписать сертификат можно и в каком-нибудь другом общедоступном центре сертификации, подобном компании VeriSign.

```
keytool -importcert -keystore cindy.certs -alias alice
        -file alice_signedby_acmeroot.pem
```

Подписание кода

Одним из самых важных применений технологии аутентификации служит подписание исполняемых программ. Копируя программу из сети, пользователь вполне обоснованно может потребовать гарантий ее подлинности, поскольку такая программа может нанести вред, если она заражена каким-нибудь вирусом. Поэтому требуется полная уверенность в том, что программа предоставлена надежным источником и во время пересылки не была изменена. Если программа написана на Java, то, обладая подобными сведениями об ее источнике, можно принять взвешенное решение о делегировании ей соответствующих полномочий. В частности, программу можно запустить в "песочнице" как обычный аплет или предоставить ей дополнительные права и наложить на нее некоторые ограничения. Так, если скопированная из Интернета программа предназначена для редактирования текста, то ей можно разрешить обращаться к печатающему устройству и файлам из определенного каталога. В то же время этой программе можно запретить доступ к сетевым соединениям, чтобы она не могла без специального разрешения рассылать файлы посторонним лицам. Реализовать эту сложную схему можно описанными ранее средствами и способам обеспечения безопасности.

Подписание JAR-файлов

В этом разделе поясняется, как подписывать аплеты и приложения Web Start для использования вместе с подключаемым модулем Java Plug-In, на примере двух сценариев доставки:

- по внутренней корпоративной сети;
- через Интернет.

В первом сценарии системный администратор устанавливает файлы правил защиты и сертификаты на локальных компьютерах. После этого подключаемый модуль Java Plug-in обращается к файлу правил защиты для проверки полномочий при каждой загрузке подписанного кода, а к хранилищу ключей — для проверки подписей. Процесс установки сертификатов и правил защиты выглядит довольно просто и может быть выполнен только один раз для каждой настольной системы. По его завершении конечные пользователи получают возможность запускать подписанный

корпоративный код за пределами "песочницы". При создании каждой новой программы или обновлении какой-нибудь существующей программы она должна обязательно подписываться и развертываться на веб-сервере. Но изменять настольные системы при усовершенствовании программ не требуется. На наш взгляд, такой сценарий вполне приемлем и может стать довольно привлекательной альтернативой развертыванию корпоративных приложений на каждой настольной системе.

Во втором сценарии поставщики программного обеспечения получают сертификаты, подписанные организациями вроде VeriSign. При посещении пользователем веб-сайта с подписанным аплетом на экране появляется диалоговое окно, указывающее поставщика данного программного обеспечения и предлагающее пользователю на выбор выполнить одно из следующих действий: запустить аплет с полным набором привилегий или ограничить его область действия "песочницей". Более подробно этот менее желательный сценарий будет рассматриваться далее, в разделе "Сертификаты разработчиков программного обеспечения".

В остальной части этого раздела поясняется, как создавать файлы правил защиты, предоставляющие коду из известных источников только конкретные полномочия. Создание и развертывание таких файлов не может быть доверено обычным конечным пользователям. Но системным администраторам разрешается выполнять эти задачи при подготовке к распространению программ по внутренней корпоративной сети.

Допустим, компании ACME Software требуется, чтобы ее сотрудники могли запускать определенные программы, требующие доступа к локальным файлам, но сделать так, чтобы эти программы были доступны через браузер в виде аплетов или приложений Web Start. Работать в пределах "песочницы" такие программы не смогут, поэтому компании ACME Software придется установить файлы прав защиты на компьютерах ее сотрудников.

Как было показано ранее в этой главе, компания ACME Software могла бы идентифицировать программы по их кодовой базе. Но это означает, что ей пришлось было бы обновлять файлы прав защиты всякий раз, когда программы переносятся на другой сервер. Поэтому в компании ACME Software вместо этого принято решение *подписывать* JAR-файлы, содержащие программный код.

Для этого сначала формируется корневой сертификат по следующей команде:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
```

Разумеется, хранилище ключей, содержащее секретный корневой ключ, должно обязательно находиться в безопасном месте. Поэтому создается второе хранилище ключей `client.certs` для размещения в нем открытых сертификатов и в него сразу же вводится открытый сертификат `acmeroot`:

```
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer
keytool -importcert -keystore client.certs -alias acmeroot -file acmeroot.cer
```

В процессе создания подписанного JAR-файла программисты могут добавлять свои файлы классов в этот JAR-файл обычным образом, как показано в приведенном ниже примере.

```
javac FileReadApplet.java
jar cvf FileReadApplet.jar *.class
```

Далее уполномоченный сотрудник компании ACME Software может запустить утилиту `jarsigner`, указав JAR-файл и псевдоним секретного ключа следующим образом:

```
jarsigner -keystore acmesoft.certs FileReadApplet.jar acmeroot
```

После этого аплет можно считать готовым к развертыванию на веб-сервере. А теперь перейдем к настройке безопасности на стороне клиента. Файл правил защиты должен рассылаться каждой клиентской машине. Для ссылки на хранилище ключей файл правил защиты начинается со следующей строки:

```
keystore "URL_хранилища_ключей", "тип_хранилища_ключей";
```

URL может быть как абсолютным, так и относительным. Относительные URL являются таковыми по отношению к месту расположения файла правил защиты. Если хранилище формировалось с помощью утилиты `keytool`, оно будет относиться к типу `JKS`, как показано в приведенном ниже примере.

```
keystore "client.certs", "JKS";
```

Далее, операторы `grant` в файле правил защиты могут быть снабжены суффиксами `signedBy "псевдоним"`, как выделено ниже полужирным.

```
grant signedBy "acmeroot"
{
  . . .
};
```

В этих операторах предоставляются полномочия любому коду, который может быть проверен с помощью открытого ключа, связанного с указанным псевдонимом.

Вы можете опробовать процесс подписания кода на примере аплета (листинг 9.17), который пытается прочитать локальный файл. Правила защиты по умолчанию разрешают ему считывать файлы только из каталога кодовой базы и любых его подкаталогов. Воспользуйтесь утилитой `appletviewer` для просмотра аплетов, чтобы запустить этот аплет и удостовериться, что с его помощью можно просматривать файлы только из каталога и подкаталогов кодовой базы и никаких других каталогов.

Для целей рассматриваемого здесь примера предоставляется файл правил защиты `applets.policy`, содержащий следующее:

```
keystore "client.certs", "JKS";
grant signedBy "acmeroot"
{
  permission java.lang.RuntimePermission "usePolicy";
  permission java.io.FilePermission "/etc/*", "read";
};
```

В этих правилах защиты полномочия `usePolicy` переопределяют полномочия типа "все или ничего", используемые по умолчанию для подписанных аплетов, а также указывается, что всем аплетам, которые подписаны с помощью сертификата `acmeroot`, разрешается считывать файлы из каталога `/etc` (пользователям Windows этот каталог нужно заменить на какой-нибудь другой, например `C:\Windows`).

Теперь осталось лишь указать утилите `appletviewer` для просмотра аплетов, чтобы она использовала этот файл правил защиты. Ниже показано, как это делается. После этого аплет сможет считывать файлы из каталога `/etc`, демонстрируя тем самым принцип действия механизма, применяемого для подписания кода.

```
appletviewer -J-Djava.security.policy=applet.policy FileReadApplet.html
```

СОВЕТ. Если вы испытываете затруднения при выполнении упомянутого выше действия, введите дополнительный параметр **`-J-Djava.security.debug=policy`**, чтобы вывести подробные сообщения, позволяющие отслеживать порядок соблюдения программой установленных правил защиты.

В качестве заключительной проверки попробуйте запустить аплет в браузере (рис. 9.14). Для этого вам придется скопировать файл полномочий и хранилище ключей в каталог развертывания Java. В UNIX и Linux это каталог `.java/deployment`, находящийся в начальном каталоге, а в Windows Vista и Windows 7 — каталог `C:\Users\`*`регистрационное_имя_пользователя`*`\AppData\Sun\Java\Deployment`. Ради удобства он будет называться в дальнейшем *`deploydir`*, т.е. каталог развертывания программ.

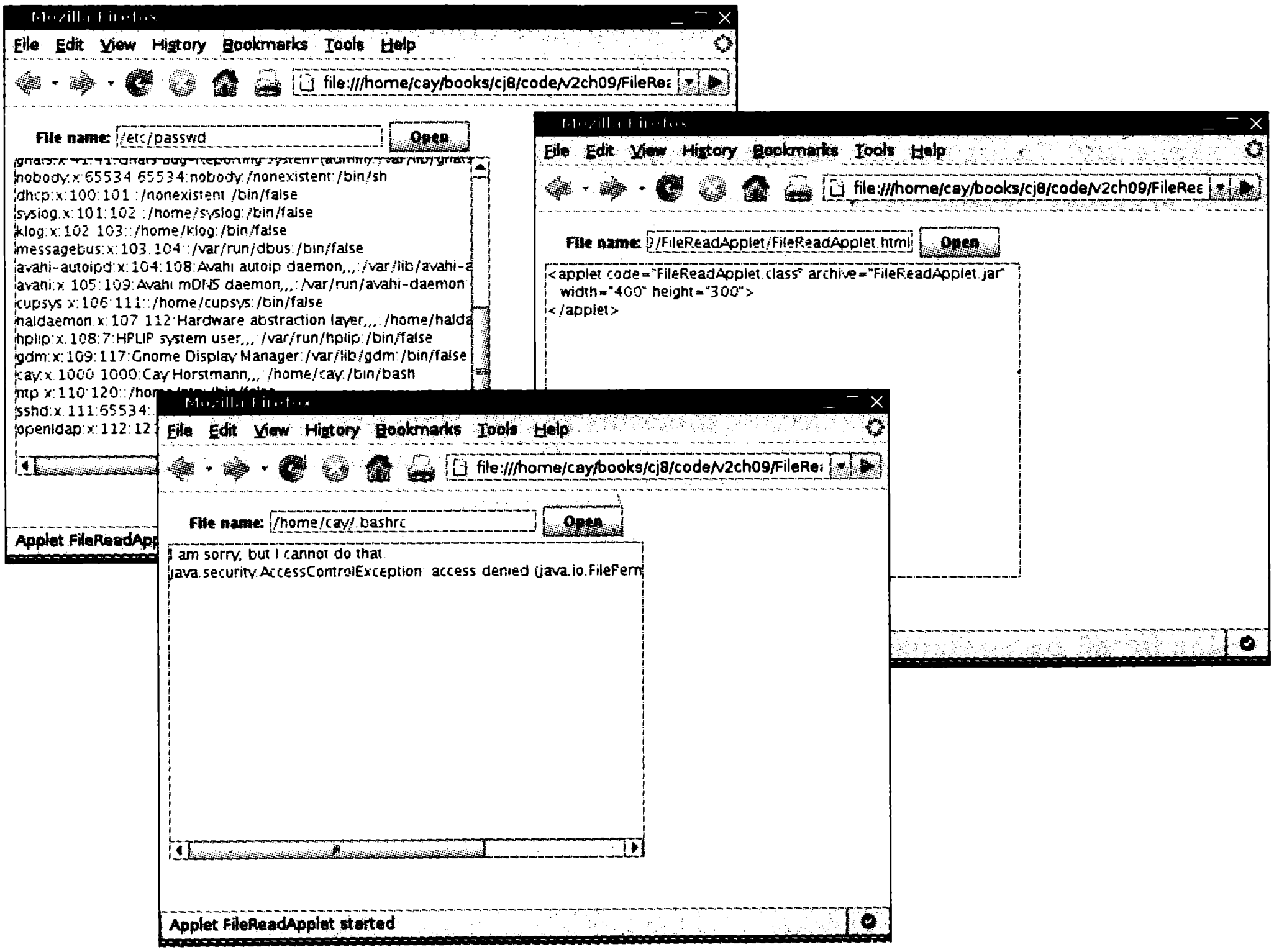


Рис. 9.14. Подписанный аплет имеет доступ к локальным файлам для чтения

Скопируйте файлы `applet.policy` и `client.certs` в каталог *`deploydir`*`/security`. Переименуйте в этом каталоге файл `applets.policy` на `java.policy`. (Сделайте это очень аккуратно, чтобы не удалить случайно существующий файл `java.policy`. Если таковой имеется, просто введите в него содержимое файла `applet.policy`.)

СОВЕТ. Подробнее о настройке клиентских параметров безопасности на платформе Java можно узнать из руководства по развертыванию Java в разделах "Deployment Configuration File and Properties" (Развертывание конфигурационного файла и свойств) и Java Control Panel (Панель управления Java) по адресу `http://docs.oracle.com/javase/7/docs/technotes/guides/deployment/deployment-guide/overview.html`.

Перезапустите браузер и загрузите файл `FileReadApplet.html`. При этом *не* должно быть никакого приглашения на принятие сертификата. Далее удостоверьтесь, что файлы можно загружать из каталога `/etc` и каталога с аплетом, а из всех остальных каталогов — нельзя.

По завершении не забудьте очистить каталог *deploydir*`/security`, удалив из него файлы `java.policy` и `client.certs`. Затем перезапустите браузер. Если после очистки каталога вы снова попробуете загрузить аплет, то у вас уже не должно быть полномочий на чтение файлов из локальной файловой системы. Вместо этого на экране появится приглашение на принятие сертификата. Подробнее о сертификатах безопасности речь пойдет в следующем разделе.

Листинг 9.17. Исходный код из файла `signed/FileReadApplet.java`

```
1  package signed;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.io.*;
6  import java.nio.file.*;
7  import javax.swing.*;
8
9  /**
10  * Этот аплет можно выполнять вне "песочницы", чтобы читать
11  * локальные файлы, как только он наделяется нужными полномочиями
12  * @version 1.12 2012-06-10
13  * @author Cay Horstmann
14 */
15 public class FileReadApplet extends JApplet
16 {
17   private JTextField fileNameField;
18   private JTextArea fileText;
19
20   public void init()
21   {
22     EventQueue.invokeLater(new Runnable()
23       {
24         public void run()
25         {
26           fileNameField = new JTextField(20);
27           JPanel panel = new JPanel();
28           panel.add(new JLabel("File name:"));
29           panel.add(fileNameField);
30           JButton openButton = new JButton("Open");
31           panel.add(openButton);
32           ActionListener listener = new ActionListener()
33           {
34             public void actionPerformed(ActionEvent event)
```

```
35              {
36                loadFile(fileNameField.getText());
37              }
38            };
39            fileNameField.addActionListener(listener);
40            openButton.addActionListener(listener);
41
42            add(panel, "North");
43
44            fileText = new JTextArea();
45            add(new JScrollPane(fileText), "Center");
46          }
47        });
48    }
49
50    /**
51     * Загружает содержимое файла в текстовую область
52     * @param filename the file name
53    */
54    public void loadFile(String filename)
55    {
56      fileText.setText("");
57      try
58      {
59        fileText.append(new String(
60           Files.readAllBytes(Paths.get(filename))));
61      }
62      catch (IOException ex)
63      {
64        fileText.append(ex + "\n");
65      }
66      catch (SecurityException ex)
67      {
68        fileText.append("I am sorry, but I cannot do that.\n");
69        fileText.append(ex + "\n");
70        ex.printStackTrace();
71      }
72    }
73 }
```

Сертификаты разработчиков программного обеспечения

До сих пор обсуждались сценарии рассылки программ по внутренней корпоративной сети. В этом случае системный администратор настраивает правила защиты, управляющие привилегиями отдельных программ. Такая методика обеспечения безопасности подходит только для программ из известных источников.

Допустим, что, выйдя в Интернет, пользователь открывает веб-страницу, на которой предлагается запустить аплет или приложение Web Start неизвестного поставщика, как, например, показано на рис. 9.15. Такой аплет подписан сертификатом разработчика программного обеспечения. В диалоговом окне указывается разработчик данной программы и создатель сертификата. Таким образом, у пользователя есть два варианта для продолжения работы.

- Запустить аплет с полными правами доступа.
- Ограничить работу аплета пределами "песочницы". (Название кнопки Cancel в диалоговом окне вводит в заблуждение. На самом деле щелчок на этой кнопке приводит не к отмене запуска аплета, а к его запуску в пределах "песочницы".)

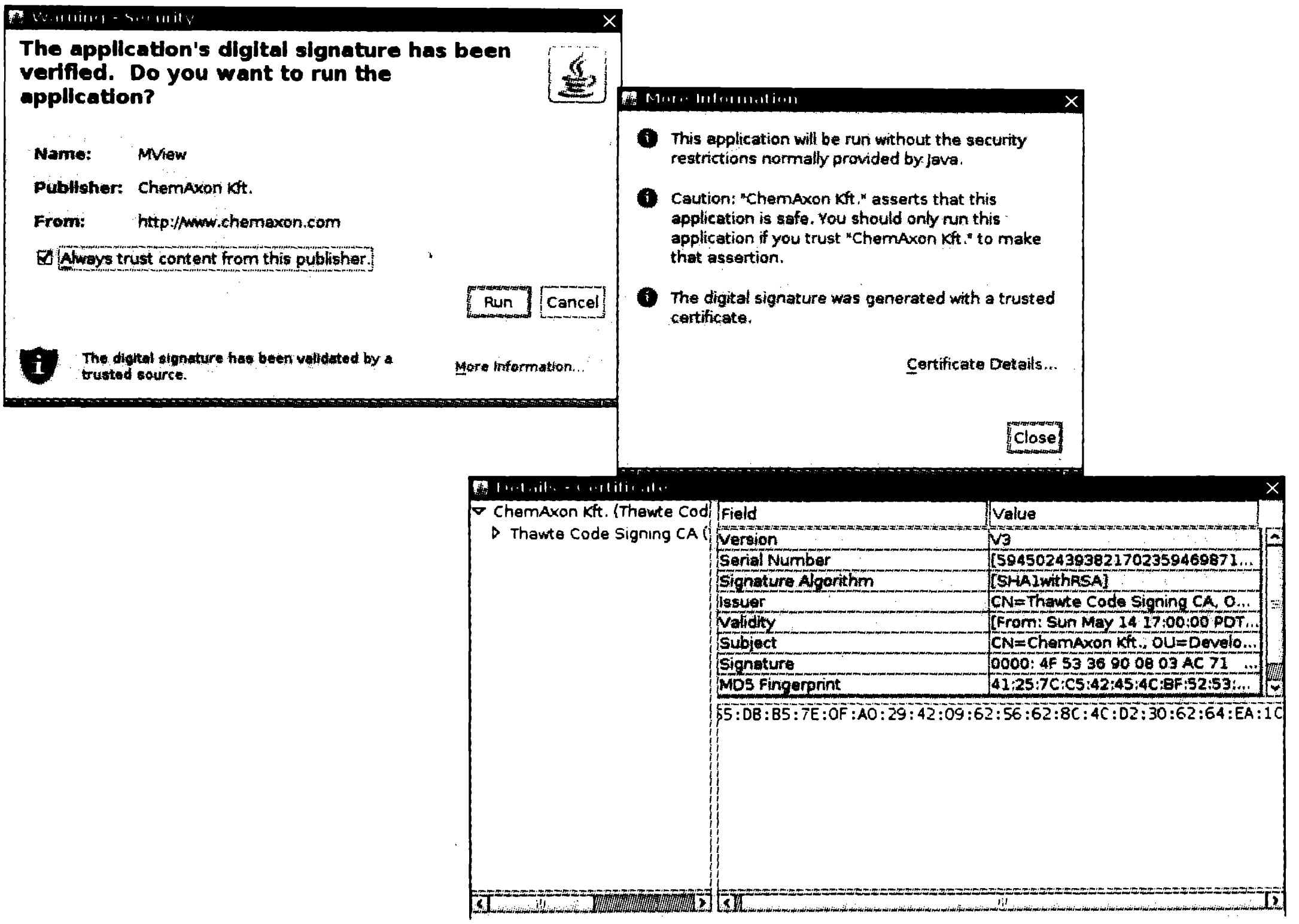


Рис. 9.15. Запуск подписанного аплета

Попробуем выяснить, что известно пользователю в подобной ситуации и что может повлиять на его решение?

- Компания Thawte продала сертификат разработчику программного обеспечения.
- Аплет действительно подписан этим сертификатом и во время пересылки не был изменен.
- Сертификат действительно подписан компанией Thawte и проверен с помощью открытого ключа, который находится в локальном файле `cacerts`.

Значит ли это, что данный код надежен? Следует ли доверять поставщику, если известно только его имя и тот факт, что компания Thawte продала ему сертификат разработчика программного обеспечения? Вряд ли компания Thawte сможет гарантировать, что под именем разработчика программного обеспечения ChemAxon Kft. не скрывается явный злоумышленник. Более того, создатели сертификатов не претендуют на это, т.е. они не занимаются проверкой честности и компетентности поставщиков программного обеспечения. Они проверяют подлинность поставщиков

программного обеспечения по сканированной копии их лицензии на ведение коммерческой деятельности или паспорта.

В ситуации, когда название поставщика данного программного обеспечения неизвестно пользователю, нужно тщательно взвесить все аргументы за и против, прежде чем предоставлять данной программе права доступа. А если название поставщика данного программного обеспечения хорошо известно пользователю, то он может принять решение, учитывая репутацию этого поставщика.

Вряд ли стоит доверять программе, которая заявляет "дай мне все права, или я вообще не буду работать". Неопытные пользователи часто совершают ошибку и необдуманно предоставляют все права такой программе, что зачастую приводит к возникновению очень опасных ситуаций. Стоит ли использовать аплеты, которые подробно объясняют свои намерения и перечисляют все необходимые им права? К сожалению, в таком случае конечному пользователю придется погрузиться в чисто технические подробности работы данной программы. Вряд ли обычный пользователь сможет разобраться, в каких ситуациях, скажем, следует предоставить аплету права доступа к очереди событий AWT.

Мы довольно скептически относимся к сертификатам, подписываемым разработчиками программного обеспечения. На наш взгляд, было бы гораздо лучше, если бы аплеты и приложения Web Start в такой общедоступной сети, как Интернет, старались функционировать в пределах своих "песочниц" и если бы эти "песочницы" совершенствовались. Первым шагом в верном направлении является прикладной интерфейс API для приложений Web Start, который обсуждался в главе 10 первого тома данной книги.

Шифрование

До сих пор рассматривалась аутентификация с помощью цифровых подписей. Еще одним важным средством обеспечения безопасности является *шифрование*. Информация, заверенная цифровой подписью, доступна для просмотра, а подпись лишь подтверждает, что эта информация не была изменена. Для просмотра зашифрованных данных этого недостаточно, поскольку их нужно расшифровать с помощью согласованного ключа.

Аутентификации оказывается достаточно для подписания кода, который не нужно скрывать. А шифрование требуется в тех случаях, когда аплеты или приложения передают конфиденциальную информацию, например, номера кредитных карточек и прочие личные данные.

В прошлом во многих компаниях существовали патентные и экспортные ограничения на использование эффективных алгоритмов шифрования. Теперь же все эти ограничения стали, к счастью, менее жесткими, а срок патентных ограничений на использование алгоритма шифрования RSA и вовсе истек в октябре 2000 г. В стандартной библиотеке текущей версии Java SE предусмотрены достаточно хорошие средства шифрования.

Симметричные шифры

Криптографические расширения Java содержат класс `Cipher`, который является суперклассом для всех классов, имеющих отношение к шифрованию. Для создания объекта, реализующего алгоритм шифрования, используется метод `getInstance()`

```
Cipher cipher = Cipher.getInstance(algorithName);
```

или

```
Cipher cipher = Cipher.getInstance(algorithName, providerName);
```

В JDK для всех шифров используется поставщик SunJCE. Если имя поставщика не указано явно, то по умолчанию принимается имя SunJCE. Если же требуется воспользоваться алгоритмами шифрования, которые не поддерживаются инструментальными средствами компании Oracle, следует указать другого поставщика. Название алгоритма шифрования задается в виде символьной строки, например "DES" или "DES/CBC/PKCS5Padding".

Алгоритм DES (Data Encryption Standard — стандарт шифрования данных) — один из наиболее старых алгоритмов шифрования с длиной ключа 56 бит. В настоящее время он считается устаревшим, поскольку может взламываться методом "грубой силы" (соответствующий пример можно найти по адресу http://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker). Намного более эффективным оказывается появившийся после него алгоритм AES (Advanced Encryption Standard — Усовершенствованный стандарт шифрования), подробное описание которого можно найти по адресу www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf. В рассматриваемых далее примерах будет использоваться алгоритм шифрования AES.

Как только объект, реализующий алгоритм шифрования, будет создан, его нужно инициализировать, указав режим и ключ шифрования, т.е. установив параметры mode и key следующим образом:

```
int mode = . . .;
Key key = . . .;
cipher.init(mode, key);
```

Параметр mode может принимать значение одной из следующих констант:

```
Cipher.ENCRYPT_MODE
Cipher.DECRYPT_MODE
Cipher.WRAP_MODE
Cipher.UNWRAP_MODE
```

Режимы *свертывания* и *развертывания* применяются для шифрования одного ключа на основе другого. Пример применения таких режимов будет приведен в следующем разделе. После этого можно повторно вызывать метод update() для шифрования всех требующихся блоков данных, как показано ниже.

```
int blockSize = cipher.getBlockSize();
byte[] inBytes = new byte[blockSize];
. . . // прочитать байты из входного массива inBytes
int outputSize= cipher.getOutputSize(blockSize);
byte[] outBytes = new byte[outputSize];
int outLength = cipher.update(inBytes, 0, outputSize, outBytes);
. . . // записать байты в выходной массив outBytes
```

По завершении следует непременно вызвать метод doFinal() один раз. Если доступен последний блок вводимых данных (меньшего объема, чем указано в переменной blockSize), то данный метод нужно вызвать следующим образом:

```
outBytes = cipher.doFinal(inBytes, 0, inLength);
```

А если были зашифрованы все вводимые данные, то данный метод нужно вызвать так:

```
outBytes = cipher.doFinal();
```

Вызывать метод `doFinal()` требуется для того, чтобы заполнить завершающий блок данных. Рассмотрим, например, алгоритм шифрования DES. В этом алгоритме размер блока составляет 8 байт. Допустим, размер последнего блока вводимых данных оказывается меньше 8 байт. Разумеется, недостающие байты можно дополнить нулями таким образом, чтобы блок занимал 8 байт, а затем зашифровать его. Но после расшифровки блоков данных полученный результат будет содержать несколько присоединенных в конце нулей, а следовательно, будет несколько отличаться от исходного файла данных. Это может представлять определенное затруднение, и для ее преодоления как раз и требуется *схема заполнения*. Одной из наиболее часто применяемых является схема заполнения, описанная в документе Public Key Cryptography Standard (PKCS — стандарт шифрования открытым ключом) #5 специалистами из компании RSA Security, Inc. (`ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf`). В этой схеме последний блок заполняется не нулями, а числами, равными недостающему количеству байтов. Иными словами, если `L` — это последний (неполный) блок данных, то он будет заполнен следующим образом:

```
L 01                          если length(L) = 7
L 02 02                       если length(L) = 6
L 03 03 03                    если length(L) = 5
...
L 07 07 07 07 07 07 07        если length(L) = 1
```

И наконец, если длина вводимых данных делится нацело на 8, то к вводимым данным присоединяется и затем шифруется только один такой блок, как показано ниже. А при расшифровке на количество отбрасываемых заполняющих символов указывает самый последний байт простого текста.

```
08 08 08 08 08 08 08 08
```

Генерирование ключей шифрования

Для шифрования требуется сгенерировать ключ. Каждый алгоритм шифрования предусматривает свой формат для ключей, но самое главное, чтобы их генерирование выполнялось произвольным образом. Чтобы получить сгенерированный совершенно произвольно ключ, требуется выполнить три действия.

1. Создать объект типа `KeyGenerator`.
2. Инициализировать генератор случайных чисел. Если блок шифра имеет переменную длину, необходимо также указать желаемую длину блока.
3. Вызвать метод `generateKey()`.

В качестве примера ниже показано, каким образом генерируется ключ шифрования по алгоритму AES.

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecureRandom random = new SecureRandom(); // см. пояснение ниже
keygen.init(random);
Key key = keygen.generateKey();
```

С другой стороны, ключ шифрования можно также сгенерировать из фиксированного набора необработанных данных (например, пароля или вводимых с клавиатуры символов). В таком случае следует воспользоваться классом `SecretKeyFactory`, как показано ниже.

```
byte[] keyData = . . .; // 16 байт для ключа шифрования по алгоритму AES
SecretKey key = new SecretKeySpec(keyData, "AES");
```

При генерировании ключей следует непременно удостовериться, что для этой цели используются *подлинно случайные* числа. Например, обычный генератор случайных чисел, реализуемый в классе `Random`, где в роли начальных значений выступают текущие дата и время, не будет генерировать случайные в достаточной степени числа. Допустим, системные часы показывают время с точностью до 1/10 секунды. Это означает, что на каждый день может приходиться максимум по 864000 начальных значений. Следовательно, узнав день генерирования ключа шифрования (его зачастую нетрудно вычислить по дате сообщения или дате истечения срока действия сертификата), злоумышленнику останется лишь сгенерировать все возможные значения за этот день.

Класс `SecureRandom` генерирует намного более надежные случайные числа, чем класс `Random`. Но предоставлять ему начальное значение, с которого должна начинаться числовая последовательность в произвольной точке, все равно нужно. Для этого удобнее всего получить случайные данные, вводимые из какого-нибудь аппаратного устройства вроде генератора белого шума. С другой стороны, случайные данные можно получить, пригласив пользователя к вводу любой бессмысленной последовательности символов, из которой в качестве начального случайного значения должен выбираться только один или два бита. После накопления таких случайных битов в массив байтов они должны быть просто переданы методу `setSeed()`, как показано ниже.

```
SecureRandom secrand = new SecureRandom();
byte[] b = new byte[20];
// заполнить массив подлинно случайными битами
secrand.setSeed(b);
```

Если не указать генератору случайных чисел никакого начального значения, он сформирует собственное 20-байтовое значение, запустив несколько потоков на исполнение, переведя их в режим ожидания и рассчитав точное время их выхода из режима ожидания.

НА ЗАМЕТКУ! Трудно сказать, является ли такой алгоритм безопасным. Раньше алгоритмы, действие которых основывалось на вычислении временных показателей некоторых компонентов компьютера, например времени доступа к жесткому диску, зачастую оказывались на поверку генерирующими недостаточно случайные числа.

В примере программы из листинга 9.18 демонстрируется применение алгоритма шифрования AES. А утилита шифрования `crypt` из листинга 9.19 будет неоднократно использоваться в остальных примерах далее в этой главе. Для того чтобы воспользоваться данной программой, нужно сначала сгенерировать секретный ключ, выполнив команду

```
java aes.AESTest -genkey secret.key
```

Полученный секретный ключ следует сохранить в файле `secret.key`. После этого шифрование можно выполнить по следующей команде:

```
java aes.AESTest -encrypt plaintextFile encryptedFile secret.key
```

А расшифровывание производится по такой команде:

```
java aes.AESTest -decrypt encryptedFile decryptedFile secret.key
```

Рассматриваемая здесь программа довольно проста. Сначала с помощью параметра `-genkey` генерируется новый секретный ключ, который затем сохраняется в указанном файле. Эта операция обычно занимает много времени, поскольку бо́льшая ее часть уходит на инициализацию генератора случайных чисел. Параметры **`-encrypt`** и **`-decrypt`** предусматривают вызов одного и того же метода `crypt()`, который, в свою очередь, вызывает такие методы для объекта шифрования, как `update()` и `doFinal()`. Следует иметь в виду, что метод `update()` вызывается повторно до тех пор, пока блоки вводимых данных не достигнут полной длины, и что метод `doFinal()` вызывается с частично заполненным блоком вводимых данных (который затем заполняется) *или* вообще безо всяких дополнительных данных (для формирования одного заполняющего блока).

Листинг 9.18. Исходный код из файла `aes/AESTest.java`

```
1  package aes;
2
3  import java.io.*;
4  import java.security.*;
5  import javax.crypto.*;
6  /**
7   * В этой программе проверяется шифрование по алгоритму AES. Применение:
8   * java aes.AESTest -genkey keyfile
9   * java aes.AESTest -encrypt plaintext encrypted keyfile
10  * java aes.AESTest -decrypt encrypted decrypted keyfile
11  * @author Cay Horstmann
12  * @version 1.01 2012-06-10
13 */
14 public class AESTest
15 {
16   public static void main(String[] args)
17      throws IOException, GeneralSecurityException, ClassNotFoundException
18   {
19     if (args[0].equals("-genkey"))
20     {
21       KeyGenerator keygen = KeyGenerator.getInstance("AES");
22       SecureRandom random = new SecureRandom();
23       keygen.init(random);
24       SecretKey key = keygen.generateKey();
25       try (ObjectOutputStream out = new ObjectOutputStream(
26          new FileOutputStream(args[1])))
27       {
28         out.writeObject(key);
29       }
30     }
31     else
32     {
33       int mode;
34       if (args[0].equals("-encrypt")) mode = Cipher.ENCRYPT_MODE;
35       else mode = Cipher.DECRYPT_MODE;
36
37       try (ObjectInputStream keyIn = new ObjectInputStream(
38          new FileInputStream(args[3]));
39          InputStream in = new FileInputStream(args[1]);
40          OutputStream out = new FileOutputStream(args[2]))
```

```
41         {
42           Key key = (Key) keyIn.readObject();
43           Cipher cipher = Cipher.getInstance("AES");
44           cipher.init(mode, key);
45           Util.crypt(in, out, cipher);
46         }
47       }
48     }
49 }
```

Листинг 9.19. Исходный код из файла `aes/Util.java`

```
1  package aes;
2
3  import java.io.*;
4  import java.security.*;
5  import javax.crypto.*;
6
7  public class Util
8  {
9    /**
10    * Использует шифр для преобразования байтов из потока ввода и
11    * направляет преобразованные байты в поток вывода
12    * @param in Поток ввода
13    * @param out Поток вывода
14    * @param cipher Шифр для преобразования байтов
15   */
16   public static void crypt(InputStream in, OutputStream out,
17      Cipher cipher) throws IOException, GeneralSecurityException
18   {
19     int blockSize = cipher.getBlockSize();
20     int outputSize = cipher.getOutputSize(blockSize);
21     byte[] inBytes = new byte[blockSize];
22     byte[] outBytes = new byte[outputSize];
23     int inLength = 0;
24
25     boolean more = true;
26     while (more)
27     {
28       inLength = in.read(inBytes);
29       if (inLength == blockSize)
30       {
31         int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
32         out.write(outBytes, 0, outLength);
33       }
34       else more = false;
35     }
36     if (inLength > 0) outBytes = cipher.doFinal(inBytes, 0, inLength);
37     else outBytes = cipher.doFinal();
38     out.write(outBytes);
39   }
40 }
```

javax.crypto.Cipher 1.4

- `static Cipher getInstance(String algorithmName)`
- `static Cipher getInstance(String algorithmName, String providerName)`

  Возвращают объект типа `Cipher`, реализующий указанный алгоритм шифрования. Если же такой алгоритм не поддерживается, то генерируется исключение типа `NoSuchAlgorithmException`.
- `int getBlockSize()`

  Возвращает размер шифруемого блока в байтах или нулевое значение, если алгоритм шифрования не предусматривает обращение с блоками данных.
- `int getOutputSize(int inputLength)`

  Возвращает размер буфера выводимых данных, который требуется, если следующие выводимые данные имеют количество байтов, определяемое параметром *inputLength*. В этом методе учитываются любые байты, буферизированные в объекте шифрования.
- `void init(int mode, Key key)`

  Инициализирует объект, реализующий алгоритм шифрования. Параметр режима может принимать значение одной из следующих констант: `ENCRYPT_MODE`, `DECRYPT_MODE`, `WRAP_MODE` или `UNWRAP_MODE`.
- `byte[] update(byte[] in)`
- `byte[] update(byte[] in, int offset, int length)`
- `int update(byte[] in, int offset, int length, byte[] out)`

  Преобразуют один блок вводимых данных. Первые два метода возвращают выводимые данные, а третий метод — сведения о количестве байтов, которые были размещены в массиве *out*.
- `byte[] doFinal()`
- `byte[] doFinal(byte[] in)`
- `byte[] doFinal(byte[] in, int offset, int length)`
- `int doFinal(byte[] in, int offset, int length, byte[] out)`

  Преобразуют последний блок вводимых данных и очищают буфер этого объекта шифрования. Первые три метода возвращают вводимые данные, а четвертый метод — сведения о количестве байтов, которые были размещены в массиве `out`.
- `javax.crypto.KeyGenerator 1.4`
- `static KeyGenerator getInstance(String algorithmName)`

  Возвращает объект типа `KeyGenerator`, реализующий указанный алгоритм шифрования. Если такой алгоритм не поддерживается, то генерируется исключение типа `NoSuchAlgorithmException`.
- `void init(SecureRandom random)`
- `void init(int keySize, SecureRandom random)`

  Инициализируют генератор ключей шифрования.
- `SecretKey generateKey()`
- Генерирует новый ключ шифрования.

javax.crypto.spec.SecretKeySpec 1.4

- `SecretKeySpec(byte[] key, String algorithmName)`

  Формирует новый секретный ключ по указанной спецификации.

Потоки шифрования

В библиотеке JCE имеется удобный набор классов, реализующих потоки ввода-вывода и способных автоматически шифровать и расшифровывать данные из этих потоков. В качестве примера ниже показано, как с помощью одного из таких классов зашифровать данные и вывести их в файл.

```
Cipher cipher = . . .;
cipher.init(Cipher.ENCRYPT_MODE, key);
CipherOutputStream out = new CipherOutputStream(
   new FileOutputStream(outputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = getData(bytes); // получить данные из источника
while (inLength != -1)
{
  out.write(bytes, 0, inLength);
  inLength = getData(bytes); // получить дополнительные данные из источника
}
out.flush();
```

Подобным образом для чтения данных из файла и их расшифровки можно применить класс `CipherInputStream`:

```
Cipher cipher = . . .;
cipher.init(Cipher.DECRYPT_MODE, key);
CipherInputStream in = new CipherInputStream(
   new FileInputStream(inputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = in.read(bytes);
while (inLength != -1)
{
  putData(bytes, inLength); // вывести данные по месту назначения
  inLength = in.read(bytes);
}
```

Классы потоков шифрования прозрачно обрабатывают вызовы методов `update()` и `doFinal()`, что, безусловно, очень удобно.

javax.crypto.CipherInputStream 1.4

- `CipherInputStream(InputStream in, Cipher cipher)`

  Создает поток ввода, который считывает данные из потока ввода `in` и расшифровывает или зашифровывает их, используя указанный шифр.

- `int read()`
- `int read(byte[] b, int off, int len)`

  Считывают данные из потока ввода и автоматически расшифровывают или зашифровывают их.

javax.crypto.CipherOutputStream 1.4

- `CipherOutputStream(OutputStream` *`out`*`, Cipher` *`cipher`*`)`
- Создает поток вывода, который записывает данные в поток вывода `out` и зашифровывает или расшифровывает их, используя указанный шифр.
- `void write(int` *`ch`*`)`
- `void write(byte[]` *`b`*`, int` *`off`*`, int` *`len`*`)`
- Записывают данные в поток вывода и автоматически зашифровывают или расшифровывают их.
- `void flush()`
- Выводит данные из буфера шифрования, очищая его, заполняя при необходимости недостающие биты.

Шифрование открытым ключом

Алгоритмы шифрования DES и AES, рассматривавшиеся в предыдущем разделе, являются *симметричными*. Это означает, что один и тот же ключ применяется как для шифрования, так и для расшифровывания. Уязвимым местом всех симметричных алгоритмов шифрования является передача ключа. Так, если Алиса отправляет Бобу зашифрованное сообщение, для его расшифровки Бобу необходим тот же самый ключ, которым пользовалась Алиса. При изменении ключа Алиса снова должна отправить Бобу вместе с сообщением новую версию этого ключа по какому-нибудь защищенному каналу. Но если у нее нет доступа к такому каналу связи с Бобом, то ей придется сначала зашифровать все отправляемые ему сообщения.

В качестве выхода из этого положения можно воспользоваться шифрованием открытым ключом. В таком случае у Боба будет пара ключей: открытый и соответствующий ему секретный ключ. Он сможет передавать свой открытый ключ кому угодно, а секретный ключ хранить в тайне. Алисе же останется только использовать этот открытый ключ для шифрования всех своих сообщений Бобу.

К сожалению, все не так просто. Дело в том, что все алгоритмы шифрования открытым ключом действуют *намного* медленнее, чем алгоритмы шифрования симметричными ключами вроде DES или AES. Поэтому было бы неэффективно и непрактично использовать открытые ключи для шифрования данных большого объема. Но это затруднение нетрудно разрешить, сочетая шифрование открытым ключом с симметричным шифрованием. Рассмотрим пример.

1. Алиса формирует произвольный симметричный ключ и шифрует им свое сообщение.
2. Используя открытый ключ Боба, Алиса зашифровывает этот симметричный ключ.
3. Алиса посылает Бобу зашифрованное сообщение вместе с зашифрованным симметричным ключом.
4. Используя секретный ключ, Боб расшифровывает симметричный ключ.
5. Используя расшифрованный симметричный ключ, Боб расшифровывает полученное сообщение.

В данном примере никто, кроме Боба, не сможет расшифровать симметричный ключ, потому что секретный ключ имеется только у него. Таким образом, неэффективный алгоритм шифрования открытым ключом применяется только для небольшого объема данных симметричного ключа.

Наиболее распространенным алгоритмом шифрования открытым ключом является RSA, изобретенный Райвестом, Шамиром и Адлеманом. До октября 2000 года этот алгоритм был защищен патентом, выданным компании RSA Security Inc., а для получения лицензии на его использование нужно было платить 3%-ную пошлину с минимальным ежегодным взносом 50 тыс. долларов США. В настоящее время этот алгоритм уже не является коммерческим и стал всеобщим достоянием.

Для того чтобы воспользоваться алгоритмом шифрования RSA, нужно создать открытый и секретный ключи с помощью класса `KeyPairGenerator`:

```
KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
SecureRandom random = new SecureRandom();
pairgen.initialize(KEYSIZE, random);
KeyPair keyPair = pairgen.generateKeyPair();
Key publicKey = keyPair.getPublic();
Key privateKey = keyPair.getPrivate();
```

В примере программы из листинга 9.20 применяются три параметра. В частности, параметр **`-genkey`** служит для создания пары ключей, а параметр **`-encrypt`** — для генерирования ключа по алгоритму шифрования AES и его свертывания с помощью открытого ключа, как показано ниже.

```
Key key = . . .; // ключ по алгоритму шифрования AES
Key publicKey = . . .; // открытый ключ по алгоритму шифрования RSA
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.WRAP_MODE, publicKey);
byte[] wrappedKey = cipher.wrap(key);
```

Далее создается файл, состоящий из следующих элементов.

- Длина свернутого ключа.
- Байты свернутого ключа.
- Открытый текст, зашифрованный ключом по алгоритму AES.

Третий параметр, **`-decrypt`**, служит для расшифровывания файла. Чтобы опробовать данную программу, нужно сначала создать ключи по алгоритму шифрования RSA, выполнив следующую команду:

```
java rsa.RSATest -genkey public.key private.key
```

Затем необходимо зашифровать файл по команде

```
java rsa.RSATest -encrypt plaintextFile encryptedFile public.key
```

И наконец, файл нужно расшифровать и проверить, совпадают ли полученные данные с ранее зашифрованными, как показано ниже.

Листинг 9.20. Исходный код из файла `rsa/RSATest.java`

```
1  package rsa;
2
3  import java.io.*;
4  import java.security.*;
5  import javax.crypto.*;
```

```
6  /**
7   * В этой программе проверяется шифрование по алгоритму RSA. Применение:
8   * java rsa.RSATest -genkey public private
9   * java rsa.RSATest -encrypt plaintext encrypted public
10  * java rsa.RSATest -decrypt encrypted decrypted private
11  * @author Cay Horstmann
12  * @version 1.01 2012-06-10
13 */
14 public class RSATest
15 {
16   private static final int KEYSIZE = 512;
17
18   public static void main(String[] args)
19      throws IOException, GeneralSecurityException, ClassNotFoundException
20   {
21     if (args[0].equals("-genkey"))
22     {
23       KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
24       SecureRandom random = new SecureRandom();
25       pairgen.initialize(KEYSIZE, random);
26       KeyPair keyPair = pairgen.generateKeyPair();
27       try (ObjectOutputStream out = new ObjectOutputStream(
28          new FileOutputStream(args[1])))
29       {
30         out.writeObject(keyPair.getPublic());
31       }
32       try (ObjectOutputStream out = new ObjectOutputStream(
33          new FileOutputStream(args[2])))
34       {
35         out.writeObject(keyPair.getPrivate());
36       }
37     }
38     else if (args[0].equals("-encrypt"))
39     {
40       KeyGenerator keygen = KeyGenerator.getInstance("AES");
41       SecureRandom random = new SecureRandom();
42       keygen.init(random);
43       SecretKey key = keygen.generateKey();
44
45       // свернуть с помощью открытого ключа по алгоритму RSA
46       try (ObjectInputStream keyIn = new ObjectInputStream(
47            new FileInputStream(args[3]));
48         DataOutputStream out = new DataOutputStream(
49            new FileOutputStream(args[2]));
50         InputStream in = new FileInputStream(args[1]) )
51       {
52         Key publicKey = (Key) keyIn.readObject();
53         Cipher cipher = Cipher.getInstance("RSA");
54         cipher.init(Cipher.WRAP_MODE, publicKey);
55         byte[] wrappedKey = cipher.wrap(key);
56         out.writeInt(wrappedKey.length);
57         out.write(wrappedKey);
58
59         cipher = Cipher.getInstance("AES");
60         cipher.init(Cipher.ENCRYPT_MODE, key);
61         Util.crypt(in, out, cipher);
62       }
```

```
63      }
64      else
65      {
66        try (DataInputStream in = new DataInputStream(
67              new FileInputStream(args[1]));
68          ObjectInputStream keyIn =
69             new ObjectInputStream(new FileInputStream(args[3]));
70          OutputStream out = new FileOutputStream(args[2]))
71        {
72          int length = in.readInt();
73          byte[] wrappedKey = new byte[length];
74          in.read(wrappedKey, 0, length);
75
75          // развернуть с помощью секретного ключа по алгоритму RSA
76          Key privateKey = (Key) keyIn.readObject();
77
78          Cipher cipher = Cipher.getInstance("RSA");
79          cipher.init(Cipher.UNWRAP_MODE, privateKey);
80          Key key = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);
81
82          cipher = Cipher.getInstance("AES");
83          cipher.init(Cipher.DECRYPT_MODE, key);
84
85          Util.crypt(in, out, cipher);
86        }
87      }
88    }
89  }
```

В этой главе было показано, каким образом модель безопасности обеспечивает контролируемое выполнение кода, что является отличительной и очень важной особенностью платформы Java. Кроме того, в этой главе были рассмотрены различные службы, предоставляемые в библиотеке Java для аутентификации и шифрования. Тем не менее имеется еще немало дополнительных и специальных вопросов безопасности, которые не были затронуты в этой главе.

- Прикладной интерфейс GSS API для "универсальных служб безопасности", обеспечивающий поддержку протокола Kerberos (и, в принципе, других протоколов, предназначенных для безопасного обмена сообщениями). Подробнее об этом прикладном интерфейсе можно узнать по адресу `http://docs.oracle.com/javase/7/docs/technotes/guides/security/jgss/tutorials`.
- Поддержка механизма SASL (Simple Authentication and Security Layer — простой уровень аутентификации и безопасности), применяемого в протоколах LDAP и IMAP. Если вам потребуется реализовать механизм SASL в своей прикладной программе, необходимые сведения по данному вопросу вы найдете по адресу `http://docs.oracle.com/javase/7/docs/technotes/guides/security/sasl/sasl-refguide.html`.
- Поддержка протокола SSL (Secure Sockets Layer — уровень защищенных сокетов). Применение протокола SSL через протокол HTTP вполне прозрачно для разработчиков прикладных программ и состоит в том, чтобы указывать префикс **`https`** в URL. Если вам потребуется ввести протокол SSL в свою прикладную программу, необходимые сведения по данному вопросу вы сможете

найти в документе JSEE (Java Secure Socket Extension — расширение Java для защищенных сокетов) по адресу `http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html`.

На этом рассмотрение вопросов безопасности на платформе Java завершается. В следующей главе будут обсуждаться вопросы создания сценариев, компиляции кода и обработки аннотаций.

ГЛАВА 10

Создание сценариев, компиляция и обработка аннотаций

В этой главе...

- Написание сценариев для платформы Java
- Прикладной интерфейс API для компилятора
- Применение аннотаций
- Синтаксис аннотаций
- Стандартные аннотации
- Обработка аннотаций на уровне исходного кода
- Конструирование байт-кодов

В этой главе рассматриваются три методики обработки кода. Прикладной интерфейс API для создания сценариев позволяет вызывать код на языке сценариев наподобие JavaScript или Groovy, прикладной интерфейс API для компилятора дает возможность компилировать исходный код Java в самой прикладной программе, а обработчики аннотаций — обрабатывать файлы классов или исходного кода Java, содержащие аннотации. Из этой главы вы узнаете, что существует немало приложений для обработки аннотаций, начиная с простой диагностики и заканчивая так называемым "конструированием байт-кодов", вставкой байт-кодов в файлы классов и даже выполнением программ.

Написание сценариев для платформы Java

Язык сценариев — это такой язык программирования, который позволяет избегать обычного цикла операций редактирования, компиляции, компоновки и выполнения благодаря интерпретации исходного текста программы во время выполнения. Языки сценариев обладают рядом следующих преимуществ.

- Быстрый цикл обработки, стимулирующий стремление к экспериментированию.
- Возможность изменять поведение выполняющейся программы.
- Возможность для пользователей специально настраивать программы.

С другой стороны, у большинства языков сценариев отсутствуют средства, необходимые для разработки сложных прикладных программ, включая строгий контроль типов, инкапсуляцию и модульность.

В связи с этим возникает соблазн объединить преимущества языков сценариев с преимуществами традиционных языков программирования. Именно это и позволяет сделать прикладной интерфейс API для создания сценариев на платформе Java. В частности, он предоставляет возможность вызывать из программы на Java сценарии, написанные на JavaScript, Groovy, Ruby и даже таких экзотических языках, как Scheme и Haskell. (Возможность работать в обратном направлении, т.е. получать доступ к программе на Java из сценария, зависит уже от поставщика конкретного языка сценариев. У большинства языков сценариев, способных выполняться на виртуальной машине Java, такая возможность имеется.)

В последующих разделах будет показано, как выбирается механизм сценариев для конкретного языка, как выполняются сценарии и как пользоваться дополнительными преимуществами, которые дают некоторые механизмы сценариев.

Получение механизма сценариев

Механизм сценариев — это, по существу, библиотека, позволяющая выполнять сценарии на конкретном языке. При запуске виртуальная машина обнаруживает все доступные механизмы сценариев. Получить их перечень можно, создав объект типа `ScriptEngineManager` и вызвав метод `getEngineFactories()`. Далее у каждой фабрики механизмов можно запросить сведения об именах поддерживаемых механизмов, типах MIME и расширениях файлов. В табл. 10.1 перечислены наиболее употребительные механизмы сценариев и соответствующие им типы и расширения.

Таблица 10.1. Свойства фабрик механизмов сценариев

Механизм	Имена	Типы MIME	Расширения
Rhino (входит в состав Java SE)	`js`, `rhino`, `JavaScript`, `javascript`, `ECMAScript`, `ecmascript`	`application/javascript`, `application/ecmascript`, `text/javascript`, `text/ecmascript`	`js`
Groovy	`groovy`	Отсутствует	`groovy`
SISC Scheme	`scheme`, `sisc`	Отсутствует	`scc`, `sce`, `scm`, `shp`

Обычно требующийся механизм известен и может запрашиваться просто по имени, типу MIME или расширению, как показано в приведенном ниже примере.

```
ScriptEngine engine = manager.getEngineByName("JavaScript");
```

В версии Java SE 7 предоставляется механизм Rhino, который представляет собой разработанный Mozilla Foundation интерпретатор JavaScript. Имеется также возможность внедрять дополнительные языки, указав в пути к классам нужные JAR-файлы. Для этой цели обычно требуются два набора JAR-файлов: один для реализации самого языка сценариев, другой — для реализации механизма, приспосабливающего этот язык под прикладной интерфейс API для создания сценариев. На веб-сайте по адресу `http://java.net/projects/scripting` предлагаются механизмы для самых разных языков сценариев. Чтобы, например, внедрить поддержку Groovy, достаточно указать в пути к классам каталог `groovy/lib/*` (из веб-сайта `http://groovy.codehaus.org`) и файл `groovy-engine.jar` (из веб-сайта `http://java.net/projects/scripting`).

javax.script.ScriptEngineManager 6

- `List<ScriptEngineFactory> getEngineFactories()`

  Получает список всех обнаруженных фабрик механизмов сценариев.

- `ScriptEngine getEngineByName(String` *name*`)`
- `ScriptEngine getEngineByExtension(String` *extension*`)`
- `ScriptEngine getEngineByMimeType(String` *mimeType*`)`

  Получают механизм сценариев с заданным именем, расширением файла сценария или типом MIME.

javax.script.ScriptEngineFactory 6

- `List<String> getNames()`
- `List<String> getExtensions()`
- `List<String> getMimeTypes()`

  Получают имена, расширения файлов сценариев и типы MIME, по которым известна данная фабрика.

Оценка сценариев и привязки

Получив механизм, можно приступить к вызову сценария. Это делается следующим образом:

```
Object result = engine.eval(scriptString);
```

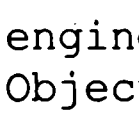
Если сценарий хранится в файле, нужно открыть считывающий поток типа `Reader` и сделать следующий вызов:

```
Object result = engine.eval(reader);
```

С помощью одного и того же механизма можно вызвать целый ряд сценариев. Если какой-нибудь из сценариев содержит определения переменных, функций или

классов, большинство механизмов сценариев будет сохранять их для последующего использования. Так, в приведенном ниже примере кода возвращается значение **1729**.

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```



НА ЗАМЕТКУ! Для того чтобы выяснить, является ли безопасным параллельное выполнение сценариев во многих потоках, достаточно сделать следующий вызов:

```
Object param = factory.getParameter("THREADING");
```

В итоге возвращается одно из перечисленных ниже значений.

- `null`: параллельное выполнение небезопасно.
- `"MULTITHREADED"`: параллельное выполнение безопасно. Результаты выполнения одного потока могут быть доступны из другого потока.
- `"THREAD-ISOLATED"`: то же, что и значение `"MULTITHREADED"`, но только для каждого потока поддерживаются разные привязки переменных.
- `"STATELESS"`: то же, что и значение `"THREAD-ISOLATED"`, но только сценарии не могут изменять привязки переменных.

К механизму сценариев нередко требуется добавлять привязки переменных. Каждая привязка состоит из имени и связываемого объекта Java. Рассмотрим для примера следующие операторы:

```
engine.put(k, 1728);
Object result = engine.eval("k + 1");
```

Код сценария читает определение объекта `k` из привязок в "области действия механизма". И это очень важно, ведь почти все языки сценариев могут получать доступ к объектам Java и зачастую посредством более простого, чем у Java, синтаксиса. Например:

```
engine.put(b, new JButton());
engine.eval("b.text = 'Ok'");
```

С другой стороны, можно извлекать значения переменных, привязанных операторами сценария, как показано ниже.

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

Кроме области действия механизма, существует и глобальная область действия. Любые привязки, которые вводятся в объект типа `ScriptEngineManager`, становятся видимыми для всех механизмов. Вместо того чтобы вводить привязки в глобальную область действия или область действия механизма сценариев, их можно также накапливать в объекте типа `Bindings` и передавать методу `eval()`, как показано ниже. Это очень удобно, если набор привязок не требуется сохранять для последующих вызовов метода `eval()`.

```
Bindings scope = engine.createBindings();
scope.put(b, new JButton());
engine.eval(scriptString, scope);
```

НА ЗАМЕТКУ! Безусловно, может возникнуть потребность иметь и другие области действия, в отличие от глобальной области действия, а также области действия механизма сценариев. Например, веб-контейнеру могут потребоваться области действия запросов и сеансов работы. В подобных случаях разработчикам приходится самостоятельно создавать класс, реализующий интерфейс `ScriptContext`, чтобы управлять набором своих областей действия. Каждая такая область действия должна снабжаться целочисленным номером, а поиск должен выполняться в первую очередь в областях действия с наименьшим номером. (В стандартной библиотеке доступен только класс `SimpleScriptContext`, но он предусматривает лишь глобальную область действия, а также область действия механизма сценариев.)

javax.script.ScriptEngine 6

- `Object eval(String script)`
- `Object eval(Reader reader)`
- `Object eval(String script, Bindings bindings)`
- `Object eval(Reader reader, Bindings bindings)`

  Оценивают сценарий, предоставляемый в символьной строке или средством чтения с учетом заданных привязок.
- `Object get(String key)`
- `void put(String key, Object value)`

  Получают или размещают привязку в области действия механизма сценариев.

javax.script.ScriptEngineManager 6

- `Object get(String key)`
- `void put(String key, Object value)`

  Получают или размещают привязку в глобальной области действия.

javax.script.Bindings 6

- `Object get(String key)`
- `void put(String key, Object value)`

  Получают или *размещают* привязку в области действия, представляемую данным объектом типа `Bindings`.

Переадресация ввода-вывода

Стандартный ввод-вывод в сценарии можно переадресовывать, вызывая метод `setReader()` или `SetWriter()` соответственно в контексте сценария, как показано в приведенном ниже примере. Любые данные, выводимые с помощью таких функций JavaScript, как `print()` или `println()`, направляются объекту `writer`.

```
StringWriter writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

ВНИМАНИЕ! Методу `setWriter()` можно передать любой объект типа `Writer`, но если это не объект типа `PrintWriter`, механизм сценариев Rhino сгенерирует исключение.

Методы `setReader()` и `setWriter()` воздействуют только на стандартные источники ввода-вывода данных в механизме сценариев. Например, при выполнении приведенного ниже кода в сценарии JavaScript перенаправлен будет только первый вывод. Механизму сценариев Rhino ничего неизвестно о стандартном источнике ввода данных, поэтому вызов метода `setReader()` ничего не даст.

```
println("Hello");
java.lang.System.out.println("World");
```

javax.script.ScriptEngine 6

- `ScriptContext getContext()`

  Получает стандартный контекст сценариев для данного механизма.

javax.script.ScriptContext 6

- `Reader getReader()`
- `void setReader(Reader` *reader*`)`
- `Writer getWriter()`
- `void setWriter(Writer` *writer*`)`
- `Writer getErrorWriter()`
- `void setErrorWriter(Writer` *writer*`)`

  Получают или устанавливают средство чтения для вводимых данных или средство записи для обычных или уведомляющих об ошибках выводимых данных.

Вызов сценарных функций и методов

При наличии многих механизмов сценариев функция может вызываться на языке сценариев и без оценки фактического кода сценария. Это удобно, если пользователям разрешается реализовывать службу на избранном ими языке сценариев.

Те механизмы сценариев, которые предоставляют подобные функциональные возможности, реализуют интерфейс `Invocable`. В частности, этот интерфейс реализует механизм сценариев Rhino. Чтобы вызвать сценарную функцию, достаточно обратиться к методу `invokeFunction()`, указав в нем имя и параметры требуемой функции:

```
if (engine implements Invocable)
   ((Invocable) engine).invokeFunction("aFunction", param1, param2);
```

Если же язык сценариев является объектно-ориентированным, этот метод можно вызвать следующим образом:

```
((Invocable) engine).invokeMethod(
   implicitParam, "aMethod", explicitParam1, explicitParam2);
```

В данном случае объект `implicitParam` замещает соответствующий объект в языке сценариев. Это может быть результатом предшествующего вызова механизма сценариев.

НА ЗАМЕТКУ! Даже если механизм сценариев не реализует интерфейс `Invocable`, то метод все равно можно вызвать не зависящим от языка образом. Метод `getMethodCallSyntax()` из класса `ScriptEngineFactory` формирует символьную строку, которую можно затем передать методу `eval()`. Но все параметры этого метода должны быть непременно привязаны к именам, в то время как метод `invokeMethod()` может вызываться с произвольными значениями параметров.

Можно пойти еще дальше и запросить механизм сценариев реализовать интерфейс Java. В этом случае появится возможность вызывать сценарные функции и методы, используя синтаксис Java для вызова методов. И хотя это зависит от конкретного механизма сценариев, как правило, для каждого метода из интерфейса достаточно предоставить соответствующую функцию. Рассмотрим в качестве примера следующий интерфейс Java:

```
public interface Greeter
{
  String greet(String whom);
}
```

Допустим, в механизме сценариев Rhino для этого интерфейса предоставляется следующая функция:

```
function greet(x) { return "Hello, " + x + "!"; }
```

Этот код должен оцениваться первым. Далее может следовать приведенный ниже вызов.

```
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
```

После этого можно сделать простой вызов метода Java следующим образом:

```
String result = g.greet("World");
```

В результате будет подспудно вызываться метод `greet()` из сценария JavaScript. Такой прием похож на вызов удаленного метода, как поясняется в главе 11.

В объектно-ориентированном языке сценариев доступ к классу сценария можно получить через соответствующий интерфейс Java. Рассмотрим в качестве примера следующий код сценария JavaScript, в котором определяется класс `SimpleGreeter`:

```
function SimpleGreeter(salutation) { this.salutation = salutation; }
SimpleGreeter.prototype.greet = function(whom)
    { return this.salutation + ", " + whom + "!"; }
```

Этот класс можно использовать для создания сценариев с разными приветствиями (наподобие `"Hello"`, `"Goodbye"` и т.д.).

НА ЗАМЕТКУ! Более подробно о способах определения классов на JavaScript можно узнать из книги *JavaScript — The Definitive Guide, 5th Edition* Дэвида Флэнагана (David Flanagan; издательство O'Reilly, 2006 г.).

После оценки определения класса JavaScript можно сделать приведенный ниже вызов. При вызове метода `g.greet("World")` для объекта `goodbyeGreeter` в сценарии JavaScript будет вызван метод `greet()`, а в результате возвратится символьная строка `"Goodbye, World!"`.

```
Object goodbyeGreeter = engine.eval("new SimpleGreeter('Goodbye')");
Greeter g = ((Invocable) engine).getInterface(goodbyeGreeter, Greeter.class);
```

Таким образом, интерфейс `Invocable` оказывается очень полезным в тех случаях, когда код сценария требуется вызывать из прикладной программы на Java, не заботясь о синтаксисе языка сценариев.

javax.script.Invocable 6

- `Object invokeFunction(String name, Object... parameters)`
- `Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)`
- Вызывают функцию или метод с указанным именем, передавая заданные параметры.

  `<T> T getInterface(Class<T> iface)`

  Возвращает реализацию указанного интерфейса, методы которого реализуются с помощью функций из механизма сценариев.
- `<T> T getInterface(Object implicitParameter, Class<T> iface)`

  Возвращает реализацию указанного интерфейса, методы которого реализуются с помощью методов заданного объекта.

Компиляция сценариев

Некоторые механизмы сценариев способны компилировать код сценария в промежуточную форму для более эффективного выполнения. Такие механизмы реализуют интерфейс `Compilable`. В следующем примере демонстрируется компилирование и оценивание кода, содержащегося в файле сценария:

```
Reader reader = new FileReader("myscript.js");
CompiledScript script = null;
if (engine implements Compilable)
  CompiledScript script = ((Compilable) engine).compile(reader);
```

После компиляции сценария можно переходить к его выполнению. В приведенном ниже фрагменте кода демонстрируется выполнение скомпилированного кода сценария, если компиляция прошла успешно, а иначе — исходного сценария, если окажется, что механизм сценариев не поддерживает компиляцию. Безусловно, компилировать сценарий нужно лишь в том случае, если его требуется выполнить повторно.

```
if (script != null)
  script.eval();
else
  engine.eval(reader);
```

javax.script.Compilable 6

- `CompiledScript compile(String script)`
- `CompiledScript compile(Reader reader)`

  Компилируют сценарий, задаваемый символьной строкой или средством чтения.

javax.script.CompiledScript 6

- `Object eval()`
- `Object eval(Bindings bindings)`

  Оценивают данный сценарий.

Пример создания сценария для обработки событий в ГПИ

Чтобы продемонстрировать возможности прикладного интерфейс API для создания сценариев, рассмотрим пример программы, позволяющей пользователям задавать обработчики событий на избранном языке сценариев.

Проанализируйте исходный код, приведенный в листинге 10.1. В этом коде средства создания сценариев вводятся в класс произвольного фрейма. По умолчанию это класс `ButtonFrame` из листинга 10.2, аналогичный по своим функциям программе обработки событий, демонстрировавшейся в первом томе данной книги, но с двумя отличиями:

- у каждого компонента имеется свой собственный набор свойств `name`;
- отсутствуют обработчики событий.

Требующиеся обработчики событий определяются в файле свойств. А каждое свойство определяется в следующей форме:

имяКомпонента.имяСобытия = кодСценария

Так, если пользователь выбирает языка сценариев JavaScript, требующиеся обработчики событий предоставляются в файле `js.properties` приведенным ниже образом. В сопутствующем коде имеются также файлы для Groovy и SISC Scheme.

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
blueButton.action=panel.background = java.awt.Color.BLUE
redButton.action=panel.background = java.awt.Color.RED
```

Рассматриваемая здесь программа начинается с загрузки механизма сценариев для языка, указываемого в командной строке. Если же язык не указан, то по умолчанию выбирается JavaScript.

Далее выполняется обработка сценария `init.`*`язык`*, если таковой имеется. В целом такой подход кажется вполне приемлемым. К тому же интерпретатор Scheme нуждается в ряде громоздких операций инициализации, которые вряд ли стоит включать в каждый сценарий для обработки событий.

После этого осуществляется рекурсивный обход всех дочерних компонентов и ввод привязок (*имя*, *объект*) в область действия механизма сценариев. Далее выполняется чтение из файла *язык*`.`*свойства*. Для каждого свойства конструируется прокси-объект,

замещающий обработчик событий, который, собственно, и заставляет выполняться код сценария. Подробности реализации механизма замещения носят несколько технический характер, поэтому тем, кто желает разобраться в нем, рекомендуется еще раз прочитать раздел, посвященный прокси-объектам в главе 6 первого тома данной книги, а также раздел о событиях в компонентах JavaBeans из главы 8 этого тома. Но самое главное, что каждый обработчик событий вызывает следующий метод:

```
engine.eval(scriptCode);
```

Остановимся подробнее на обработке событий от кнопки выбора желтого цвета фона (объекте `yellowButton`). При обработке приведенной ниже строки кода обнаруживается компонент `JButton` под именем `"yellowButton"`.

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

Далее к этому компоненту присоединяется объект типа `ActionListener` с методом `actionPerformed()`, который выполняет сценарий, как показано ниже. Механизм сценариев содержит привязку, которая связывает имя `"panel"` с объектом типа `JPanel`. Когда наступает событие, выполняется метод `setBackground()` для этого объекта, а в итоге изменяется цвет фона панели.

```
panel.background = java.awt.Color.YELLOW
```

Запустить рассматриваемую здесь программу с обработчиками событий из сценария JavaScript можно, выполнив команду

```
java ScriptTest
```

А для того чтобы использовать обработчики событий из сценария Groovy, придется выполнить такую команду:

```
java -classpath .:groovy/lib/*:jsr223-engines/groovy/build/groovy-engine.jar
     ScriptTest groovy
```

где `groovy` — каталог, в котором установлен Groovy, а `jsr223-engines` — каталог, в котором находятся адаптеры механизма сценариев, загружаемые с веб-сайта по адресу `http://java.net/projects/scripting`. И наконец, чтобы опробовать Scheme, следует загрузить SISC Scheme из веб-сайта по адресу `http://sisc-scheme.org/` и выполнить команду

```
java -classpath .:sisc/*:jsr223-engines/scheme/build/scheme-engine.jar
     ScriptTest scheme
```

В данной программе демонстрируется применение сценариев при программировании ГПИ на платформе Java. Желающие могут пойти еще дальше и описать ГПИ с помощью XML-файла, как было показано в главе 2. В этом случае данная программа превратится в интерпретатор для ГПИ с визуальным представлением, определяемым в формате XML, а также поведением, определяемым на языке сценариев. Это очень похоже на среду создания динамических серверных сценариев и динамических HTML-страниц.

Листинг 10.1. Исходный код из файла `script/ScriptTest.java`

```
1   package script;
2
3   import java.awt.*;
4   import java.beans.*;
```

```
5   import java.io.*;
6   import java.lang.reflect.*;
7   import java.util.*;
8   import javax.script.*;
9   import javax.swing.*;
10
11  /**
12   * @version 1.01 2012-01-28
13   * @author Cay Horstmann
14  */
15  public class ScriptTest
16  {
17    public static void main(final String[] args)
18    {
19      EventQueue.invokeLater(new Runnable()
20        {
21          public void run()
22          {
23            try
24            {
25              ScriptEngineManager manager = new ScriptEngineManager();
26              String language;
27              if (args.length == 0)
28              {
29                System.out.println("Available factories: ");
30                for (ScriptEngineFactory factory :
31                     manager.getEngineFactories())
32                  System.out.println(factory.getEngineName());
33
34                language = "js";
35              }
36              else language = args[0];
37
38              final ScriptEngine engine =
39                 manager.getEngineByName(language);
40              if (engine == null)
41              {
42                System.err.println("No engine for " + language);
43                System.exit(1);
44              }
45
46              final String frameClassName =
47                 args.length < 2 ? "buttons1.ButtonFrame" : args[1];
48
49              JFrame frame = (JFrame)
50                 Class.forName(frameClassName).newInstance();
51              InputStream in = frame.getClass().getResourceAsStream(
52                 "init." + language);
53              if (in != null) engine.eval(new InputStreamReader(in));
54              getComponentBindings(frame, engine);
55
56              final Properties events = new Properties();
57              in = frame.getClass().getResourceAsStream(
58                 language + ".properties");
59              events.load(in);
60
61              for (final Object e : events.keySet())
62              {
```

```
63                String[] s = ((String) e).split("\\.");
64                addListener(s[0], s[1], (String) events.get(e), engine);
65              }
66              frame.setTitle("ScriptTest");
67              frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
68              frame.setVisible(true);
69            }
70            catch (ReflectiveOperationException | IOException
71               | ScriptException | IntrospectionException ex)
72            {
73              ex.printStackTrace();
74            }
75          }
76        });
77    }
78
79    /**
80     * Собирает все именованные компоненты в контейнер
81     * @param c Компонент
82     * @param namedComponents
83    */
84    private static void getComponentBindings(
85       Component c, ScriptEngine engine)
86    {
87      String name = c.getName();
88      if (name != null) engine.put(name, c);
89      if (c instanceof Container)
90      {
91        for (Component child : ((Container) c).getComponents())
92          getComponentBindings(child, engine);
93      }
94    }
95
96    /**
97     * Вводит в объект приемник событий, метод которого выполняет сценарий
98     * @param beanName Имя компонента JavaBeans, в который вводится
99     *                 приемник событий
100    * @param eventName Название типа приемника событий, например,
101    *                 "action" (действие) или "change" (изменение)
102    * @param scriptCode Выполняемый код сценария
103    * @param engine Механизм сценариев, выполняющий код сценария
104    * @param bindings Привязки для выполнения сценария
105    * @throws Генерирует исключение типа IntrospectionException
106   */
107   private static void addListener(String beanName, String eventName,
108      final String scriptCode, final ScriptEngine engine)
109      throws ReflectiveOperationException, IntrospectionException
110   {
111     Object bean = engine.get(beanName);
112     EventSetDescriptor descriptor =
113        getEventSetDescriptor(bean, eventName);
114     if (descriptor == null) return;
115     descriptor.getAddListenerMethod().invoke(bean,
116        Proxy.newProxyInstance(null, new Class[]
117        { descriptor.getListenerType() },
118        new InvocationHandler()
119          {
120            public Object invoke(
```

```
                Object proxy, Method method, Object[] args)
                throws Throwable
            {
              engine.eval(scriptCode);
              return null;
            }
          }));
  }

  private static EventSetDescriptor getEventSetDescriptor(
     Object bean, String eventName)
     throws IntrospectionException
  {
    for (EventSetDescriptor descriptor :
         Introspector.getBeanInfo(bean.getClass())
         .getEventSetDescriptors())
      if (descriptor.getName().equals(eventName)) return descriptor;
    return null;
    }
}
```

Листинг 10.2. Исходный код из файла `buttons1/ButtonFrame.java`

```
1  package buttons1;
2
3  import javax.swing.*;
4
5  public class ButtonFrame extends JFrame
6  {
7    private static final int DEFAULT_WIDTH = 300;
8    private static final int DEFAULT_HEIGHT = 200;
9
10    private JPanel panel;
11    private JButton yellowButton;
12    private JButton blueButton;
13    private JButton redButton;
14
15    public ButtonFrame()
16    {
17      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19      panel = new JPanel();
20      panel.setName("panel");
21      add(panel);
22
23      yellowButton = new JButton("Yellow");
24      yellowButton.setName("yellowButton");
25      blueButton = new JButton("Blue");
26      blueButton.setName("blueButton");
27      redButton = new JButton("Red");
28      redButton.setName("redButton");
29
30      panel.add(yellowButton);
31      panel.add(blueButton);
32      panel.add(redButton);
33    }
34 }
```

Прикладной интерфейс API для компилятора

В предыдущих разделах было показано, как взаимодействовать с кодом на языке сценариев. А теперь рассмотрим другой вариант: программы на Java, компилирующие код Java. Имеется немало инструментальных средств, в которых требуется вызывать компилятор Java. К их числу относятся следующие.

- Среды разработки.
- Программы, обучающие программированию на Java.
- Автоматизированные средства разработки и тестирования.
- Инструментальные средства построения по шаблону, обрабатывающие фрагменты кода Java, например страницы JSP (JavaServer Pages).

Раньше приложения вызывали компилятор Java, обращаясь к недокументированным классам из библиотеки *jdk*/lib/tools.jar. Начиная с версии Java SE 6 общедоступный прикладной интерфейс API для компиляции является частью платформы Java, и поэтому пользоваться библиотекой tools.jar больше не требуется. Об этом прикладном интерфейсе и пойдет речь в данном разделе.

Простой способ компилирования

Компилятор вызывается очень просто, как показано в приведенном ниже примере. Получаемое в итоге нулевое значение указывает на то, что компиляция прошла успешно.

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
OutputStream outStream = ...;
OutputStream errStream = ...;
int result = compiler.run(null, outStream, errStream, "-sourcepath",
    "src", "Test.java");
```

Все выводимые данные и сообщения об ошибках компилятор направляет в указанные потоки вывода. В качестве параметров метода run() можно указывать и пустое значение null. В этом случае используются стандартные потоки ввода-вывода System.out и System.err. Первый параметр метода run() обозначает поток ввода, но, поскольку никаких данных, вводимых с консоли, компилятор не принимает, значение этого параметра всегда оставляется пустым (null). Сам же метод run() наследуется из обобщенного интерфейса Tool, допускающего применение инструментальных средств для чтения вводимых данных. А остальные параметры метода run() являются просто аргументами, которые следовало было бы передать утилите javac, если бы этот метод вызывался из командной строки. Они могут обозначать как параметры командной строки, так и имена файлов.

Выполнение заданий на компиляцию

С помощью объекта типа CompilationTask можно получить еще больший контроль над процессом компиляции. Он, в частности, позволяет выполнять следующие действия.

- Управлять источником программного кода, предоставляя, например, код не в файле, а в построителе символьных строк.

- Управлять размещением файлов классов, сохраняя их, например, в базе данных.
- Принимать предупреждения и сообщения об ошибках, появляющиеся во время компиляции.
- Запускать компилятор на выполнение в фоновом режиме.

За размещение исходных файлов и файлов классов отвечает интерфейс `JavaFileManager`. В его обязанности входит определение для этих файлов экземпляров типа `JavaFileObject`. Каждый объект типа `JavaFileObject` может соответствовать какому-нибудь определенному файлу на диске или предоставлять какой-нибудь другой механизм для чтения и записи содержимого файла.

Для приема появляющихся сообщений об ошибках устанавливается интерфейс `DiagnosticListener`. Всякий раз, когда компилятор выдает предупреждение или сообщение об ошибке, этот приемник получает объект типа `Diagnostic`. Этот интерфейс реализуется классом `DiagnosticCollector`. Он просто собирает все диагностические данные для просмотра и анализа по завершении компиляции.

Объект типа `Diagnostic` содержит сведения о месте появления ошибки компиляции, включая имя файла, номер строки и столбца, а также удобочитаемое описание ошибки. А для получения объекта типа `CompilationTask` вызывается метод `getTask()` из класса `JavaCompiler` с указанием следующих аргументов.

- Объект типа `Writer` для вывода любых данных компилятора, не являющихся диагностическими (типа `Diagnostic`), или пустого значения `null` в стандартный поток `System.err`.
- Объект типа `JavaFileManager` или пустое значение `null` для использования стандартного диспетчера файлов компилятора.
- Объект типа `DiagnosticListener`.
- Символьная строка параметров командной строки или пустое значение `null`, если никаких параметров указывать не требуется.
- Имена классов для обработки аннотаций или пустые значения `null`, если аннотации отсутствуют (подробнее обработка аннотаций рассматривается далее в этой главе).
- Экземпляры типа `JavaFileObject` для исходных файлов.

Последние три аргумента необходимо предоставить в виде объектов типа `Iterable`. Например, последовательность параметров командной можно указать следующим образом:

```
Iterable<String> options = Arrays.asList("-g", "-d", "classes");
```

С другой стороны, можно воспользоваться любым из классов коллекций. Если же требуется, чтобы компилятор считывал исходные файлы с диска, то можно запросить интерфейс `StandardJavaFileManager` преобразовать символьные строки с именами файлов или объекты типа `File` в экземпляры типа `JavaFileObject`, как показано в приведенном ниже примере.

```
StandardJavaFileManager fileManager =
    compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> fileObjects =
    fileManager.getJavaFileObjectsFromStrings(fileNames);
```

Но если требуется, чтобы компилятор считывал исходный код из какого-нибудь другого места, а не из файла на диске, то придется предоставить свой собственный подкласс интерфейса `JavaFileObject`. В листинге 10.3 приведен исходный код класса для получения объекта исходного файла с данными, содержащимися в объекте типа `StringBuilder`. Этот класс расширяет простой класс `SimpleJavaFileObject` и переопределяет метод `getCharContent()`, чтобы возвращать содержимое построителя символьных строк. Он применяется в рассматриваемом здесь примере программы, где код класса Java сначала генерируется динамически, а затем компилируется.

Класс `CompilationTask` реализует интерфейс `Callable<Boolean>`. Его можно передать классу `Executor` для выполнения в другом потоке или просто вызвать с помощью метода `call()`. Возвращаемое значение `Boolean.FALSE` обозначает неудачный исход компиляции. Применение данного класса демонстрируется в приведенном ниже фрагменте кода.

```
Callable<Boolean> task = new JavaCompiler.CompilationTask(
    null, fileManager, diagnostics, options, null, fileObjects);
if (!task.call())
    System.out.println("Compilation failed");
```

Если только требуется, чтобы компилятор создавал файлы классов и сохранял их на диске, то выполнять специальную настройку интерфейса `JavaFileManager` не нужно. Тем не менее в рассматриваемой здесь программе файлы классов формируются сначала в виде байтовых массивов, а затем считываются из памяти с помощью специального загрузчика классов. В листинге 10.4 приведен исходный код класса, реализующего интерфейс `JavaFileObject`. Его метод `openOutputStream()` возвращает поток вывода типа `ByteArrayOutputStream`, в который компилятор направляет байт-коды.

Оказывается, что предписать диспетчеру файлов компилятора использовать упомянутые выше объекты файлов не так-то просто. Дело в том, что в стандартной библиотеке отсутствует класс, который реализовывал бы интерфейс `StandardJavaFileManager`. Поэтому приходится создавать подкласс, производный от класса `ForwardingJavaFileManager` и делегирующий все вызовы заданному диспетчеру файлов. В данном примере требуется лишь изменить метод `getJavaFileForOutput()`, и достичь этой цели можно с помощью следующего кода:

```
JavaFileManager fileManager =
    compiler.getStandardFileManager(diagnostics, null, null);
fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
{
  public JavaFileObject getJavaFileForOutput(
     Location location, final String className, Kind kind,
     FileObject sibling) throws IOException
  {
     возвратить специальный объект файла
  }
};
```

Таким образом, вызывать метод `run()` для задания на компиляцию типа `JavaCompiler` удобно только в тех случаях, когда требуется активизировать компилятор обычным образом и выполнять операции чтения и записи файлов на диск. Выводимые данные и сообщения об ошибках можно перехватывать, но выполнять их синтаксический анализ придется самостоятельно.

Если же требуется больший контроль над процессами обработки файлов и уведомления об ошибках, то лучше воспользоваться классом `CompilationTask`. Его прикладной интерфейс API довольно сложен, но в то же время он позволяет управлять каждым аспектом процесса компиляции.

Листинг 10.3. Исходный код из файла `compiler/StringBuilderJavaSource.java`

```
1  package compiler;
2
3  import java.net.*;
4  import javax.tools.*;
5
6  /**
7   * Источник, хранящий код Java в построителе символьных строк
8   * @version 1.00 2007-11-02
9   * @author Cay Horstmann
10 */
11 public class StringBuilderJavaSource extends SimpleJavaFileObject
12 {
13   private StringBuilder code;
14
15   /**
16    * Конструирует новый объект типа StringBuilderJavaSource
17    * @param name Имя исходного файла, представленного данным объектом
18   */
19   public StringBuilderJavaSource(String name)
20   {
21     super(URI.create("string:///" + name.replace('.', '/')
22        + Kind.SOURCE.extension), Kind.SOURCE);
23     code = new StringBuilder();
24   }
25
26   public CharSequence getCharContent(boolean ignoreEncodingErrors)
27   {
28     return code;
29   }
30
31   public void append(String str)
32   {
33     code.append(str);
34     code.append('\n');
35   }
36 }
```

Листинг 10.4. Исходный код из файла `compiler/ByteArrayJavaClass.java`

```
1  package compiler;
2
3  import java.io.*;
4  import java.net.*;
5  import javax.tools.*;
6  /**
7   * Класс Java для хранения байт-кодов в байтовом массиве
8   * @version 1.00 2007-11-02
9   * @author Cay Horstmann
```

```
10 */
11 public class ByteArrayJavaClass extends SimpleJavaFileObject
12 {
13   private ByteArrayOutputStream stream;
14
15   /**
16    * Конструирует новый объект типа ByteArrayJavaClass
17    * @param name Имя файла класса, представленного данным объектом
18   */
19   public ByteArrayJavaClass(String name)
20   {
21     super(URI.create("bytes:///" + name), Kind.CLASS);
22     stream = new ByteArrayOutputStream();
23   }
24
25   public OutputStream openOutputStream() throws IOException
26   {
27     return stream;
28   }
29
30   public byte[] getBytes()
31   {
32     return stream.toByteArray();
33   }
34 }
```

javax.tools.Tool 6

- `int run(InputStream` *in*`, OutputStream` *out*`, OutputStream` *err*`, String...` *arguments*`)`

  Запускает утилиту компиляции с указанными потоками ввода и вывода, а также потоком вывода сообщений об ошибках и аргументами командной строки. При удачном исходе компиляции возвращает нулевое значение, а при неудачном исходе — ненулевое.

javax.tools.JavaCompiler 6

- `StandardJavaFileManager getStandardFileManager(DiagnosticListener<? super JavaFileObject>` *diagnosticListener*`, Locale` *locale*`, Charset` *charset*`)`

  Получает стандартный диспетчер файлов для данного компилятора. Имеется возможность использовать стандартные сообщения об ошибках, региональные настройки и набор символов, указав в качестве соответствующих параметров пустые значения `null`.

- `JavaCompiler.CompilationTask getTask(Writer` *out*`, JavaFileManager` *fileManager*`, DiagnosticListener<? super JavaFileObject>` *diagnosticListener*`, Iterable<String>` *options*`, Iterable<String>` *classesForAnnotationProcessing*`, Iterable<? extends JavaFileObject>` *sourceFiles*`)`

  Получает задание на компиляцию, при вызове которого будут компилироваться указанные исходные файлы. Подробнее об этом см. в предыдущем разделе.

javax.tools.StandardJavaFileManager 6

- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings (Iterable<String> fileNames)`
- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles (Iterable<? extends File> files)`

  Преобразуют последовательность имен файлов или файлов в последовательность экземпляров типа `JavaFileObject`.

javax.tools.JavaCompiler.CompilationTask 6

- `Boolean call()`
- Выполняет задание на компиляцию.

javax.tools.DiagnosticCollector<S> 6

- `DiagnosticCollector()`

  Создает пустой сборщик данных.
- `List<Diagnostic<? extends S>> getDiagnostics()`

  Получает собранные диагностические данные.

javax.tools.Diagnostic<S> 6

- `S getSource()`

  Получает исходный объект, связанный с данной процедурой диагностики.
- `Diagnostic.Kind getKind()`

  Получает тип данной процедуры диагностики, принимающий значение одной из следующих констант: `ERROR`, `WARNING`, `MANDATORY_WARNING`, `NOTE` или `OTHER`.
- `String getMessage(Locale locale)`

  Получает сообщение, описывающее ошибку, обнаруженную в данной процедуре диагностики. Имеется возможность использовать стандартные региональные настройки, передав пустое значение `null` в качестве соответствующего параметра.
- `long getLineNumber()`
- `long getColumnNumber()`

  Получают местоположение ошибки, обнаруженной в данной процедуре диагностики.

javax.tools.SimpleJavaFileObject 6

- `CharSequence getCharContent(boolean ignoreEncodingErrors)`

  Этот метод переопределяется для объекта файла, представляющего исходный файл и генерирующего исходный код.

- `OutputStream openOutputStream()`

  Этот метод переопределяется для объекта файла, представляющего файл класса и формирующего поток вывода, в который можно направлять генерируемые байт-коды.

javax.tools.ForwardingJavaFileManager<M extends JavaFileManager> 6

- `protected ForwardingJavaFileManager(M fileManager)`

  Создает объект типа `JavaFileManager`, делегирующий все вызовы указанному диспетчеру файлов.

- `FileObject getFileForOutput(JavaFileManager.Location location, String className, JavaFileObject.Kind kind, FileObject sibling)`

  Вызов этого метода перехватывается, если требуется заменить объект файла для записи файлов классов. Параметр `kind` может принимать значение одной из следующих констант: `SOURCE`, `CLASS`, `HTML` или `OTHER`.

Пример динамического генерирования кода Java

В технологии JSP для динамических веб-страниц допускается сочетание разметки в коде HTML с фрагментами кода Java, как показано в следующем примере The current date and time is <%= new java.util.Date() %>.

Механизм JSP динамически компилирует код Java в сервлет. В рассматриваемой здесь прикладной программе демонстрируется более простой пример динамического генерирования кода Swing. Основной замысел состоит в том, чтобы воспользоваться построителем ГПИ с целью расположить компоненты во фрейме и обозначить их поведение во внешнем файле. В листинге 10.5 приведен очень простой пример класса фрейма, а в листинге 10.6 — код для обозначения действий кнопок. Следует заметить, что конструктор класса фрейма вызывает абстрактный метод `addEventHandlers()`. А генератор кода создает подкласс, реализующий метод `addEventHandlers()`, вводя приемник для обработки действий в каждой строке из файла `action.properties`. (Распространение возможностей генерирования кода на другие виды событий оставляется читателю в качестве упражнения для самостоятельной проработки.)

Этот подкласс размещается в пакете под именем `x`, которое нигде больше не должно использоваться в данной программе. Сгенерированный код принимает следующую форму:

```
package x;
public class Frame extends ИмяСуперкласса {
  protected void addEventHandlers() {
    ИмяКомпонента1.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(java.awt.event.ActionEvent) {
        код первого обработчика событий
      } } );
```

```
      // повторить для остальных обработчиков событий ...
} }
```

Метод `buildSource()` в программе из листинга 10.7 служит для генерирования этого кода и его размещения в объекте типа `StringBuilderJavaSource`. А затем этот объект передается компилятору Java. Построение объектов типа `ByteArrayJavaClass` для каждого класса из пакета `x` осуществляется методом `getJavaFileForOutput()` из класса `ForwardingJavaFileManager`. Эти объекты перехватывают файлы классов, формируемые при компиляции класса `x.Frame`. Метод `getJavaFileForOutput()` вводит каждый объект файла в список прежде, чем возвратить его, чтобы в дальнейшем можно было обнаружить байт-коды. Следует иметь в виду, что компиляция класса `x.Frame` приводит к генерированию файла для главного класса, а также отдельных файлов для каждого класса приемника событий.

После компиляции составляется отображение, в котором имена классов сопоставляются с массивами байт-кодов. Классы, хранящиеся в этом отображении, загружаются простым загрузчиком классов из листинга 10.8. Сначала происходит обращение к этому загрузчику для загрузки только что скомпилированного класса, а затем создается и отображается класс фрейма прикладной программы:

```
ClassLoader loader = new MapClassLoader(byteCodeMap);
Class<?> cl = loader.loadClass("x.Frame");
Frame frame = (JFrame) cl.newInstance();
frame.setVisible(true);
```

Если щелкнуть на кнопках, цвет фона изменится обычным образом. А для того чтобы убедиться, что действия кнопок действительно компилируются динамически, измените какую-нибудь из строк в файле `action.properties`, например, следующим образом:

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW);
    yellowButton.setEnabled(false);
```

После этого запустите еще раз данную программу на выполнение. Если теперь щелкнуть на кнопке Yellow, она станет недоступной. Загляните также в каталоги с исходным кодом. Вы не обнаружите там ни исходных файлов, ни файлов классов из пакета `x`. Данный пример наглядно демонстрирует применение динамической компиляции вместе с сохранением файлов исходного кода и классов в оперативной памяти.

Листинг 10.5. Исходный код из файла `buttons2/ButtonFrame.java`

```
1  package buttons2;
2  import javax.swing.*;
3
4  /**
5   * @version 1.00 2007-11-02
6   * @author Cay Horstmann
7  */
8  public abstract class ButtonFrame extends JFrame
9  {
10   public static final int DEFAULT_WIDTH = 300;
11   public static final int DEFAULT_HEIGHT = 200;
12
13   protected JPanel panel;
14   protected JButton yellowButton;
15   protected JButton blueButton;
```

```
16   protected JButton redButton;
17
18   protected abstract void addEventHandlers();
19
20   public ButtonFrame()
21   {
22     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24     panel = new JPanel();
25     add(panel);
26
27     yellowButton = new JButton("Yellow");
28     blueButton = new JButton("Blue");
29     redButton = new JButton("Red");
30
31     panel.add(yellowButton);
32     panel.add(blueButton);
33     panel.add(redButton);
34
35     addEventHandlers();
36   }
37 }
```

Листинг 10.6. Исходный код из файла `buttons2/action.properties`

```
1 yellowButton=panel.setBackground(java.awt.Color.YELLOW);
2 blueButton=panel.setBackground(java.awt.Color.BLUE);
```

Листинг 10.7. Исходный код из файла `compiler/CompilerTest.java`

```
1   package compiler;
2
3   import java.awt.*;
4   import java.io.*;
5   import java.util.*;
6   import java.util.List;
7   import javax.swing.*;
8   import javax.tools.*;
9   import javax.tools.JavaFileObject.*;
10
11  /**
12   * @version 1.00 2007-10-28
13   * @author Cay Horstmann
14  */
15  public class CompilerTest
16  {
17    public static void main(final String[] args) throws IOException,
18       ClassNotFoundException
19    {
20      JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
21
22      final List<ByteArrayJavaClass> classFileObjects = new ArrayList<>();
23
24      DiagnosticCollector<JavaFileObject> diagnostics =
25         new DiagnosticCollector<>();
26
```

```
27      JavaFileManager fileManager = compiler.getStandardFileManager(
28         diagnostics, null, null);
29      fileManager = new ForwardingJavaFileManager<JavaFileManager>(
30         fileManager)
31        {
32          public JavaFileObject getJavaFileForOutput(Location location,
33             final String className, Kind kind, FileObject sibling)
34             throws IOException
35          {
36            if (className.startsWith("x."))
37            {
38              ByteArrayJavaClass fileObject =
39                 new ByteArrayJavaClass(className);
40              classFileObjects.add(fileObject);
41              return fileObject;
42            }
43            else return super.getJavaFileForOutput(
44               location, className, kind, sibling);
45          }
46        };
47
48      String frameClassName = args.length ==
49         0 ? "buttons2.ButtonFrame" : args[0];
50      JavaFileObject source = buildSource(frameClassName);
51      JavaCompiler.CompilationTask task = compiler.getTask(
52         null, fileManager, diagnostics, null,
53         null, Arrays.asList(source));
54      Boolean result = task.call();
55
56      for (Diagnostic<? extends JavaFileObject> d :
57         diagnostics.getDiagnostics())
58      System.out.println(d.getKind() + ": " + d.getMessage(null));
59      fileManager.close();
60      if (!result)
61      {
62        System.out.println("Compilation failed.");
63        System.exit(1);
64      }
65
66      EventQueue.invokeLater(new Runnable()
67        {
68          public void run()
69          {
70            try
71            {
72              Map<String, byte[]> byteCodeMap = new HashMap<>();
73              for (ByteArrayJavaClass cl : classFileObjects)
74                 byteCodeMap.put(cl.getName().substring(1), cl.getBytes());
75              ClassLoader loader = new MapClassLoader(byteCodeMap);
76              JFrame frame =
77                 (JFrame) loader.loadClass("x.Frame").newInstance();
78              frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
79              frame.setTitle("CompilerTest");
80              frame.setVisible(true);
81            }
82            catch (Exception ex)
83            {
84              ex.printStackTrace();
```

```
85            }
86          }
87        });
88    }
89
90    /*
91     * Генерирует исходный код подкласса, реализующего
92     * метод addEventHandlers()
93     * @return Возвращает объект файла, содержащий исходный
94     *         код в построителе символьных строк
95    */
96    static JavaFileObject buildSource(String superclassName)
97       throws IOException, ClassNotFoundException
98    {
99      StringBuilderJavaSource source =
100        new StringBuilderJavaSource("x.Frame");
101     source.append("package x;\n");
102     source.append("public class Frame extends "
103        + superclassName + " {");
104     source.append("protected void addEventHandlers() {");
105     final Properties props = new Properties();
106     props.load(Class.forName(superclassName).getResourceAsStream(
108        "action.properties"));
109     for (Map.Entry<Object, Object> e : props.entrySet())
110     {
111       String beanName = (String) e.getKey();
112       String eventCode = (String) e.getValue();
113       source.append(beanName + ".addActionListener(
114          new java.awt.event.ActionListener() {");
115       source.append("public void actionPerformed(
116          java.awt.event.ActionEvent event) {");
117       source.append(eventCode);
118       source.append("} } );");
119     }
120     source.append("} }");
121     return source;
122   }
123 }
```

Листинг 10.8. Исходный код из файла `compiler/MapClassLoader.java`

```
1  package compiler;
2
3  import java.util.*;
4
5  /**
6   * Загрузчик классов, загружающий классы из отображения,
7   * где в качестве ключей служат имена классов, а соответствующих
8   * им значений — массивы байт-кодов
9   * @version 1.00 2007-11-02
10  * @author Cay Horstmann
11 */
12 public class MapClassLoader extends ClassLoader
13 {
14   private Map<String, byte[]> classes;
15
16   public MapClassLoader(Map<String, byte[]> classes)
```

```
17    {
18      this.classes = classes;
19    }
20
21    protected Class<?> findClass(String name) throws ClassNotFoundException
22    {
23      byte[] classBytes = classes.get(name);
24      if (classBytes == null) throw new ClassNotFoundException(name);
25      Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
26      if (cl == null) throw new ClassNotFoundException(name);
27      return cl;
28    }
29 }
```

Применение аннотаций

Аннотациями называются дескрипторы, которые разработчики вставляют в свой исходный код, чтобы их можно было обработать соответствующими инструментальными средствами. Эти инструментальные средства могут работать как на уровне исходного кода, так и на уровне файлов классов, в которых компилятор размещает аннотации. Аннотации не влияют на способ компиляции программ. Компилятор Java генерирует одинаковые инструкции виртуальной машины как с аннотациями, так и без них.

Для того чтобы извлечь наибольшую пользу из аннотаций, нужно выбрать подходящее *средство обработки*. В исходный код следует вводить такие аннотации, которые распознаются избранным средством обработки, способным правильно интерпретировать их и выполнять соответствующие действия над исходных кодом. У аннотаций существует немало областей применения, и поэтому их универсальность может поначалу вызывать недоразумения. Ниже перечислены некоторые из областей применения аннотаций.

- Автоматическое генерирование вспомогательных файлов, например, файлов дескрипторов развертывания или классов информации о компонентах JavaBeans.
- Автоматическое генерирование кода для тестирования, регистрации, семантической обработки транзакций и т.д.

Итак, начнем обсуждение аннотаций с основных понятий и продемонстрируем их практическое применение на конкретном примере, пометив методы как приемники событий для компонентов AWT и представив обработчик аннотаций, способный анализировать аннотации и подключать приемники событий. Далее мы подробно рассмотрим синтаксические правила. И в завершение главы мы продемонстрируем два более сложных примера обработки аннотаций: первый — на уровне исходного кода, а второй — на уровне файлов классов благодаря применению библиотеки Apache Bytecode Engineering Library и вставке в аннотированные методы дополнительных байт-кодов.

Ниже приведен пример простой аннотации. Аннотация `@Test` помечает метод `checkRandomInsertions()`.

```
public class MyClass
{
  . . .
  @Test public void checkRandomInsertions()
}
```

В Java аннотация применяется подобно модификатору и размещается перед аннотируемым элементом *без* точки с запятой. (*Модификатор* — это ключевое слово вроде `public` или `static`.) Перед именем каждой аннотации ставится знак `@` подобно тому, как это делается в комментариях к документам, формируемым средствами Javadoc. Но комментарии в Javadoc размещаются между разделителями `/**...*/`, тогда как аннотации являются частью исходного кода.

Сама аннотация `@Test` ничего не делает. Для того чтобы приносить какую-то пользу, ей необходимо подходящее инструментальное средство. Например, инструментальное средство тестирования JUnit 4 (доступное по адресу `http://junit.org`) способно вызвать все помеченные аннотацией `@Test` методы при тестировании класса. Другое инструментальное средство может удалять все тестовые методы из файла класса, чтобы исключить их из исходного кода программы после ее тестирования. Аннотации могут быть определены вместе со своими *элементами*, как показано в приведенном ниже примере.

```
@Test(timeout="10000")
```

Эти элементы могут обрабатываться инструментальными средствами, читающими аннотации. Элементы могут выглядеть и по-другому. Подробнее о них речь пойдет далее в этой главе. Помимо методов, снабжаться аннотациями могут также классы, поля и локальные переменные, а сами аннотации могут размещаться на тех же местах, где и модификаторы типа `public` или `static`.

Каждая аннотация должна определяться с помощью *интерфейса аннотаций*. Методы такого интерфейса должны соответствовать элементам определяемой им аннотации. Например, аннотация `Test` для блочного тестирования средствами JUnit определяется с помощью такого интерфейса:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test
{
  long timeout() default 0L;
  . . .
}
```

Объявление `@interface` создает конкретный интерфейс Java, а инструментальные средства, обрабатывающие аннотации, получают объекты, классы которых реализуют сам интерфейс аннотаций. Например, для извлечения элемента `timeout` из конкретной аннотации `Test` инструментальное средство вызывает метод `timeout()`.

Аннотации `Target` и `Retention` являются *мета-аннотациями*. Они помечают аннотацию `Target`, превращая ее в такую аннотацию, которая может применяться только к методам и должна сохраняться при загрузке файла класса в виртуальную машину. Более подробно они рассматриваются далее, в разделе "Мета-аннотации".

Итак, в этом разделе были представлены основные понятия метаданных и аннотаций к программам. А в следующем разделе будет рассмотрен конкретный пример обработки аннотаций.

Пример аннотирования обработчиков событий

Одной из наиболее утомительных задач в программировании пользовательских интерфейсов является присоединение приемников к источникам событий. Многие приемники событий имеют следующий вид:

```
myButton.addActionListener(new
  ActionListener()
  {
    public void actionPerformed(ActionEvent event)
    {
      doSomething();
    }
  });
```

Во избежание подобной утомительной работы в этом разделе будет разработана специальная аннотация. Определение этой аннотации приведено далее в листинге 10.9, а ниже — ее общая форма.

```
@ActionListenerFor(source="myButton") void doSomething() { . . . }
```

Разработчику больше не нужно делать вызовы метода `addActionListener()`. Вместо этого каждый метод приемника событий помечается соответствующей аннотацией. В листинге 10.10 представлен класс `ButtonFrame`, рассматривавшийся в главе 8 первого тома данной книги и реализованный заново вместе с подобными аннотациями. Для этого придется также реализовать интерфейс аннотаций, как показано в листинге 10.11.

Разумеется, сами аннотации ничего не делают. Они просто хранятся в исходном файле. Компилятор размещает их в файле классов, а виртуальная машина загружает их. Следовательно, требуется какой-то механизм для анализа аннотаций и установки приемников действий. Эта обязанность возлагается на класс `ActionListenerInstaller`. Конструктор класса `ButtonFrame` вызывает следующий метод:

```
ActionListenerInstaller.processAnnotations(this);
```

Статический метод `processAnnotations()` перечисляет все методы объекта, который он получает. Для каждого метода он извлекает объект аннотации типа `ActionListenerFor` и обрабатывает его, как показано ниже.

```
Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
  ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
  if (a != null) . . .
}
```

В данном случае применяется метод `getAnnotation()`, определенный в интерфейсе `AnnotatedElement`. Этот интерфейс реализуют такие классы, как `Method`, `Constructor`, `Field`, `Class` и `Package`. Имя исходного поля хранится в объекте аннотации. Оно извлекается в результате вызова метода `source()`, а затем осуществляется поиск соответствующего ему поля, как следует из приведенного ниже фрагмента кода.

```
String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);
```

Этот пример демонстрирует ограниченность рассматриваемой здесь аннотации. Исходный элемент должен непременно обозначать имя поля. А представлять локальную переменную он не может.

Остальная часть кода рассматриваемого здесь примера носит довольно технический характер. В частности, для каждого аннотируемого метода создается прокси-объект, класс которого реализует интерфейс `ActionListener` с методом

`actionPerformed()`, вызывающим данный аннотируемый метод. (Подробнее о прокси-объектах см. в главе 6 первого тома данной книги.) Не особенно вдаваясь в подробности, которые не так важны, отметим самое главное: функциональные возможности аннотаций устанавливаются с помощью метода `processAnnotations()`. Весь процесс обработки аннотаций схематически представлен на рис. 10.1.

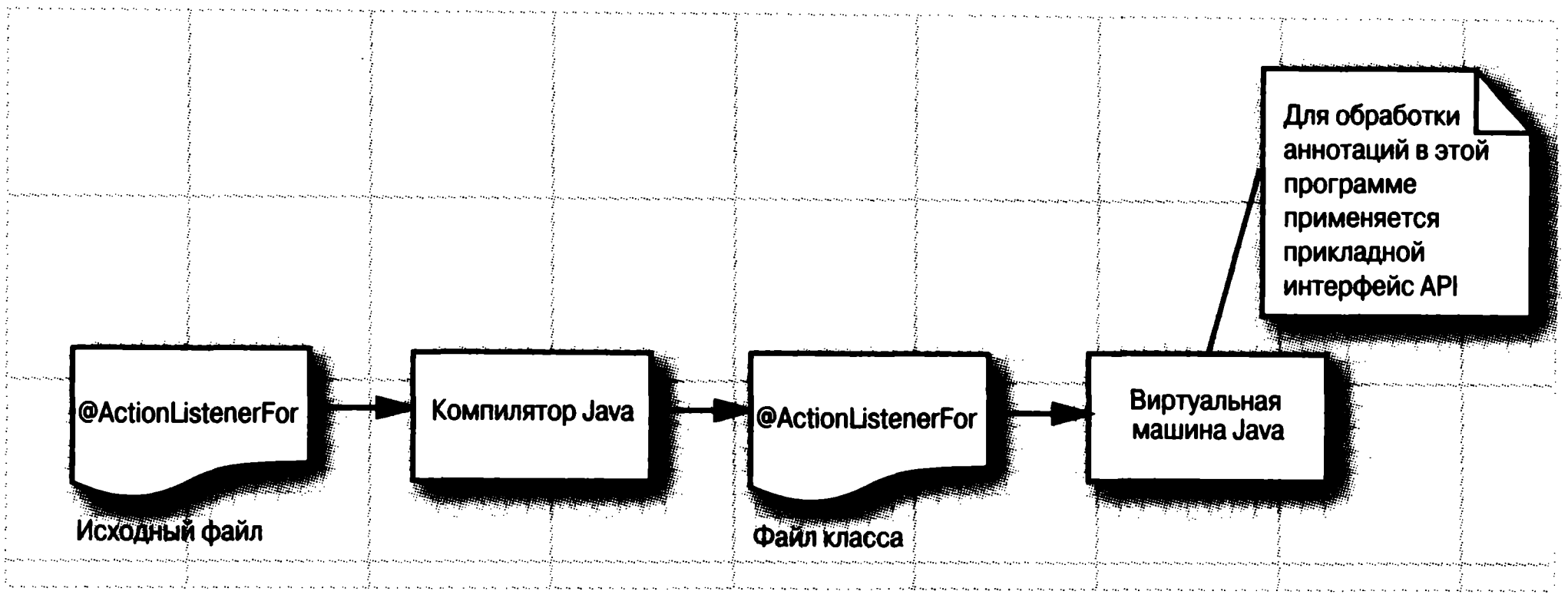


Рис. 10.1. Обработка аннотаций во время выполнения

В рассматриваемом здесь примере аннотации обрабатываются во время выполнения. Но их можно также обрабатывать и на уровне исходного кода. В частности, генератор исходного кода может сгенерировать код для ввода приемников событий. С другой стороны, аннотации можно также обрабатывать на уровне байт-кода. Для этого редактор байт-кода может вставлять вызовы метода `addActionListener()` в конструктор фрейма. На первый взгляд эта задача кажется слишком сложной, но в настоящее время доступны библиотеки, упрощающие ее решение. Удостовериться в этом у вас будет возможность на конкретном примере в разделе "Конструирование байт-кодов" далее в этой главе.

Пример, рассматриваемый в этом разделе, ни в коей мере нельзя рекомендовать в качестве серьезного средства для разработки пользовательских интерфейсов. Применение служебного метода для ввода приемников событий может оказаться не менее удобным, чем вставка аннотаций в исходный код. (На самом деле попытка реализовать именно такой подход предпринята в классе `java.beans.EventHandler`. Этот класс нетрудно сделать действительно полезным, предоставив метод, вводящий обработчик событий, вместо того чтобы создавать его.)

Но данный пример призван продемонстрировать внутренний механизм аннотирования программы и анализа аннотаций. Его рассмотрение должно помочь читателю как следует подготовиться к изучению материала последующих разделов, где приводится подробное описание синтаксиса аннотаций.

Листинг 10.9. Исходный код из файла `runtimeAnnotations/ActionListenerInstaller.java`

```
1  package runtimeAnnotations;
2
3  import java.awt.event.*;
4  import java.lang.reflect.*;
5
```

```
6  /**
7   * @version 1.00 2004-08-17
8   * @author Cay Horstmann
9  */
10 public class ActionListenerInstaller
11 {
12   /**
13    * Обрабатывает все аннотации типа ActionListenerFor в заданном объекте
14    * @param obj Объект, методы которого могут иметь аннотации
15    *                    типа ActionListenerFor
16   */
17   public static void processAnnotations(Object obj)
18   {
19     try
20     {
21       Class<?> cl = obj.getClass();
22       for (Method m : cl.getDeclaredMethods())
23       {
24         ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
25         if (a != null)
26         {
27           Field f = cl.getDeclaredField(a.source());
28           f.setAccessible(true);
29           addListener(f.get(obj), obj, m);
30         }
31       }
32     }
33     catch (ReflectiveOperationException e)
34     {
35       e.printStackTrace();
36     }
37   }
38
39   /**
40    * Вводит приемник действий, вызывающий заданный метод
41    * @param source Источник событий, в который вводится приемник действий
42    * @param param Неявный параметр метода, вызываемого приемником действий
43    * @param m Метод, вызываемый приемником действий
44   */
45   public static void addListener(Object source, final Object param,
46       final Method m) throws ReflectiveOperationException
47   {
48     InvocationHandler handler = new InvocationHandler()
49       {
50         public Object invoke(Object proxy, Method mm, Object[] args)
51             throws Throwable
52         {
53           return m.invoke(param);
54         }
55       };
56
57     Object listener = Proxy.newProxyInstance(null,
58     new Class[] { java.awt.event.ActionListener.class }, handler);
59     Method adder = source.getClass().getMethod("addActionListener",
60         ActionListener.class);
61     adder.invoke(source, listener);
62   }
63 }
```

Листинг 10.10. Исходный код из файла buttons3/ButtonFrame.java

```
1  package buttons3;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import runtimeAnnotations.*;
6
7  /**
8   * Фрейм с панелью кнопок
9   * @version 1.00 2004-08-17
10  * @author Cay Horstmann
11 */
12 public class ButtonFrame extends JFrame
13 {
14   private static final int DEFAULT_WIDTH = 300;
15   private static final int DEFAULT_HEIGHT = 200;
16
17   private JPanel panel;
18   private JButton yellowButton;
19   private JButton blueButton;
20   private JButton redButton;
21
22   public ButtonFrame()
23   {
24     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
25
26     panel = new JPanel();
27     add(panel);
28     yellowButton = new JButton("Yellow");
29     blueButton = new JButton("Blue");
30     redButton = new JButton("Red");
31
32     panel.add(yellowButton);
33     panel.add(blueButton);
34     panel.add(redButton);
35
36     ActionListenerInstaller.processAnnotations(this);
37   }
38
39   @ActionListenerFor(source = "yellowButton")
40   public void yellowBackground()
41   {
42     panel.setBackground(Color.YELLOW);
43   }
44
45   @ActionListenerFor(source = "blueButton")
46   public void blueBackground()
47   {
48     panel.setBackground(Color.BLUE);
49   }
50
51   @ActionListenerFor(source = "redButton")
52   public void redBackground()
53   {
54     panel.setBackground(Color.RED);
55   }
56 }
```

Листинг 10.11. Исходный код из файла runtimeAnnotations/ActionListenerFor.java

```
1  package runtimeAnnotations;
2
3  import java.lang.annotation.*;
4
5  /**
6   * @version 1.00 2004-08-17
7   * @author Cay Horstmann
8  */
9
10 @Target(ElementType.METHOD)
11 @Retention(RetentionPolicy.RUNTIME)
12 public @interface ActionListenerFor
13 {
14   String source();
15 }
```

java.lang.reflect.AnnotatedElement 5.0

- `boolean isAnnotationPresent(Class<? extends Annotation>` *annotationType*`)`

  Возвращает логическое значение `true`, если данный элемент имеет аннотацию указанного типа.

- `<T extends Annotation> T getAnnotation(Class<T>` *annotationType*`)`

  Получает аннотацию заданного типа или пустое значение `null`, если такая аннотация отсутствует у данного элемента.

- `Annotation[] getAnnotations()`

  Получает все аннотации, доступные для данного элемента, включая унаследованные. Если доступные аннотации отсутствуют, возвращает массив нулевой длины.

- `Annotation[] getDeclaredAnnotations()`

  Получает все аннотации, объявленные для данного элемента, исключая унаследованные. Если доступные аннотации отсутствуют, возвращает массив нулевой длины.

Синтаксис аннотаций

В этом разделе поясняется все, что требуется знать о синтаксисе аннотаций. Аннотация определяется с помощью интерфейса аннотаций следующим образом:

```
модификаторы @interface ИмяАннотации
{
  объявлениеЭлемента1
  объявлениеЭлемента2
  . . .
}
```

Каждый элемент может быть объявлен в одной из следующих форм:

```
тип имяЭлемента();
```

или

```
тип имяЭлемента() default значение;
```

Например, приведенная ниже аннотация состоит из двух элементов: `assignedTo` и `severity`.

```
public @interface BugReport
{
  String assignedTo() default "[none]";
  int severity() = 0;
}
```

Таким образом, каждая аннотация имеет следующий формат:

```
@ ИмяАннотации(имяЭлемента1=значение1, имяЭлемента2=значение2, . . .)
```

В качестве примера ниже приведено объявление аннотации.

```
@BugReport(assignedTo="Harry", severity=10)
```

Порядок следования элементов в аннотации особого значения не имеет. Так, следующая аннотация идентичная приведенной выше:

```
@BugReport(severity=10, assignedTo="Harry")
```

Значение, указываемое в объявлении аннотации по умолчанию, используется в том случае, если для элемента аннотации не задано конкретное значение. Ниже приведен пример аннотации, где в качестве значения для элемента `assignedTo` выбирается символьная строка `"[none]"`.

```
@BugReport(severity=10)
```

ВНИМАНИЕ! Устанавливаемые по умолчанию значения не хранятся вместе с аннотацией, а вычисляются динамически. Так, если заменить значение, устанавливаемое по умолчанию в элементе `assignedTo`, значением `"[]"` и скомпилировать интерфейс `BugReport` заново, в аннотации `@BugReport(severity=10)` будет использоваться новое устанавливаемое по умолчанию значение — даже в тех файлах классов, которые были скомпилированы до изменения этого значения.

Аннотации можно упростить с помощью двух специальных сокращений. Так, если элементы не указаны, потому что они просто отсутствуют в аннотации или все они принимают значения, устанавливаемые по умолчанию, то круглые скобки не требуются. Например, аннотация

```
@BugReport
```

равнозначна аннотации

```
@BugReport(assignedTo="[none]", severity=0)
```

Подобная аннотация называется *маркерной*.

Другим специальным сокращением является *однозначная аннотация*. Если элемент аннотации имеет специальное имя `value` и больше никаких элементов не указано, то имя элемента и знак = могут быть опущены. Так, если определить интерфейс аннотаций `ActionListenerFor`, упоминавшийся в предыдущем разделе, следующим образом:

```
public @interface ActionListenerFor
{
  String value();
}
```

то сами аннотации можно записать так:

```
@ActionListenerFor("yellowButton")
```

а не так, как показано ниже.

```
@ActionListenerFor(value="yellowButton")
```

Все интерфейсы аннотаций неявно расширяют интерфейс `java.lang.annotation.Annotation`. Это обычный интерфейс, а *не* интерфейс аннотаций. Предоставляемые им методы можно найти в конце этого раздела, где описывается прикладной интерфейс API. Самостоятельно расширять интерфейсы аннотаций нельзя. Иными словами, все интерфейсы аннотаций могут только расширять непосредственно интерфейс `java.lang.annotation.Annotation`.

Нельзя также предоставить классы, реализующие интерфейсы аннотаций. Вместо этого виртуальная машина сама генерирует подходящие прокси-классы и объекты по мере необходимости. Например, при запрашивании аннотации типа `ActionListenerFor` виртуальная машина выполнит операцию, аналогичную следующей:

```
return Proxy.newProxyInstance(classLoader, ActionListenerFor.class,
  new
  InvocationHandler()
  {
    public Object invoke(Object proxy, Method m, Object[] args)
       throws Throwable
    {
      if (m.getName().equals("source")) return значение аннотации source;
      . . .
    }
  })
```

Объявления элементов в интерфейсе аннотаций фактически являются объявлениями методов. Эти методы могут не иметь ни параметров, ни операторов `throws`, но не могут быть обобщенными.

Элемент аннотации может принимать один из следующих типов.

- Примитивный тип (`int`, `short`, `long`, `byte`, `char`, `double`, `float` или `boolean`).
- Строковый тип `String`.
- Тип `Class` (с каким-нибудь необязательным параметром вроде `Class<? extends MyClass>`).
- Перечислимый тип `enum`.
- Тип аннотации.
- Массив перечисленных выше типов (массив массивов не является допустимым типом элемента аннотации).

Ниже приведены примеры объявления элементов аннотации, имеющих допустимые типы.

```
public @interface BugReport
{
  enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
  boolean showStopper() default false;
  String assignedTo() default "[none]";
  Class<?> testCase() default Void.class;
  Status status() default Status.UNCONFIRMED;
  Reference ref() default @Reference(); // тип аннотации
  String[] reportedBy();
}
```

Аннотации вычисляются компилятором, поэтому значения всех элементов аннотаций должны быть непременно статическими константами. Ниже приведен характерный тому пример.

```
@BugReport(showStopper=true, assignedTo="Harry", testCase=MyTestCase.class,
    status=BugReport.Status.CONFIRMED, . . .)
```

ВНИМАНИЕ! Элемент аннотации вообще не может принимать пустое значение `null`. Даже по умолчанию не допускается устанавливать пустое значение `null`. Это не очень удобно, поскольку для установки по умолчанию придется выбирать другие значения наподобие `""` или `Void.class`.

Если в качестве значения элемента аннотации указывается массив, значения элементов такого массива заключаются в фигурные скобки:

```
@BugReport(. . ., reportedBy={"Harry", "Carl"})
```

Если же элемент аннотации принимает единственное значение, фигурные скобки можно опустить:

```
@BugReport(. . ., reportedBy="Joe") // допускается и подобно {"Joe"}
```

В качестве элемента аннотации может быть какая-нибудь другая аннотация. Это дает возможность создавать довольно сложные аннотации, как показано в приведенном ниже примере.

```
@BugReport(ref=@Reference(id="3352627"), . . .)
```

НА ЗАМЕТКУ! Внедрение циклических зависимостей в аннотациях считается ошибкой. Например, вследствие того, что аннотация `BugReport` содержит элемент типа `Reference`, аннотация `Reference` не может содержать элемент типа `BugReport`.

Вводить аннотации можно в следующие элементы кода.

- Пакеты.
- Классы (включая перечисления типа `enum`).
- Интерфейсы (включая интерфейсы аннотаций).
- Методы.
- Конструкторы.
- Поля экземпляров (включая перечислимые константы типа `enum`).
- Локальные переменные.
- Переменные параметров.

Но аннотации к локальным переменным могут обрабатываться только на уровне исходного кода. Файлы классов не описывают локальные переменные. Вследствие этого при компиляции класса аннотации всех локальных переменных отбрасываются. Аналогичным образом обстоит дело и с аннотациями к пакетам: они также не сохраняются вне уровня исходного кода.

НА ЗАМЕТКУ! Аннотирование пакетов осуществляется в файле `package-info.java`, где может содержаться только оператор пакета с предшествующими ему аннотациями.

У каждого элемента может быть несколько аннотаций, при условии, что они относятся к разным типам. Использовать один и тот же тип аннотации больше одного раза при аннотировании отдельного элемента нельзя. Например, компиляция следующего фрагмента кода приведет к ошибке:

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
void myMethod()
```

В качестве выхода из подобного затруднительного положения можно создать аннотацию со значением, состоящую из массива более простых аннотаций:

```
@BugReports({
  @BugReport(showStopper=true, reportedBy="Joe"),
  @BugReport(reportedBy={"Harry", "Carl"})})
void myMethod()
```

java.lang.annotation.Annotation 5.0

- `Class<? extends Annotation> annotationType()`

  Возвращает объект типа `Class`, который представляет интерфейс аннотаций данного объекта аннотации. Следует иметь в виду, что вызов метода `getClass()` для объекта аннотации привел бы к возврату конкретного класса, а не интерфейса.

- `boolean equals(Object` *other*`)`

  Возвращает логическое значение `true`, если параметр *other* обозначает объект, класс которого реализует тот же самый интерфейс аннотаций, что и данный объект аннотации, и если все элементы этого объекта равны параметру *other*.

- `int hashCode()`

  Возвращает хеш-код, совместимый с методом `equals()`, и получается из имени интерфейса аннотаций и значений его элементов.

- `String toString()`

  Возвращает строковое представление, содержащее название интерфейса аннотаций и значения элементов, например `@BugReport(assignedTo=[none], severity=0)`.

Стандартные аннотации

В Java SE имеется ряд интерфейсов аннотаций, определенных в таких пакетах, как `java.lang`, `java.lang.annotation` и `javax.annotation`. Четыре из них являются мета-аннотациями, описывающими поведение интерфейсов аннотаций, а остальные — обычными аннотациями, которые разработчики могут применять для аннотирования элементов в своем исходном коде. Все эти аннотации вкратце перечислены в табл. 10.2 и более подробно будут описываются в двух последующих разделах.

Таблица 10.2. Стандартные аннотации

Интерфейс аннотаций	Применение	Назначение
`Deprecated`	Все элементы кода	Помечает элемент кода как не рекомендуемый для применения
`SuppressWarnings`	Все элементы кода, кроме пакетов и аннотаций	Подавляет предупреждения указанного типа
`Override`	Методы	Проверяет, переопределяет ли данный метод соответствующий метод из суперкласса
`PostConstruct` `PreDestroy`	Методы	Помеченный метод должен вызываться сразу же после создания или непосредственно перед удалением
`Resource`	Классы, интерфейсы, методы, поля	Если это класс или интерфейс, помечает его как ресурс для использования в каком-нибудь другом месте. А если это метод или поле, помечает его как ресурс для "внесения"
`Resources`	Классы, интерфейсы	Помечает массив ресурсов
`Generated`	Все элементы кода	Помечает элемент как исходный код, сгенерированный каким-нибудь инструментальным средством
`Target`	Аннотации	Обозначает элементы, к которым может быть применена данная аннотация
`Retention`	Аннотации	Обозначает, как долго должна сохраняться данная аннотация
`Documented`	Аннотации	Обозначает, что данная аннотация должна быть включена в документацию на аннотируемые элементы кода
`Inherited`	Аннотации	Обозначает, что если данная аннотация применяется к классу, она будет автоматически наследоваться всеми его подклассами.

Аннотации для компиляции

Аннотация `@Deprecated` может присоединяться к любым элементам, применение которых впредь не рекомендуется. В таком случае компилятор будет выдавать соответствующее предупреждение, если нерекомендуемый элемент все-таки применяется в исходном коде. Эта аннотация имеет то же назначение, что и дескриптор `@deprecated` в Javadoc.

Аннотация `@SuppressWarnings` указывает компилятору подавлять предупреждения определенного типа, как показано в приведенном ниже примере.

```
@SuppressWarnings("unchecked")
```

Аннотация `@Override` применяется только к методам. Компилятор проверяет, чтобы метод с такой аннотацией действительно переопределял соответствующий метод из суперкласса. Так, если сделать приведенное ниже объявление, компилятор

выдаст ошибку, поскольку метод `equals()` не переопределяет аналогичный метод `equals()` из класса `Object`. Этот метод имеет параметр типа `Object`, а не `MyClass`.

```
public MyClass
{
  @Override public boolean equals(MyClass other);
  . . .
}
```

Аннотация `@Generated` предназначена для использования инструментальными средствами генерирования кода. Любой генерируемый исходный код может снабжаться аннотациями, чтобы отличаться от кода, предоставляемого программистом. Например, редактор кода может скрывать сгенерированный код, а генератор кода — удалять более старые версии сгенерированного кода. Каждая такая аннотация должна непременно содержать уникальный идентификатор для генератора кода. Символьные строки даты (в формате ISO 8601) и комментариев указывать необязательно. Ниже приведен характерный пример аннотации для генерирования кода.

```
@Generated("com.horstmann.beanproperty", "2008-01-04T12:08:56.235-0700");
```

Аннотации для управления ресурсами

Аннотации `@PostConstruct` и `@PreDestroy` применяются в таких средах, которые управляют жизненным циклом объектов, например, в веб-контейнерах и серверах приложений. Методы, помечаемые такими аннотациями, должны вызываться сразу же после создания объекта или непосредственно перед его удалением.

Аннотация `@Resource` предназначена для внедрения ресурсов. Рассмотрим в качестве примера веб-приложение, получающее доступ к базе данных. Разумеется, сведения о получении доступа к базе данных не должны жестко кодироваться в таком приложении. Вместо этого у веб-контейнера должен быть какой-то пользовательский интерфейс для установки параметров подключения, а также имя JNDI для источника данных. Сослаться на источник данных в веб-приложении можно следующим образом:

```
@Resource(name="jdbc/mydb")
private DataSource source;
```

В итоге при создании объекта, содержащего приведенное выше поле, контейнер будет автоматически "вносить" ссылку на источник данных.

Мета-аннотации

Мета-аннотация `@Target` применяется к аннотации, ограничивая те элементы, которые должны быть снабжены данной аннотацией, как показано в приведенном ниже примере.

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

В табл. 10.3 перечислены все возможные значения элементов данной мета-аннотации. Они относятся к перечислимому типу `ElementType`. Указывать можно любое количество типов элементов, заключая их в фигурные скобки.

Таблица 10.3. Типы элементов для мета-аннотации `@Target`

Тип элемента	Применение аннотации
`ANNOTATION_TYPE`	Объявления типов аннотаций
`PACKAGE`	Пакеты
`TYPE`	Классы (включая перечисления типа `enum`) и интерфейсы (включая типы аннотаций)
`METHOD`	Методы
`CONSTRUCTOR`	Конструкторы
`FIELD`	Поля (включая перечислимые константы типа `enum`)
`PARAMETER`	Параметры методов или конструкторов
`LOCAL_VARIABLE`	Локальные переменные

Аннотацией без ограничений, накладываемых мета-аннотацией `@Target`, можно снабдить *любой* элемент кода. Компилятор проверяет, чтобы аннотацией снабжался только разрешенный элемент кода. Так, если снабдить аннотацией `@BugReport` поле, компилятор выдаст во время компиляции соответствующую ошибку.

Мета-аннотация `@Retention` обозначает, насколько долго должна сохраняться аннотация. Указать можно не больше одного значения из тех, что приведены в табл. 10.4. По умолчанию устанавливается значение константы `RetentionPolicy.CLASS`.

Таблица 10.4. Правила сохраняемости для мета-аннотации `@Retention`

Правило сохраняемости	Описание
`SOURCE`	Аннотации не включаются в файлы классов
`CLASS`	Аннотации включаются в файлы классов, но виртуальной машине не нужно их загружать
`RUNTIME`	Аннотации включаются в файлы классов и загружаются виртуальной машиной. Они доступны через прикладной интерфейс API для рефлексии

В упоминавшемся ранее примере реализации интерфейса аннотаций из листинга 10.11 аннотация `@ActionListenerFor` объявлена со значением константы `RetentionPolicy.RUNTIME`, поскольку для обработки аннотаций в данном примере применялась рефлексия. В двух последующих разделах будут приведены примеры обработки аннотаций как на уровне исходного кода, так и на уровне файлов классов.

Мета-аннотация `@Documented` выдает подсказку о средствах документирования вроде Javadoc. Документируемые аннотации должны применяться в целях документирования таким же образом, как и другие модификаторы, подобные `protected` или `static`. О применении остальных аннотаций в документации не упоминается. Допустим, аннотация `@ActionListenerFor` объявляется как документируемая следующим образом:

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

В таком случае документация на каждый аннотированный метод будет содержать эту аннотацию, как показано на рис. 10.2. Если же аннотация оказывается временной (как, например, `@BugReport`), то документировать ее применение вряд ли стоит.

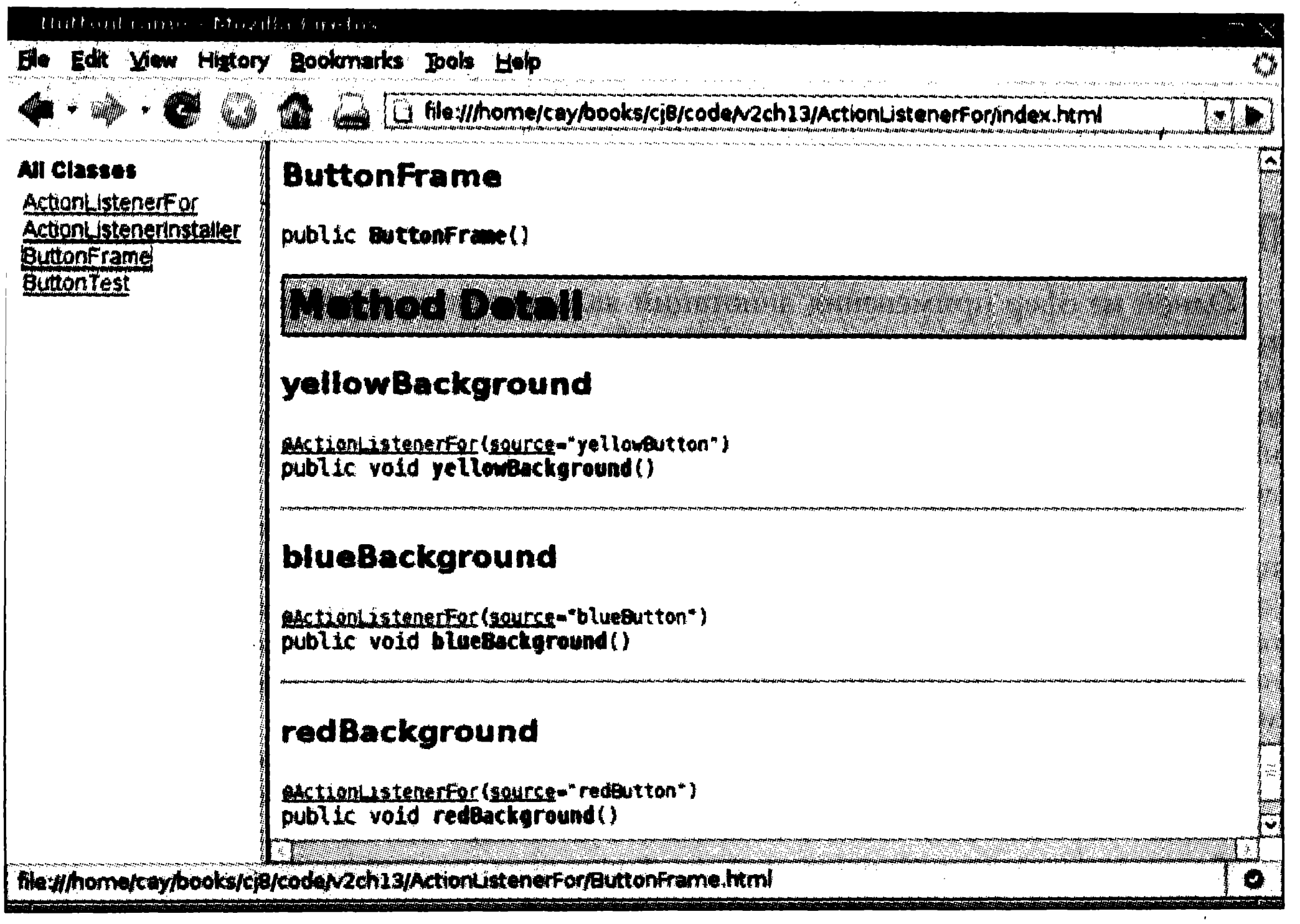


Рис. 10.2. Документируемые аннотации

НА ЗАМЕТКУ! Вполне допустимо применять аннотацию к самой себе. Например, аннотация `@Documented` аннотируется как `@Documented`. Поэтому в документации Javadoc на аннотации показано, являются ли они документируемыми.

Мета-аннотация `@Inherited` применяется только к аннотациям классов. Когда в класс вводится наследуемая аннотация, все его подклассы автоматически снабжаются точно такой же аннотацией. Это позволяет легко создавать аннотации, действующие таким же образом, как и маркерные интерфейсы вроде `Serializable`.

На самом деле аннотация `@Serializable` будет более уместной, чем маркерный интерфейс `Serializable` без единого метода. Класс является сериализируемым из-за наличия во время выполнения поддержки для чтения и записи его полей, а не из-за каких-то принципов объектно-ориентированного проектирования. Аннотация описывает данное обстоятельство намного лучше, чем наследование интерфейсов, ведь интерфейс `Serializable` появился еще в версии JDK 1.1, т.е. много раньше, чем аннотации.

Допустим, для указания на то, что объекты класса могут сохраняться в базе данных, определяется наследуемая аннотация `@Persistent`. В таком случае все

подклассы этого класса будут автоматически аннотироваться как сохраняемые следующим образом:

```
@Inherited @interface Persistent { }
@Persistent class Employee { . . . }
class Manager extends Employee { . . . } // также @Persistent
```

При поиске объектов, сохраняемых в базе данных, механизм сохраняемости будет обнаруживать как объекты типа `Employee`, так и объекты типа `Manager`.

Обработка аннотаций на уровне исходного кода

Одной из сфер применения аннотаций является автоматическое генерирование так называемых "сопроводительных файлов", содержащих дополнительные сведения о программах. Пользуясь ранними версиями Java Enterprise Edition, программисты вынуждены были иметь дело с большими объемами шаблонного кода. А в современных версиях Java EE доступны аннотации, позволяющие значительно упростить модель программирования.

В этом разделе упомянутый выше подход демонстрируется на примере простой программы, которая автоматически генерирует классы информации о компонентах JavaBeans. Свойства компонентов JavaBeans снабжаются аннотациями, после чего запускается инструментальное средство, выполняющее синтаксический анализ исходного файла и всех аннотаций, записывая результат в исходный файл класса информации о компоненте JavaBeans.

Как упоминалось в главе 8, класс информации о компоненте JavaBeans описывает этот компонент точнее, чем это можно сделать в ходе автоматического самоанализа. Он выдает перечень всех свойств компонента JavaBeans. Эти свойства могут обладать необязательными редакторами свойств. Типичным тому примером служит класс `ChartBeanBeanInfo`, рассматривавшийся в главе 8.

Для того чтобы избежать утомительной работы по написанию классов информации о компонентах JavaBeans, в рассматриваемом здесь примере вводится аннотация `@Property`. Такой аннотацией можно снабдить метод получения или установки свойства, как показано ниже.

```
@Property String getTitle() { return title; }
```

или

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

Определение аннотации `@Property` представлено в листинге 10.12. Следует заметить, что на эту аннотацию распространяется правило сохраняемости, которому соответствует значение константы `SOURCE`. Это означает, что аннотация может анализироваться только на уровне исходного кода. Она не включается в файлы классов и не доступна во время рефлексии.

Листинг 10.12. Исходный код из файла `sourceAnnotations/Property.java`

```
1  package sourceAnnotations;
2  import java.lang.annotation.*;
3
4  @Documented
```

```
5  @Target(ElementType.METHOD)
6  @Retention(RetentionPolicy.SOURCE)
7  public @interface Property
8  {
9    String editor() default "";
10 }
```

НА ЗАМЕТКУ! Имело бы смысл объявить элемент `editor` с типом `Class`. Но обработчик аннотаций не может извлекать аннотации типа `Class`, поскольку значение класса может зависеть от внешних факторов (например, от пути к классам или от загрузчиков классов). Поэтому для указания имени класса редактора служит символьная строка.

Далее для автоматического генерирования класса информации о компоненте JavaBeans типа *КлассJavaBeans* выполняются следующие действия.

1. Записывается исходный файл *КлассJavaBeans*`BeanInfo.java`. Объявляется класс *КлассJavaBeans*`BeanInfo` для расширения класса `SimpleBeanInfo` и переопределяется метод `getPropertyDescriptors()`.
2. Восстанавливается имя свойства для каждого аннотированного метода. С этой целью удаляется префикс **`get`** или **`set`**, а в оставшейся части имени свойства прописные буквы преобразуются в строчные.
3. Пишется оператор, создающий объект типа `PropertyDescriptor` для каждого свойства.
4. Если свойство снабжается редактором, то делается вызов метода `setPropertyEditorClass()`.
5. Пишется код для возврата массива дескрипторов всех свойств.

Например, следующая аннотация в классе `ChartBean`:

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

преобразуется в приведенный ниже код, шаблонная часть которого *не* выделена полужирным.

```
public class ChartBeanBeanInfo extends java.beans.SimpleBeanInfo
{
  public java.beans.PropertyDescriptor[] getProperties()
  {
    java.beans.PropertyDescriptor titlePositionDescriptor=
       new java.beans.PropertyDescriptor("titlePosition", ChartBean.class);
    titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class)
    . . .
    return new java.beans.PropertyDescriptor[]
    {
      titlePositionDescriptor,
      . . .
    }
  }
}
```

Все это нетрудно сделать, при условии, что удастся установить местонахождение всех методов, которые были помечены аннотацией `@Property`. Начиная с версии Java SE 6 *обработчики аннотаций* могут присоединяться к компилятору Java. (В версии Java SE 5 для этой цели применялась отдельная утилита `apt`.) Обработка аннотаций вызывается по следующей команде:

```
javac -processor ИмяКлассаОбработчика1, ИмяКлассаОбработчика2,
      . . . исходные_файлы
```

Компилятор обнаруживает сначала аннотации в исходных файлах, а затем выбирает те обработчики аннотаций, которые следует применить. Каждый обработчик аннотаций выполняется по очереди. Если обработчик аннотаций создает новый исходный файл, то процесс повторяется. Если цикл обработки больше не приводит к созданию исходных файлов, то выполняется компиляция всех исходных файлов. Порядок обработки аннотаций `@Property` схематически показан на рис. 10.3.

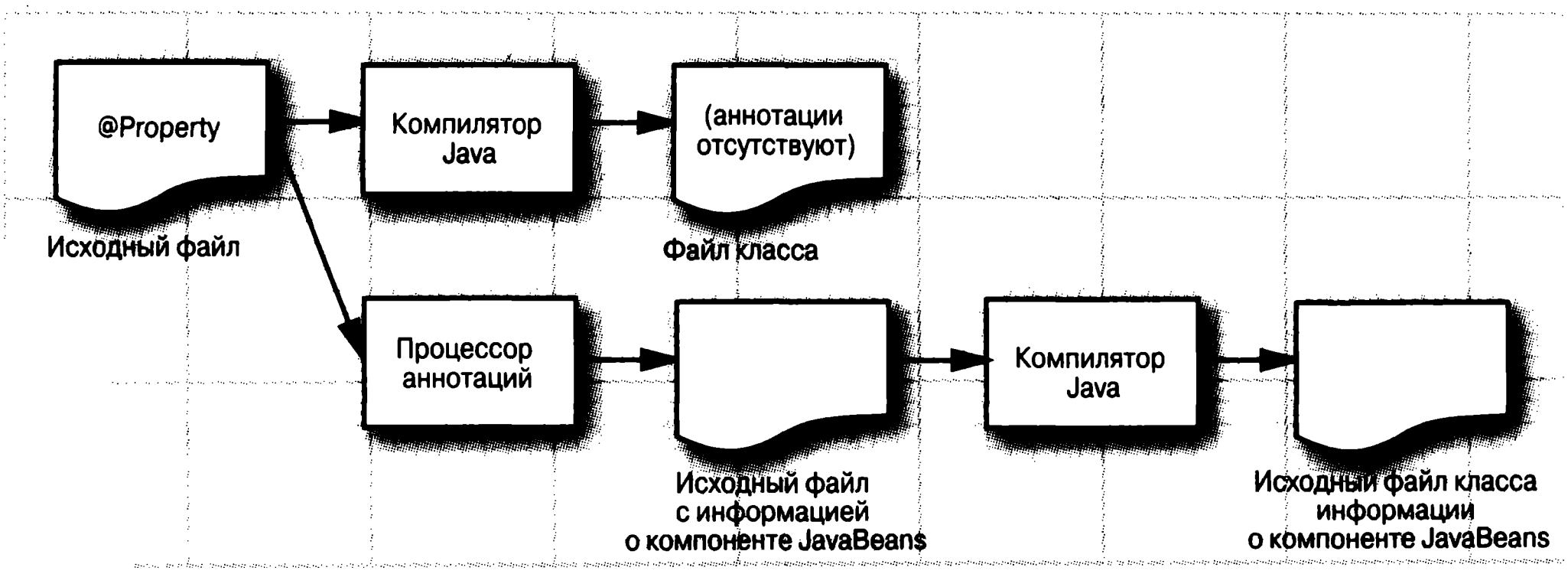


Рис. 10.3. Обработка аннотаций на уровне исходного кода

Здесь нет места для более подробного рассмотрения прикладного интерфейса API обработки аннотаций. Но пример программы из листинга 10.13 дает общее представление о его возможностях.

Обработчик аннотаций реализует интерфейс `Processor` — обычно путем расширения класса `AbstractProcessor`. При этом нужно указать те аннотации, которые поддерживает обработчик. И самим разработчикам прикладных интерфейсов API очень нравятся аннотации, поэтому они нередко пользуются аннотациями в следующих целях:

```
@SupportedAnnotationTypes("com.horstmann.annotations.Property")
public class BeanInfoAnnotationProcessor extends AbstractProcessor
```

Обработчик может запрашивать аннотации конкретных типов, интерпретировать подстановочные символы вроде `"com.horstmann.*"` (т.е. все аннотации из пакета `com.horstmann` или любого его подпакета) и даже метасимвол `"*"` (т.е. все аннотации).

У класса `BeanInfoAnnotationProcessor` имеется единственный открытый метод `process()`, который вызывается для каждого исходного файла. Этот метод принимает два параметра: ряд аннотаций, обрабатываемых в данном цикле, а также ссылку `RoundEnv` со сведениями о текущем цикле обработки.

В методе `process()` обходятся все аннóтированные методы. Для каждого из этих методов получается имя свойства. С этой целью отбрасывается префикс **`get`**, **`set`** или **`is`**, а следующая за ним прописная буква заменяется строчной. В целом структура этого кода выглядит следующим образом:

```
public boolean process(Set<? extends TypeElement> annotations,
    RoundEnvironment roundEnv)
{
  for (TypeElement t : annotations)
  {
    Map<String, Property> props = new LinkedHashMap<String, Property>();
    for (Element e : roundEnv.getElementsAnnotatedWith(t))
    {
      props.put(property name, e.getAnnotation(Property.class));
    }
  }
  write bean info source file
  return true;
}
```

Метод `process()` должен возвращать логическое значение `true`, если он предъявляет требования ко всем представленным ему аннотациям, т.е. если эти аннотации не должны передаваться дальше другим обработчикам аннотаций. Код для записи исходного файла выглядит довольно просто и представляет собой всего лишь последовательность операторов `out.print`. Обратите внимание на то, что средство записи выводимых данных создается следующим образом:

```
JavaFileObject sourceFile = processingEnv.getFiler().
createSourceFile(beanClassName
    + "BeanInfo");
PrintWriter out = new PrintWriter(sourceFile.openWriter());
```

Класс `AbstractProcessor` имеет защищенное поле `processingEnv` для получения доступа к различным службам обработки. Интерфейс `Filter` отвечает за создание новых файлов и их отслеживание, чтобы они могли обрабатываться в последующих циклах обработки.

Когда обработчик аннотаций обнаруживает ошибку, он использует средства обмена сообщениями для взаимодействия с пользователем. Так, если метод снабжен аннотацией `@Property`, но его имя не начинается с префикса **`get`**, **`set`** или **`is`**, выдается следующее сообщение об ошибке:

```
if (!found) processingEnv.getMessager().printMessage(Kind.ERROR,
    "@Property must be applied to getXxx, setXxx, or isXxx method", e);
```

В загружаемом коде к данной книге предлагается аннотированный файл `ChartBean.java`. Скомпилируйте сначала обработчик аннотаций по следующей команде:

```
javac sourceAnnotations/BeanInfoAnnotationProcessor.java
```

Затем выполните приведенную ниже команду и проанализируйте автоматически сгенерированный файл `ChartBeanBeanInfo.java`.

```
javac -processor sourceAnnotations.BeanInfoAnnotationProcessor
    chart/ChartBean.java
```

Чтобы посмотреть, каким образом происходит процесс обработки аннотаций, введите в команде `javac` параметр командной строки **`XprintRounds`**, чтобы получить следующий результат:

```
Round 1:
        input files: {com.horstmann.corejava.ChartBean}
        annotations: [com.horstmann.annotations.Property]
        last round: false
Round 2:
        input files: {com.horstmann.corejava.ChartBeanBeanInfo}
        annotations: []
        last round: false
Round 3:
        input files: {}
        annotations: []
        last round: true
```

Этот пример демонстрирует, каким образом инструментальные средства могут собирать аннотации из исходных файлов для генерирования других файлов. Генерируемые таким образом файлы необязательно должны быть исходными. Обработчики аннотаций могут отдавать предпочтение генерированию дескрипторов разметки в формате XML, файлов свойств, сценариев командной оболочки, документации в формате HTML и т.д.

НА ЗАМЕТКУ! Были предложения использовать аннотации, чтобы еще больше сократить объем рутинной работы. В самом деле, было бы замечательно, если бы тривиальные методы получения и установки генерировались автоматически. Например, следующая аннотация:

```
@Property private String title;
```

могла бы автоматически генерировать приведенные ниже методы.

```
public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }
```

Но эти методы должны быть введены *в один и тот же* класс. А для этого потребуется редактирование исходного файла, а не только генерирование еще одного файла, что выходит за пределы возможностей обработчиков аннотаций. С этой целью можно было бы создать другое инструментальное средство, но оно уже не вписывалось бы в рамки основного назначения аннотаций. Ведь аннотация предназначена для описания элемента кода, а *не* в качестве директивы для добавления или изменения кода.

Листинг 10.13. Исходный код из файла `sourceAnnotations/BeanInfoAnnotationProcessor.java`

```
1   package sourceAnnotations;
2
3   import java.beans.*;
```

```
4   import java.io.*;
5   import java.util.*;
6   import javax.annotation.processing.*;
7   import javax.lang.model.*;
8   import javax.lang.model.element.*;
9   import javax.tools.*;
10  import javax.tools.Diagnostic.*;
11
12  /**
13   * Этот класс реализует обработчик аннотаций, анализирующий
14   * аннотации к свойствам @Property
15   * @version 1.11 2012-01-26
16   * @author Cay Horstmann
17  */
18  @SupportedAnnotationTypes("sourceAnnotations.Property")
19  @SupportedSourceVersion(SourceVersion.RELEASE_7)
20  public class BeanInfoAnnotationProcessor extends AbstractProcessor
21  {
22    @Override
23    public boolean process(Set<? extends TypeElement> annotations,
24       RoundEnvironment roundEnv)
25    {
26      for (TypeElement t : annotations)
27      {
28        Map<String, Property> props = new LinkedHashMap<>();
29        String beanClassName = null;
30        for (Element e : roundEnv.getElementsAnnotatedWith(t))
31        {
32          String mname = e.getSimpleName().toString();
33          String[] prefixes = { "get", "set", "is" };
34          boolean found = false;
35          for (int i = 0; !found && i < prefixes.length; i++)
36            if (mname.startsWith(prefixes[i]))
37            {
38              found = true;
39              int start = prefixes[i].length();
40              String name = Introspector.decapitalize(
41                 mname.substring(start));
42              props.put(name, e.getAnnotation(Property.class));
43            }
44
45          if (!found) processingEnv.getMessager().printMessage(
46             Kind.ERROR, "@Property must be applied to getXxx,
47             setXxx, or isXxx method", e);
48          else if (beanClassName == null)
49             beanClassName = ((TypeElement) e.getEnclosingElement())
50             .getQualifiedName().toString();
51        }
52        try
53        {
54          if (beanClassName != null) writeBeanInfoFile(
55             beanClassName, props);
56        }
57        catch (IOException e)
58        {
59          e.printStackTrace();
```

```
60          }
61        }
62        return true;
63      }
64
65      /**
66       * Записывает исходный файл класса информации о компоненте JavaBeans
67       * @param beanClassName Имя класса информации о компоненте JavaBeans
68       * @param props Отображение имен свойств и их аннотаций
69      */
70      private void writeBeanInfoFile(String beanClassName, Map<String,
71         Property> props) throws IOException
72      {
73        JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(
74           beanClassName + "BeanInfo");
75        PrintWriter out = new PrintWriter(sourceFile.openWriter());
76        int i = beanClassName.lastIndexOf(".");
77        if (i > 0)
78        {
79          out.print("package ");
80          out.print(beanClassName.substring(0, i));
81          out.println(";");
82        }
83        out.print("public class ");
84        out.print(beanClassName.substring(i + 1));
85        out.println("BeanInfo extends java.beans.SimpleBeanInfo");
86        out.println("{");
87        out.println(" public java.beans.PropertyDescriptor[]
88           getPropertyDescriptors()");
89        out.println(" {");
90        out.println(" try");
91        out.println(" {");
92        for (Map.Entry<String, Property> e : props.entrySet())
93        {
94          out.print(" java.beans.PropertyDescriptor ");
95          out.print(e.getKey());
96          out.println("Descriptor");
97          out.print(" = new java.beans.PropertyDescriptor(\"");
98          out.print(e.getKey());
99          out.print("\", ");
100         out.print(beanClassName);
101         out.println(".class);");
102         String ed = e.getValue().editor().toString();
103         if (!ed.equals(""))
104         {
105           out.print(" ");
106           out.print(e.getKey());
107           out.print("Descriptor.setPropertyEditorClass(");
108           out.print(ed);
109           out.println(".class);");
110         }
111       }
112       out.println(" return new java.beans.PropertyDescriptor[]");
113       out.print(" {");
114       boolean first = true;
115       for (String p : props.keySet())
```

```
116     {
117       if (first) first = false;
118       else out.print(",");
119       out.println();
120       out.print(" ");
121       out.print(p);
122       out.print("Descriptor");
123     }
124     out.println();
125     out.println(" };");
126     out.println(" }");
127     out.println(" catch (java.beans.IntrospectionException e)");
128     out.println(" {");
129     out.println(" e.printStackTrace();");
130     out.println(" return null;");
131     out.println(" }");
132     out.println(" }");
133     out.println("}");
134     out.close();
135   }
136 }
```

Конструирование байт-кодов

Как пояснялось ранее, аннотации могут обрабатываться во время выполнения и на уровне исходного кода. Но существует еще и третий способ обработки аннотаций на уровне байт-кода. Если аннотации не удаляются на уровне исходного кода, они присутствуют в файлах классов. Формат этих файлов документирован (см. `http://docs.oracle.com/javase/specs/jvms/se7/html`). Это довольно сложный формат, и поэтому обрабатывать файлы классов без специальных библиотек было бы совсем не просто. К их числу относится библиотека BCEL (Bytecode Engineering Library — Библиотека конструирования байт-кодов), доступная по адресу `http://jakarta.apache.org/bcel`.

В этом разделе будет показано, как пользоваться этой библиотекой для добавления в аннотированные методы протокольных сообщений. Так, если метод снабжен следующей аннотацией:

```
@LogEntry(logger=ИмяРегистратора)
```

то в начале этого метода вводятся байт-коды для приведенного ниже оператора.

```
Logger.getLogger(ИмяРегистратора).entering(ИмяКласса, ИмяМетода);
```

Если, например, снабдить такой аннотацией метод `hashCode()` из класса `Item` следующим образом:

```
@LogEntry(logger="global") public int hashCode()
```

то при каждом вызове этого метода будет выводиться приблизительно такое сообщение:

```
Aug 17, 2004 9:32:59 PM Item hashCode
FINER: ENTRY
```

Для того чтобы добиться такого результата, нужно выполнить следующие действия.

1. Загрузить байт-коды в файл классов.
2. Определить местонахождение всех методов.
3. Выполнить для каждого метода проверку на присутствие в нем аннотации `LogEntry`.
4. Если такая аннотация присутствует, ввести байт-коды для следующих инструкций в начале метода:

```
ldc ИмяРегистратора
invokestatic java/util/logging/Logger.getLogger:(Ljava/lang/String;)
   Ljava/util/logging/Logger;
ldc ИмяКласса
ldc ИмяМетода
invokevirtual java/util/logging/Logger.entering:
   (Ljava/lang/String;Ljava/lang/String;)V
```

Вставка этих байт-кодов может показаться, на первый взгляд, сложной задачей, но библиотека BCEL существенно упрощает ее. Не вдаваясь в подробности анализа и вставки байт-кодов, обратимся к конкретному примеру программы из листинге 10.14. В этой программе редактируется файл классов и вставляется вызов регистратора в начале всех методов, снабженных аннотацией `LogEntry`.

НА ЗАМЕТКУ! Тем, кого интересуют подробности процесса конструирования байт-кодов, рекомендуется ознакомиться с руководством по библиотеке BCEL, доступным по адресу `http://jakarta.apache.org/bcel/manual.html`.

Для компиляции и запуска рассматриваемой здесь программы `EntryLogger` требуется версия библиотеки BCEL 6.0 или более поздняя. (На момент написания этой главы версия 6.0 все еще находилась на стадии разработки. Если она до сих пор не готова, следует использовать промежуточную версию, например 5.2.) Ниже показано, каким образом инструкции для регистрации вводятся в исходный файл `Item.java`, представленный в листинге 10.15.

```
javac set/Item.java
javac -classpath .:bcel-версия.jar bytecodeAnnotations/EntryLogger.java
java -classpath .:bcel-версия.jar bytecodeAnnotations.EntryLogger set.Item
```

Попробуйте выполнить следующую команду до и после изменения файла класса `Item`:

```
javap -c set.Item
```

В итоге инструкции для регистрации должны быть вставлены в начале методов `hashCode()`, `equals()` и `compareTo()`, как показано ниже.

```
public int hashCode();
  Code:
    0: ldc    #85; //String global
    2: invokestatic #80;
      //Method java/util/logging/Logger.getLogger:
        (Ljava/lang/String;)Ljava/util/logging/Logger;
    5: ldc    #86; //String Item
    7: ldc    #88; //String hashCode
    9: invokevirtual    #84;
```

```
  //Method java/util/logging/Logger.entering:
    (Ljava/lang/String;Ljava/lang/String;)V
12: bipush 13
14: aload_0
15: getfield        #2; //Field description:Ljava/lang/String;
18: invokevirtual   #15; //Method java/lang/String.hashCode:()I
21: imul
22: bipush 17
24: aload_0
25: getfield        #3; //Field partNumber:I
28: imul
29: iadd
30: ireturn
```

Программа `SetTest` из листинга 10.16 вставляет объекты типа `Item` в хеш-множество. Если вы запустите ее с измененным файлом классов, то увидите протокольные сообщения, аналогичные приведенным ниже.

```
Aug 18, 2004 10:57:59 AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item equals
FINER: ENTRY
[[description=Toaster, partNumber=1729], [description=Microwave,
    partNumber=4104]]
```

Обратите внимание на вызов метода `equals()` при вставке одного и того же элемента дважды. Данный пример демонстрирует эффективность конструирования байт-кодов. Аннотации используются для ввода в прикладную программу директив, а инструментальное средство редактирования байт-кодов собирает эти директивы и соответственно видоизменяет инструкции виртуальной машины.

Листинг 10.14. Исходный код из файла `bytecodeAnnotations/EntryLogger.java`

```
1  package bytecodeAnnotations;
2
3  import java.io.*;
4  import org.apache.bcel.*;
5  import org.apache.bcel.classfile.*;
6  import org.apache.bcel.generic.*;
7
8  /**
9   * Вводит инструкции для регистрации записей в начале всех
10  * методов из класса, снабженного аннотацией LogEntry
11  * @version 1.10 2007-10-27
12  * @author Cay Horstmann
13 */
14 public class EntryLogger
15 {
16   private ClassGen cg;
17   private ConstantPoolGen cpg;
18   /**
19    * Вводит код регистрации записей в указанный класс
```

```
20      * @param args Имя файла класса для вставки кода
21     */
22     public static void main(String[] args)
23     {
24       try
25       {
26         if (args.length == 0)
27           System.out.println("USAGE:
28              java bytecodeAnnotations.EntryLogger classname");
29         else
30         {
31           JavaClass jc = Repository.lookupClass(args[0]);
32           ClassGen cg = new ClassGen(jc);
33           EntryLogger el = new EntryLogger(cg);
34           el.convert();
35           String f = Repository.lookupClassFile(cg.getClassName()).getPath();
36           System.out.println("Dumping " + f);
37           cg.getJavaClass().dump(f);
38         }
39       }
40       catch (Exception e)
41       {
42         e.printStackTrace();
43       }
44     }
45
46     /**
47      * Конструирует объект типа EntryLogger, вставляющий инструкции
48      * для регистрации в аннотируемые методы указанного класса
49      * @param cg Имя класса
50     */
51     public EntryLogger(ClassGen cg)
52     {
53       this.cg = cg;
54       cpg = cg.getConstantPool();
55     }
56
57     /**
58      * Преобразует класс, вставляя в него вызовы регистратора
59     */
60     public void convert() throws IOException
61     {
62       for (Method m : cg.getMethods())
63       {
64         AnnotationEntry[] annotations = m.getAnnotationEntries();
65         for (AnnotationEntry a : annotations)
66         {
67           if (a.getAnnotationType().equals("LbytecodeAnnotations/LogEntry;"))
68           {
69             for (ElementValuePair p : a.getElementValuePairs())
70             {
71               if (p.getNameString().equals("logger"))
72               {
73                 String loggerName = p.getValue().stringifyValue();
74                 cg.replaceMethod(m, insertLogEntry(m, loggerName));
75               }
76             }
```

```
77            }
78         }
79      }
80   }
81
82   /**
83    * Вводит вызов регистратора в начале метода
84    * @param m Имя метода
85    * @param loggerName Имя вызываемого регистратора
86   */
87   private Method insertLogEntry(Method m, String loggerName)
88   {
89     MethodGen mg = new MethodGen(m, cg.getClassName(), cpg);
90     String className = cg.getClassName();
91     String methodName = mg.getMethod().getName();
92     System.out.printf("Adding logging instructions to %s.%s%n",
93        className, methodName);
94
95     int getLoggerIndex = cpg.addMethodref("java.util.logging.Logger",
96        "getLogger", "(Ljava/lang/String;)Ljava/util/logging/Logger;");
97     int enteringIndex = cpg.addMethodref("java.util.logging.Logger",
98        "entering", "(Ljava/lang/String;Ljava/lang/String;)V");
99
100    InstructionList il = mg.getInstructionList();
101    InstructionList patch = new InstructionList();
102    patch.append(new PUSH(cpg, loggerName));
103    patch.append(new INVOKESTATIC(getLoggerIndex));
104    patch.append(new PUSH(cpg, className));
105    patch.append(new PUSH(cpg, methodName));
106    patch.append(new INVOKEVIRTUAL(enteringIndex));
107    InstructionHandle[] ihs = il.getInstructionHandles();
108    il.insert(ihs[0], patch);
109
110    mg.setMaxStack();
111    return mg.getMethod();
112  }
113 }
```

Листинг 10.15. Исходный код из файла `set/Item.java`

```
1  package set;
2
3  import java.util.*;
4  import bytecodeAnnotations.*;
5
6  /**
7   * Элемент с описанием и номенклатурным номером
8   * @version 1.01 2012-01-26
9   * @author Cay Horstmann
10 */
11 public class Item
12 {
13   private String description;
14   private int partNumber;
15
16   /**
17    * Конструирует элемент
```

```
18     * @param aDescription Описание элемента
19     * @param aPartNumber Номенклатурный номер
20    */
21   public Item(String aDescription, int aPartNumber)
22   {
23     description = aDescription;
24     partNumber = aPartNumber;
25   }
26
27   /**
28    * Получить описание данного элемента
29    * @return Возвращает описание
30   */
31   public String getDescription()
32   {
33     return description;
34   }
35
36   public String toString()
37   {
38     return "[description=" + description + ", partNumber="
39        + partNumber + "]";
40   }
41
42   @LogEntry(logger = "global")
43   public boolean equals(Object otherObject)
44   {
45     if (this == otherObject) return true;
46     if (otherObject == null) return false;
47     if (getClass() != otherObject.getClass()) return false;
48     Item other = (Item) otherObject;
49     return Objects.equals(description, other.description)
50        && partNumber == other.partNumber;
51   }
52
53   @LogEntry(logger = "global")
54   public int hashCode()
55   {
56     return Objects.hash(description, partNumber);
57   }
58 }
```

Листинг 10.16. Исходный код из файла `set/SetTest.java`

```
1  package set;
2
3  import java.util.*;
4  import java.util.logging.*;
5
6  /**
7   * @version 1.02 2012-01-26
8   * @author Cay Horstmann
9  */
10 public class SetTest
11 {
12   public static void main(String[] args)
13   {
```

```
14      Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).setLevel(Level.FINEST);
15      Handler handler = new ConsoleHandler();
16      handler.setLevel(Level.FINEST);
17      Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).addHandler(handler);
18
19      Set<Item> parts = new HashSet<>();
20      parts.add(new Item("Toaster", 1279));
21      parts.add(new Item("Microwave", 4104));
22      parts.add(new Item("Toaster", 1279));
23      System.out.println(parts);
24    }
25 }
```

Видоизменение байт-кодов во время загрузки

В предыдущем разделе было представлено инструментальное средство, способное редактировать файлы классов. Но добавление еще одного такого инструментального средства во время компоновки прикладной программы может оказаться непростым делом. Поэтому существует довольно привлекательный альтернативный способ, который заключается в том, чтобы отложить конструирование байт-кодов до стадии *загрузки*, на которой загружаются классы.

До появления версии Java SE 5.0 для реализации такого способа приходилось разрабатывать специальный загрузчик классов. А ныне в прикладном интерфейсе API оснащения инструментальными средствами имеется перехватчик, позволяющий устанавливать преобразователь байт-кодов. Этот преобразователь должен устанавливаться перед вызовом главного метода программы. Для того чтобы удовлетворить данному требованию, определяется *агент* (т.е. библиотека, загружаемая для контроля над программой). Код этого агента может выполнять в методе `premain()` операции, требующиеся для инициализации.

Для того чтобы создать агент, необходимо выполнить следующие действия.

1. Реализовать класс с помощью метода

   ```
   public static void premain(String arg, Instrumentation instr)
   ```

2. Этот метод вызывается при загрузке агента. Агент может получить единственный аргумент командной строки, передаваемый в качестве параметра **`arg`**. А параметр **`instr`** можно использовать для установки различных перехватчиков.

3. Создать файл манифеста, в котором устанавливается атрибут `Premain-Class`, например, следующим образом:

   ```
   Premain-Class: bytecodeAnnotations.EntryLoggingAgent
   ```

4. Упаковать код агента и манифест в JAR-файл, например, так, как показано ниже.

   ```
   javac -classpath .:bcel-версия.jar bytecodeAnnotations.EntryLoggingAgent
   jar cvfm EntryLoggingAgent.jar EntryLoggingAgent.mf
       bytecodeAnnotations/Entry*.class
   ```

Чтобы запустить программу на Java вместе с агентом, нужно указать следующие параметры в командной строке:

```
java -javaagent:JAR-файл_с_агентом=АргументАгента . . .
```

Например, чтобы запустить программу `SetTest` с агентом регистрации записей, потребуется выполнить приведенные ниже команды, где аргумент `Item` обозначает имя класса, который агент должен видоизменить.

```
javac SetTest.java
java -javaagent:EntryLoggingAgent.jar=set.Item -classpath .:
    bcel-версия.jar set.SetTest
```

В листинге 10.17 представлен исходный код агента, устанавливающего преобразователь файлов классов. Этот преобразователь сначала проверяет, соответствует ли имя класса аргументу агента. Если соответствует, то он использует класс `EntryLogger`, упоминавшийся в предыдущем разделе, для видоизменения байт-кодов. Но видоизмененные байт-коды не сохраняются в файле. Вместо этого преобразователь возвращает их для загрузки в виртуальную машину (рис. 10.4). Иными словами, данный способ позволяет видоизменять байт-коды на лету, т.е. динамически.

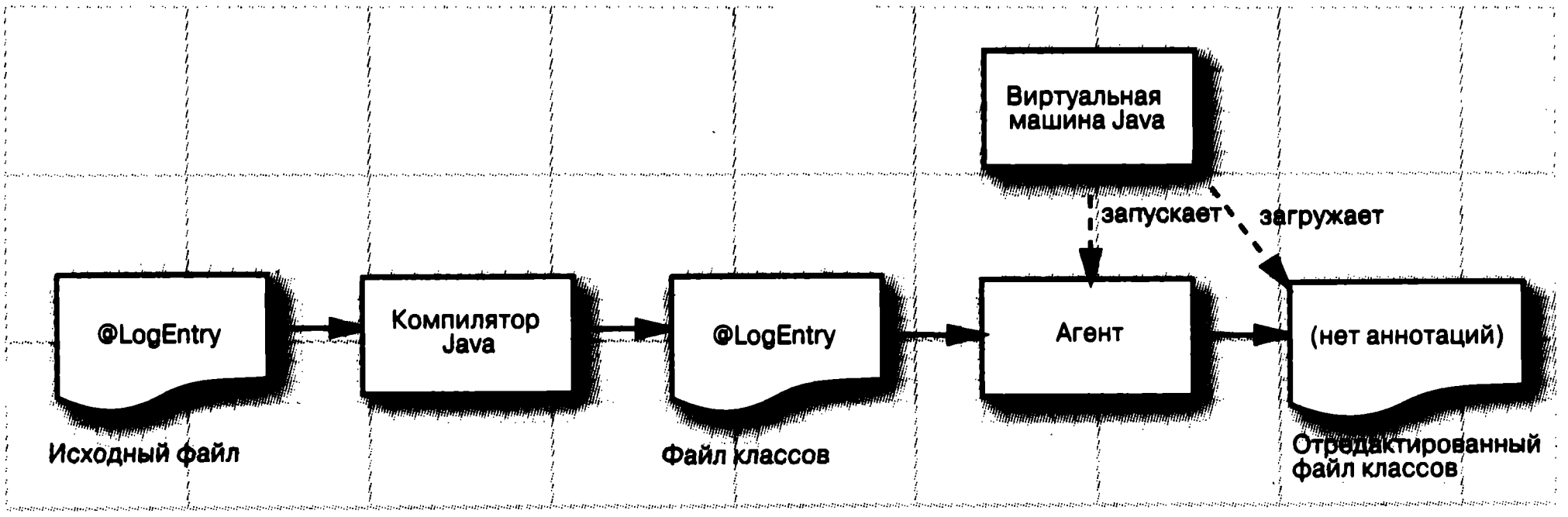


Рис. 10.4. Видоизменение классов во время загрузки

Листинг 10.17. Исходный код из файла `bytecodeAnnotations/EntryLoggingAgent.java`

```
1  package bytecodeAnnotations;
2
3  import java.lang.instrument.*;
4  import java.io.*;
5  import java.security.*;
6  import org.apache.bcel.classfile.*;
7  import org.apache.bcel.generic.*;
8
9  /**
10  * @version 1.00 2004-08-17
11  * @author Cay Horstmann
12 */
13 public class EntryLoggingAgent
14 {
15   public static void premain(final String arg, Instrumentation instr)
16   {
17     instr.addTransformer(new ClassFileTransformer()
18     {
19       public byte[] transform(ClassLoader loader, String className,
20          Class<?> cl, ProtectionDomain pd, byte[] data)
21       {
22         if (!className.equals(arg)) return null;
```

```
23          try
24          {
25            ClassParser parser = new ClassParser(
26               new ByteArrayInputStream(data), className + ".java");
27            JavaClass jc = parser.parse();
28            ClassGen cg = new ClassGen(jc);
29            EntryLogger el = new EntryLogger(cg);
30            el.convert();
31            return cg.getJavaClass().getBytes();
32          }
33          catch (Exception e)
34          {
35            e.printStackTrace();
36            return null;
37          }
38        }
39      });
40    }
41  }
```

Из этой главы вы узнали, как:

- вводить аннотации в исходный код программ на Java;
- создавать свои собственные интерфейсы аннотаций;
- реализовывать инструментальные средства, применяющие аннотации.

В этой главе были также продемонстрированы три методики обработки кода: написание сценариев, компиляция программ на Java и обработка аннотаций. Первые две методики довольно просты. Что же касается создания инструментальных средств обработки аннотаций, то эта методика, безусловно, сложна, и поэтому ею вряд ли рискнут воспользоваться многие разработчики. Следует иметь в виду, что в этой главе были представлены лишь самые основные сведения, необходимые для правильного понимания внутреннего механизма работы типичных инструментальных средств обработки аннотаций. Но, возможно, они вызовут у вас интерес к созданию своих собственных инструментальных средств для этих целей.

В следующей главе будут рассмотрены механизм вызова удаленных методов и модель распределенных объектов для программ на Java.

ГЛАВА 11

Распределенные объекты

В этой главе...

- Роли клиента и сервера
- Вызовы удаленных методов
- Модель программирования RMI
- Параметры и возвращаемые значения в удаленных методах
- Активизация удаленных объектов

Время от времени программисты возвращаются к идее о "повсеместном использовании объектов" как к решению всех проблем. Эта идея состоит в том, чтобы создать набор эффективно взаимодействующих распределенных объектов, которые можно было бы размещать где угодно. В таком случае, если бы объекту на одном компьютере потребовалось вызывать метод для объекта на другом компьютере, он мог бы отправить по сети сообщение с подробным запросом. А удаленный объект сформировал бы ответ, получив, например, доступ к базе данных или совершив обмен данными с другими объектами. Подготовив ответ на запрос клиента, он отослал бы его обратно по сети. С теоретической точки зрения этот процесс выглядит очень просто, но для того, чтобы эффективно использовать распределенные объекты, нужно ясно понимать, что же происходит внутри данного процесса.

В этой главе речь пойдет о протоколе RMI (Remote Method Invocation — вызов удаленных методов), обеспечивающем взаимодействие двух виртуальных машин Java, которые могут находиться на разных компьютерах. Некогда протокол RMI считался надежной технологией для прикладного программирования, но теперь он представляет интерес лишь как удобный пример для изучения простой системы распределенных объектов.

Роли клиента и сервера

Главный принцип, положенный в основу всех технологий распределенного программирования, довольно прост. Клиентский компьютер формирует запрос и отправляет его по сети на серверный компьютер. Сервер обрабатывает запрос и отправляет ответ обратно клиенту для дальнейшего анализа. Весь этот процесс схематически показан на рис. 11.1.

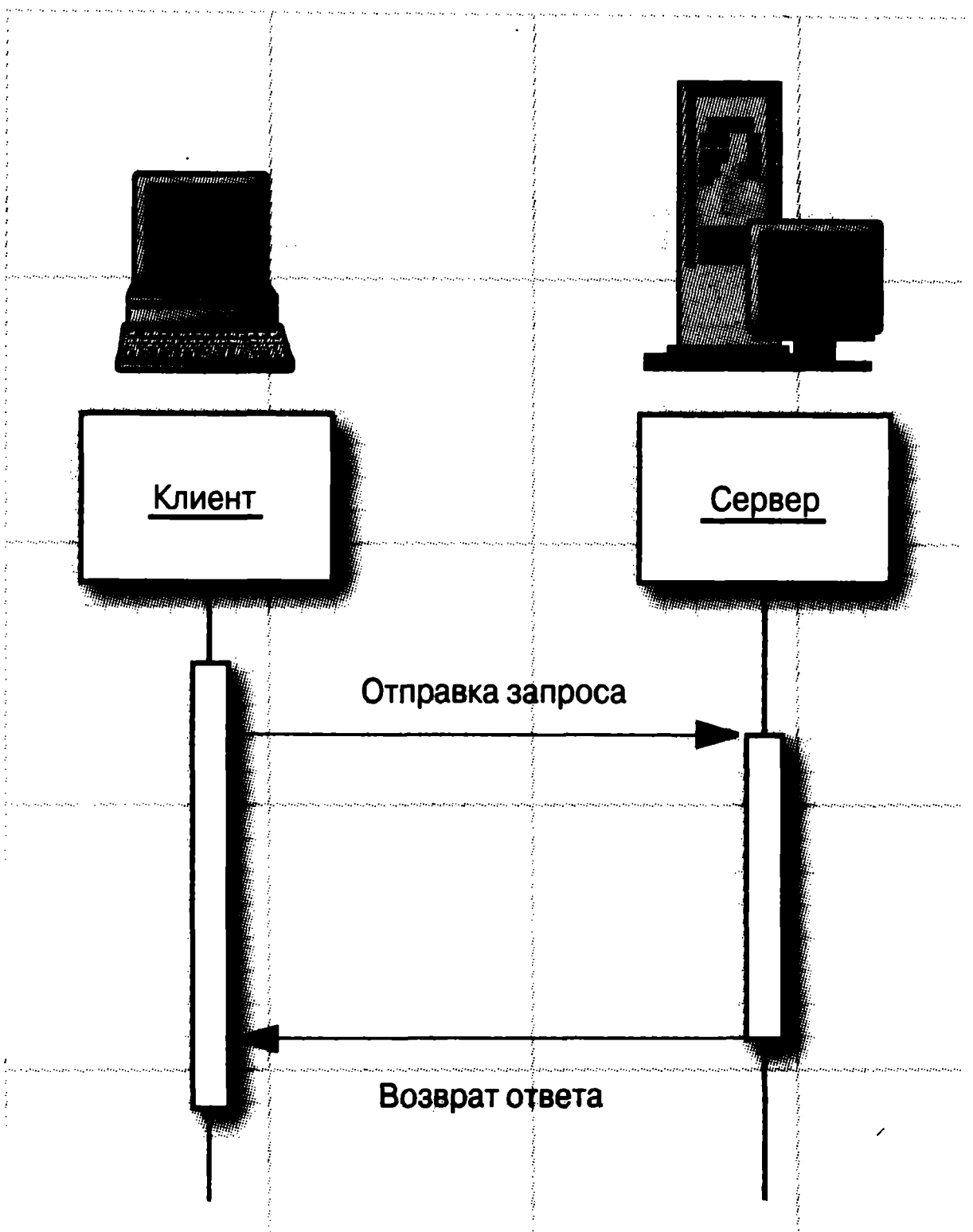


Рис. 11.1. Обмен объектами между клиентом и сервером

Прежде всего следует заметить, что подобные запросы и ответы не похожи на те, что обычно используются в веб-приложениях. В роли клиента в данном случае выступает не веб-браузер, а любое приложение, реализующее бизнес-правила какой угодно сложности. Клиентское приложение может как предусматривать, так и не предусматривать взаимодействие с пользователем. И если такое взаимодействие предусматривается, то клиентское приложение может иметь как интерфейс командной строки, так и графический интерфейс на основе компонентов Swing. Протокол, применяемый в этом приложении для отправки запросов и получения ответов, должен поддерживать пересылку произвольных объектов, тогда как в традиционных веб-приложениях для отправки запросов может применяться только протокол HTTP, а для получения ответов — только формат HTML.

Таким образом, разработчикам клиентских приложений требуется механизм, с помощью которого они могли бы делать обычные вызовы методов, не особенно беспокоясь ни о передаче данных по сети, ни о синтаксическом анализе получаемых в ответ данных. Оптимальным для них решением является установка на клиентском компьютере так называемого *прокси-объекта*. Прокси-объект — это такой объект, который размещается на виртуальной машине клиента и воспринимается клиентским приложением как удаленный объект. Клиент может обращаться к прокси-объекту, вызывая метод обычным образом, а сам прокси-объект — связываться с сервером по сетевому протоколу.

Аналогично разработчикам серверных служб необходим механизм, который бы позволял им не особенно беспокоиться об обмене данными с клиентами. Для них оптимальным решением является установка на сервере второго прокси-объекта. Серверный прокси-объект может взаимодействовать с клиентским прокси-объектом и вызывать методы обычным образом для обмена данными с объектом, реализующим серверную службу (рис. 11.2).

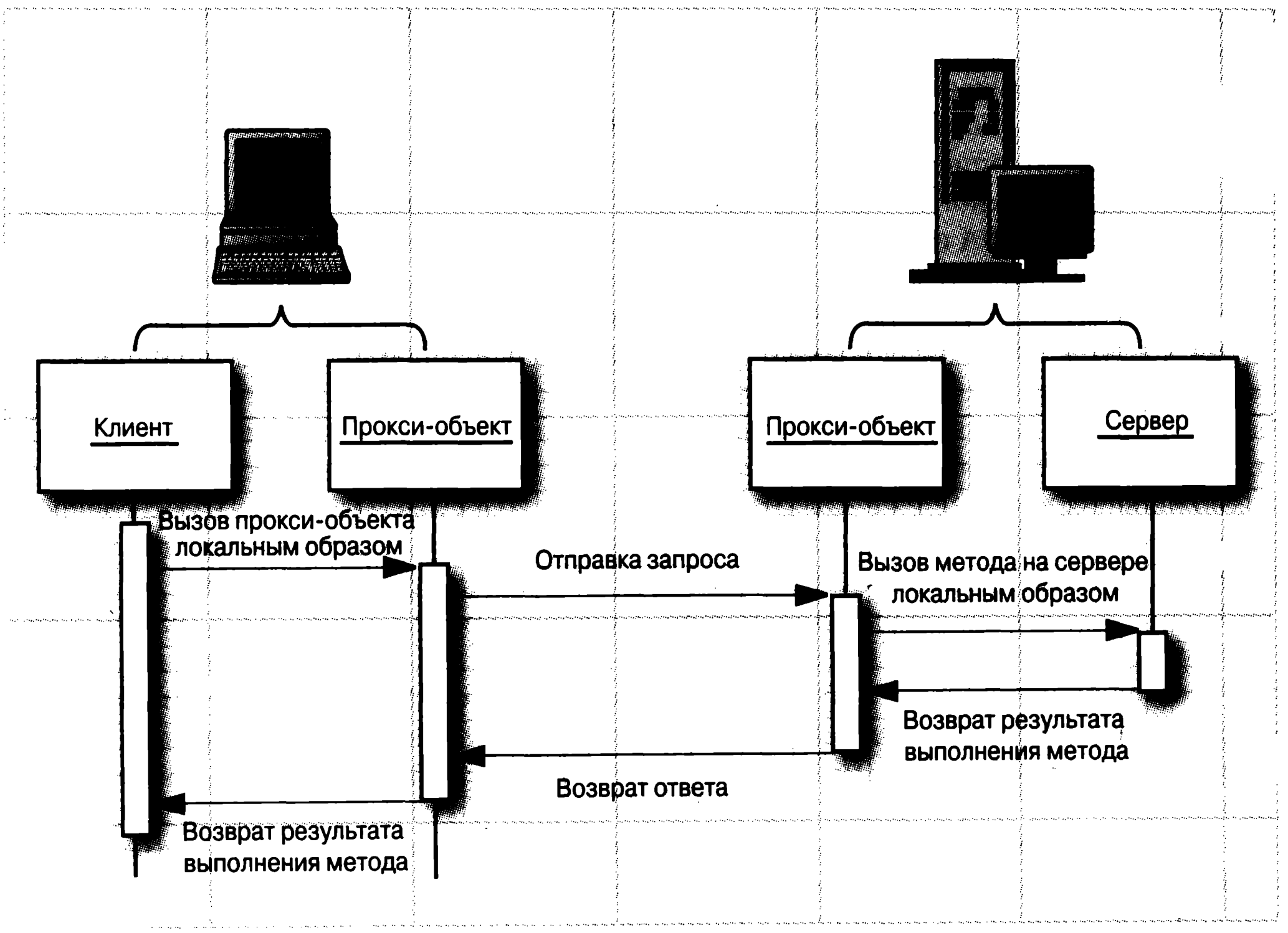


Рис. 11.2. Вызов удаленных методов с помощью прокси-объектов

Каким же образом могут взаимодействовать прокси-объекты? Это зависит от технологии, применяемой для их реализации. Наиболее распространенными являются три технологии.

- Технология CORBA (Common Object Request Broker Architecture — общая архитектура посредника запросов к объектам), обеспечивающая вызовы методов между объектами, созданными на любом языке программирования. Взаимодействие объектов по этой технологии реализуется на основе протокола IIOP

(Internet Inter-ORB Protocol — протокол обмена запросами между посредниками через Интернет).

- Архитектура веб-служб, которая представляет собой целую коллекцию протоколов, зачастую обозначаемых общим префиксом WS-*. Она также подходит для объектов, созданных на любом языке программирования, но для организации взаимодействия между ними предусматривает применение форматов, основанных на XML. Так, для передачи объектов служит протокол SOAP (Simple Object Access Protocol — простой протокол доступа к объектам).
- Технология Java RMI (Remote Method Invocation — вызов удаленных методов), обеспечивающая вызов методов между распределенными объектами Java.

Технологии CORBA и WS-* не требуют использования какого-нибудь конкретного языка программирования. Они позволяют создавать клиентские и серверные прикладные программы на C, C++, Java и любом другом языке программирования. От разработчика требуется предоставить описание интерфейса с указанием сигнатур методов и типов данных, которые его объекты способны обрабатывать. Это описание должно составляться на специальном языке: IDL (Interface Definition Language — язык описания интерфейсов), если речь идет о CORBA, или WSDL (Web Services Description Language — язык описания веб-служб), если речь идет о WS-*.

Долгое время многие считали CORBA объектной моделью будущего. Но, откровенно говоря, CORBA имеет теперь репутацию (порой вполне заслуженную) архитектуры с низкой производительностью, сложной реализацией и проблемами взаимодействия и поэтому не пользуется особым успехом.

Веб-службы считались не менее перспективными, когда они только появились, поскольку они имели более простой вид и, разумеется, основывались на ценных качествах Всемирной паутины и XML. Но в результате неоднократных доработок многих комитетов стек протоколов веб-служб стал намного сложнее, включив в себя многие из тех возможностей, которые уже давно присутствовали в CORBA. Протокол на основе формата XML примечателен тем, что делает данные удобочитаемыми и понятными и тем самым упрощает процесс отладки. Но, с другой стороны, обработка данных в формате XML заметно снижает производительность. Со временем стек протоколов WS-* утратил львиную долю своей привлекательности и стал пользоваться (порой вполне заслуженно) репутацией сложной для реализации технологии с массой проблем совместимости.

Для реализации рассчитанных на взаимодействие программ в коде Java нет совершенно никакой необходимости прибегать к универсальным, но сложным технологиям CORBA и WS-*. Ведь для организации взаимодействия прикладных программ на Java компанией Sun был специально разработан простой механизм, называемый протоколом RMI. Ознакомиться с протоколом RMI стоит даже тем, кто вообще не собирается применять его в своих прикладных программах. На примере довольно простой архитектуры данного протокола можно изучить все основные механизмы, необходимые для программирования распределенных приложений.

Вызовы удаленных методов

Ключевым понятием в области распределенной обработки данных является *вызов удаленного метода*. Допустим, из некоторого кода на одном компьютере, называемом *клиентом*, требуется вызвать метод для так называемого *удаленного объекта* на другом компьютере, называемом *сервером*. Для того чтобы подобное стало возможным,

из этого кода нужно каким-то образом переслать параметры вызываемого метода на другой компьютер, указать серверу на необходимость отыскать удаленный объект и выполнить для него этот метод, а затем найти способ возвратить полученный результат обратно клиенту.

Прежде чем подробно рассматривать данный процесс, следует обратить внимание на то, что терминология "клиент–сервер" касается только единственного вызова метода. Компьютер, вызывающий удаленный метод, выступает при этом вызове в роли клиента, а компьютер, на котором находится объект, требующийся для обработки того же самого вызова, — в роли сервера. При выполнении следующей операции эти компьютеры вполне могут поменяться ролями, т.е. сервер из предыдущего вызова может сам стать клиентом, вызывая удаленный метод для объекта, находящегося на другом компьютере, который прежде выступал в роли клиента, а теперь стал сервером.

Заглушки и упаковка параметров

Когда возникает потребность вызвать из клиентского кода метод для удаленного объекта, на самом деле вызывается обычный метод для прокси-объекта, называемого *заглушкой*, как показано в приведенном ниже примере кода.

```
Warehouse centralWarehouse = get stub object;
double price = centralWarehouse.getPrice("Blackwell Toaster");
```

Заглушка находится на компьютере клиента, а не на сервере и вообще не знает, как связаться с сервером по сети. Она просто упаковывает параметры удаленного метода в блок байтов. Процесс кодирования параметров называется *упаковкой параметров*. Его главной целью является преобразование параметров в формат, пригодный для их передачи с одной виртуальной машины на другую. Если же применяется протокол RMI, объекты кодируются с помощью механизма сериализации, рассматривавшегося в главе 1, а если применяется протокол SOAP, то они кодируются в формате XML.

В целом вызов метода заглушки на клиентском компьютере приводит к созданию информационного блока, состоящего из следующих элементов.

- Идентификатор целевого удаленного объекта.
- Описание вызываемого метода.
- Параметры.

Далее заглушка отправляет эти данные серверу. А на стороне сервера получающий объект выполняет с их помощью следующие действия.

1. Находит требуемый объект.
2. Вызывает нужный метод, передавая ему предоставляемые параметры.
3. Перехватывает возвращаемое значение или исключение, сгенерированное при вызове данного метода.
4. Отправляет пакет, в который упакованы возвращаемые данные, обратно заглушке на клиентском компьютере.

После этого заглушка на клиентском компьютере распаковывает возвращаемое с сервера значение или исключение. Полученное таким образом значение становится возвращаемым из вызова, сделанного заглушкой, а исключение просто выдается повторно на виртуальной машине вызывавшей программы. Весь этот процесс схематически показан на рис. 11.3.

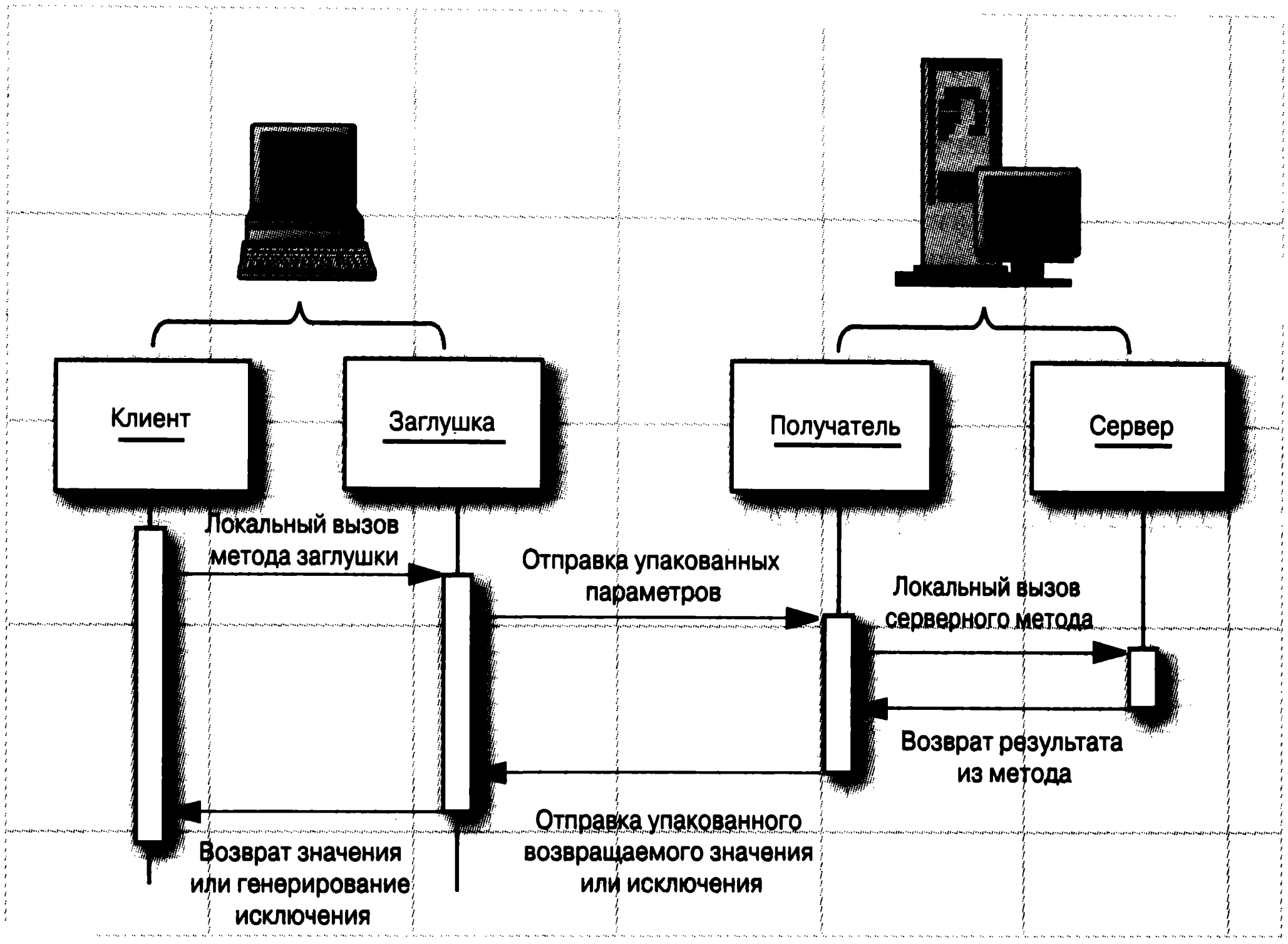


Рис. 11.3. Упаковка и распаковка параметров

Очевидно, что данный процесс непрост. Правда, он выполняется полностью автоматически и в значительной степени прозрачен для программиста. А особенности реализации удаленных объектов и получения клиентских заглушек зависят от конкретной технологии, применяемой для распределенных объектов. В последующих разделах будет подробно рассмотрена технология RMI.

Модель программирования RMI

Итак, начнем представление модели программирования RMI с простого примера, где в роли удаленного объекта будет выступать товарный склад, а клиентская программа будет запрашивать на этом складе данные о цене на конкретный товар. В последующих разделах будет показано, как реализовать и запустить на выполнение серверную и клиентскую программы, необходимые для целей данного примера.

Интерфейсы и реализации

Возможности удаленных объектов выражаются в интерфейсах, которые разделяются между клиентом и сервером. Например, интерфейс, исходный код которого приведен в листинге 11.1, описывает службу, предоставляемую удаленным объектом товарного склада.

Листинг 11.1. Исходный код из файла `warehouse1/Warehouse.java`

```
1  import java.rmi.*;
2
3  /**
4   * Удаленный интерфейс для простого товарного склада
5   * @version 1.0 2007-10-09
6   * @author Cay Horstmann
7  */
8  public interface Warehouse extends Remote
9  {
10   double getPrice(String description) throws RemoteException;
11 }
```

НА ЗАМЕТКУ! В примерах программ из этой главы пакеты не используются. Как станет ясно в дальнейшем, развертывание удаленных приложений — дело непростое, а каталоги с пакетами могут только усложнить его еще больше.

Интерфейсы удаленных объектов должны всегда быть расширением интерфейса `Remote`, входящего в состав пакета `java.rmi`. В объявлениях всех методов из этих интерфейсов должно быть непременно указано, что они могут генерировать исключение типа `RemoteException`. Дело в том, что вызовы удаленных методов по своему характеру являются не такими надежными, как вызовы локальных методов, а это означает, что всегда существует вероятность неудачного завершения вызова удаленного метода. Например, сервер может оказаться временно недоступным, или же в сети могут возникнуть какие-то сбои. Клиентский код должен быть всегда готовым к обработке подобных ситуаций. Именно поэтому при каждом вызове удаленного метода требуется перехватывать исключение типа `RemoteException` и указывать действие, которое должно быть выполнено при неудачном завершении вызова удаленного метода.

Далее на стороне сервера необходимо создать класс, который на самом деле будет выполнять все операции, указанные в удаленном интерфейсе, как показано в листинге 11.2.

Листинг 11.2. Исходный код из файла `warehouse1/WarehouseImpl.java`

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  import java.util.*;
4
5  /**
6   * Этот класс реализует удаленный интерфейс товарного склада Warehouse
7   * @version 1.01 2012-01-26
8   * @author Cay Horstmann
9  */
10 public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
11 {
12   private Map<String, Double> prices;
13
14   public WarehouseImpl() throws RemoteException
15   {
16     prices = new HashMap<>();
17     prices.put("Blackwell Toaster", 24.95);
18     prices.put("ZapXpress Microwave Oven", 49.95);
19   }
20
```

```
21   public double getPrice(String description) throws RemoteException
22   {
23     Double price = prices.get(description);
24     return price == null ? 0 : price;
25   }
26 }
```

НА ЗАМЕТКУ! В конструкторе класса `WarehouseImpl` указывается, что он должен генерировать исключение типа `RemoteException`, поскольку и конструктор суперкласса может сгенерировать такое же исключение. Это обычно происходит в связи с трудностями подключения к службе, отвечающей за отслеживание удаленных объектов.

Можно сказать, что данный класс выступает в роли целевого объекта для вызовов удаленных методов, поскольку он служит расширением класса `UnicastRemoteObject`. Конструктор этого класса делает объекты доступными удаленно. Наследование от класса `UnicastRemoteObject` является наиболее простым путем к желанной цели, поэтому все демонстрируемые в этой главе классы реализации удаленных служб наследуются именно от него.

Иногда наследование от класса `UnicastRemoteObject` может оказаться нежелательным, например, из-за того, что класс реализации уже наследуется от другого класса. В подобных случаях нужно создавать экземпляры удаленных объектов вручную и передавать их статическому методу `exportObject()`. А вместо расширения класса `UnicastRemoteObject` в конструкторе удаленного объекта придется сделать приведенный ниже вызов. Второй параметр вызываемого метода принимает нулевое значение, указывая на то, что для соединения с клиентами может использоваться любой подходящий порт.

```
UnicastRemoteObject.exportObject(this, 0);
```

НА ЗАМЕТКУ! Термин *одноадресный* (`Unicast`) означает, что удаленный объект должен обнаруживаться в результате обращения по одному IP-адресу и к одному порту. Это единственный механизм, который поддерживается в Java SE. Более сложные системы распределенных объектов наподобие JINI позволяют выполнять *многоадресный* поиск удаленных объектов, которые могут находиться на многих серверах.

Реестр RMI

Для доступа к удаленному объекту на сервере клиенту требуется локальный объект заглушки. Каким же образом клиент может запрашивать такой объект? Чаще всего с этой целью вызывается удаленный метод другого удаленного объекта и получается объект заглушки в виде возвращаемого значения. Но в таком случае возникает проблема "курицы и яйца", где трудно отделить причину от следствия, поскольку нужно каким-то образом обнаружить *самый первый* удаленный объект. Для этой цели в JDK предоставляется *служба реестра самозагрузки*.

С помощью этой службы серверная программа регистрирует как минимум один удаленный объект. Для регистрации удаленного объекта требуется только URL по протоколу RMI и ссылка на объект реализации. URL по протоколу RMI начинается с префикса **`rmi:`** и содержит необязательное имя хоста, необязательный номер порта, а также однозначное (желательно) имя удаленного объекта, как показано в приведенном ниже примере.

```
rmi://regserver.mycompany.com:99/central_warehouse
```

По умолчанию выбирается имя хоста `localhost` и номер порта **1099**. Сервер дает команду реестру в указанном месте связать или "привязать" имя к объекту. Ниже приведен код, который требуется для регистрации объекта типа `WarehouseImpl` в реестре RMI на том же самом сервере. А в примере программы из листинга 11.3 демонстрируется построение и регистрация объекта типа `WarehouseImpl`.

```
WarehouseImpl centralWarehouse = new WarehouseImpl();
Context namingContext = new InitialContext();
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

НА ЗАМЕТКУ! Из соображений безопасности приложение может привязывать, отвязывать или снова привязывать ссылки на объекты в реестре только в том случае, если оно находится на том же компьютере, что и реестр. Это исключает вероятность внесения изменений в данные реестра недружелюбными клиентами. Но просматривать объекты все равно может любой клиент.

Клиент может перечислить все объекты, зарегистрированные в реестре RMI, вызвав следующий метод:

```
Enumeration<NameClassPair> e =
    namingContext.list("rmi://regserver.mycompany.com");
```

где `NameClassPair` — вспомогательный класс, в котором содержится имя привязанного объекта, а также его класса. Например, с помощью следующего кода можно отобразить имена всех зарегистрированных объектов:

```
while (e.hasMoreElements())
    System.out.println(e.nextElement().getName());
```

Клиент получает заглушку, необходимую для доступа к удаленному объекту, указывая сервер и имя удаленного объекта, как выделено ниже полужирным.

```
String url = "rmi://regserver.mycompany.com/central_warehouse";
Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);
```

НА ЗАМЕТКУ! Как известно, поддерживать однозначность имен в глобальном реестре очень трудно, поэтому пользоваться этим общепринятым приемом для обнаружения объектов на сервере не следует. Вместо этого следует зарегистрировать с помощью службы реестра самозагрузки относительно небольшое количество удаленных объектов, которые должны быть в состоянии сами обнаруживать другие объекты.

Листинг 11.3. Исходный код из файла `warehouse1/WarehouseServer.java`

```
1  import java.rmi.*;
2  import javax.naming.*;
3
4  /**
5   * Эта серверная программа служит для получения экземпляра
6   * удаленного объекта товарного склада и его регистрации
7   * с помощью службы именования, ожидая от клиентов вызова
8   * удаленных методов
9   * @version 1.12 2007-10-09
10  * @author Cay Horstmann
11 */
12 public class WarehouseServer
```

```
13 {
14   public static void main(String[] args)
15      throws RemoteException, NamingException
16   {
17     System.out.println("Constructing server implementation...");
18     WarehouseImpl centralWarehouse = new WarehouseImpl();
19
20     System.out.println("Binding server implementation to registry...");
21     Context namingContext = new InitialContext();
22     namingContext.bind("rmi:central_warehouse", centralWarehouse);
23
24     System.out.println("Waiting for invocations from clients...");
25   }
26 }
```

В исходном коде из листинга 11.4 демонстрируется клиент, получающий заглушку для доступа к удаленному объекту товарного склада и вызывающий удаленный метод `getPrice()`. Происходящий при этом процесс передачи управления схематически показан на рис. 11.4. Сначала клиент получает объект заглушки типа `Warehouse` и вызывает для него метод `getPrice()`. Затем этот объект заглушки подспудно связывается с сервером, что в конечном итоге приводит к вызову метода `getPrice()` для объекта типа `WarehouseImpl`.

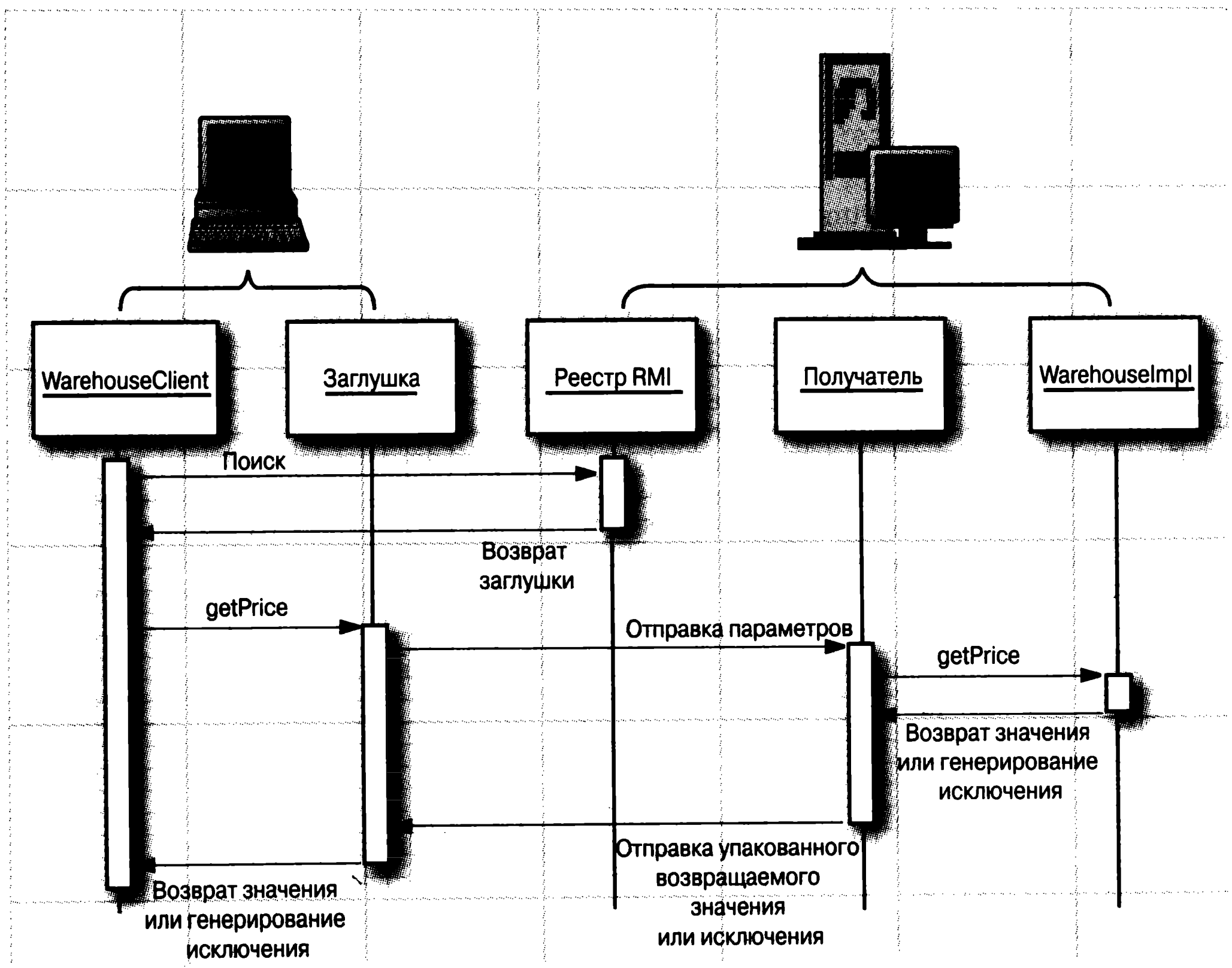


Рис. 11.4. Вызов удаленного метода `getPrice()`

Листинг 11.4. Исходный код из файла warehouse1/WarehouseClient.java

```
1  import java.rmi.*;
2  import java.util.*;
3  import javax.naming.*;
4
5  /**
6   * Клиент, вызывающий удаленный метод
7   * @version 1.0 2007-10-09
8   * @author Cay Horstmann
9  */
10 public class WarehouseClient
11 {
12   public static void main(String[] args)
13       throws NamingException, RemoteException
14   {
15     Context namingContext = new InitialContext();
16
17     System.out.print("RMI registry bindings: ");
18     Enumeration<NameClassPair> e = namingContext.list("rmi://localhost/");
19     while (e.hasMoreElements())
20       System.out.println(e.nextElement().getName());
21
22     String url = "rmi://localhost/central_warehouse";
23     Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);
24
25     String descr = "Blackwell Toaster";
26     double price = centralWarehouse.getPrice(descr);
27     System.out.println(descr + ": " + price);
28   }
29 }
```

javax.naming.InitialContext 1.3

- `InitialContext()`

  Создает контекст именования, который может использоваться для получения доступа к реестру RMI.

javax.naming.Context 1.3

- `static Object lookup(String name)`

  Возвращает объект по указанному имени. Если указанное имя в данный момент не связано ни с одним из объектов, генерируется исключение типа `NamingException`.

- `static void bind(String name, Object obj)`

  Привязывает имя `name` к удаленному объекту `obj`. Если с указанным объектом уже связано какое-нибудь имя, генерируется исключение типа `AlreadyBoundException`.

- `static void unbind(String name)`

  Отвязывает указанное имя от объекта. Допускается отвязывать несуществующее имя.

- `static void rebind(String name, Object obj)`

  Повторно привязывает имя `name` к удаленному объекту *obj*. Заменяет любую имеющуюся привязку имени к объекту.

- `NamingEnumeration<NameClassPair> list(String name)`

  Возвращает список с перечислением привязанных объектов, совпадающих с указанным именем. Для получения списка всех объектов из реестра RMI при вызове данного метода следует указывать символьную строку `"rmi:"`.

javax.naming.NameClassPair 1.3

- `String getName()`

  Получает имя именованного объекта.

- `String getClassName()`

  Получает имя класса, которому принадлежит именованный объект.

java.rmi.Naming 1.1

- `static Remote lookup(String url)`

  Возвращает удаленный объект по указанному URL. Если указанное имя в данный момент не связано ни с одним из объектов, генерируется исключение типа `NotBoundException`.

- `static void bind(String name, Remote obj)`

  Привязывает имя `name` к удаленному объекту *obj*. Если с указанным объектом уже связано какое-нибудь имя, генерируется исключение типа `AlreadyBoundException`.

- `static void unbind(String name)`

  Отвязывает указанное имя от объекта. Если это имя в данный момент не связано ни с одним из объектов, генерируется исключение типа `NotBoundException`.

- `static void rebind(String name, Remote obj)`

  Повторно привязывает имя `name` к удаленному объекту *obj*. Заменяет любую имеющуюся привязку имени к объекту.

- `static String[] list(String url)`

  Возвращает массив символьных строк с URL из реестра, расположенного по указанному URL. Этот массив содержит моментальный снимок всех представленных в реестре имен.

Развертывание программы

Развертывание прикладной программы, использующей протокол RMI, может оказаться непростой задачей, поскольку многое в ходе данного процесса может пойти не так, как нужно, а выводимые при этом сообщения об ошибках могут дать лишь скудные сведения о происходящем. Поэтому процесс развертывания лучше всего тестировать в реальных условиях, разделяя классы для клиента и сервера. С этой целью создаются два отдельных каталога для хранения классов, отвечающих за запуск сервера и клиента, как показано ниже.

```
server/
  WarehouseServer.class
  Warehouse.class
  WarehouseImpl.class

client/
  WarehouseClient.class
  Warehouse.class
```

При развертывании приложений RMI обычно требуется динамически доставлять классы выполняющимся программам. Одним из характерных тому примеров служит реестр RMI. Следует, однако, иметь в виду, что один экземпляр реестра будет обслуживать многие приложения RMI. А это означает, что реестру RMI потребуется доступ к файлам классов всех интерфейсов регистрируемых служб. Но при запуске службы реестра RMI нельзя заранее предугадать, как будут выглядеть запросы на регистрацию. Поэтому реестру RMI приходится динамически загружать файлы классов тех удаленных интерфейсов, которые еще не встречались.

Динамически доставляемые файлы классов распределяются по стандартным веб-серверам. Но в данном случае серверная программа должна сделать доступным для реестра RMI файл `Warehouse.class`, поэтому этот файл размещается в третьем каталоге, `download`, как показано ниже.

```
download/
    Warehouse.class
```

Для обслуживания содержимого этого каталога выделяется отдельный веб-сервер. При развертывании приложения сервер, реестр RMI, веб-сервер и клиент могут размещаться на четырех совершенно разных компьютерах (рис. 11.5). Но для целей тестирования достаточно и одного компьютера.

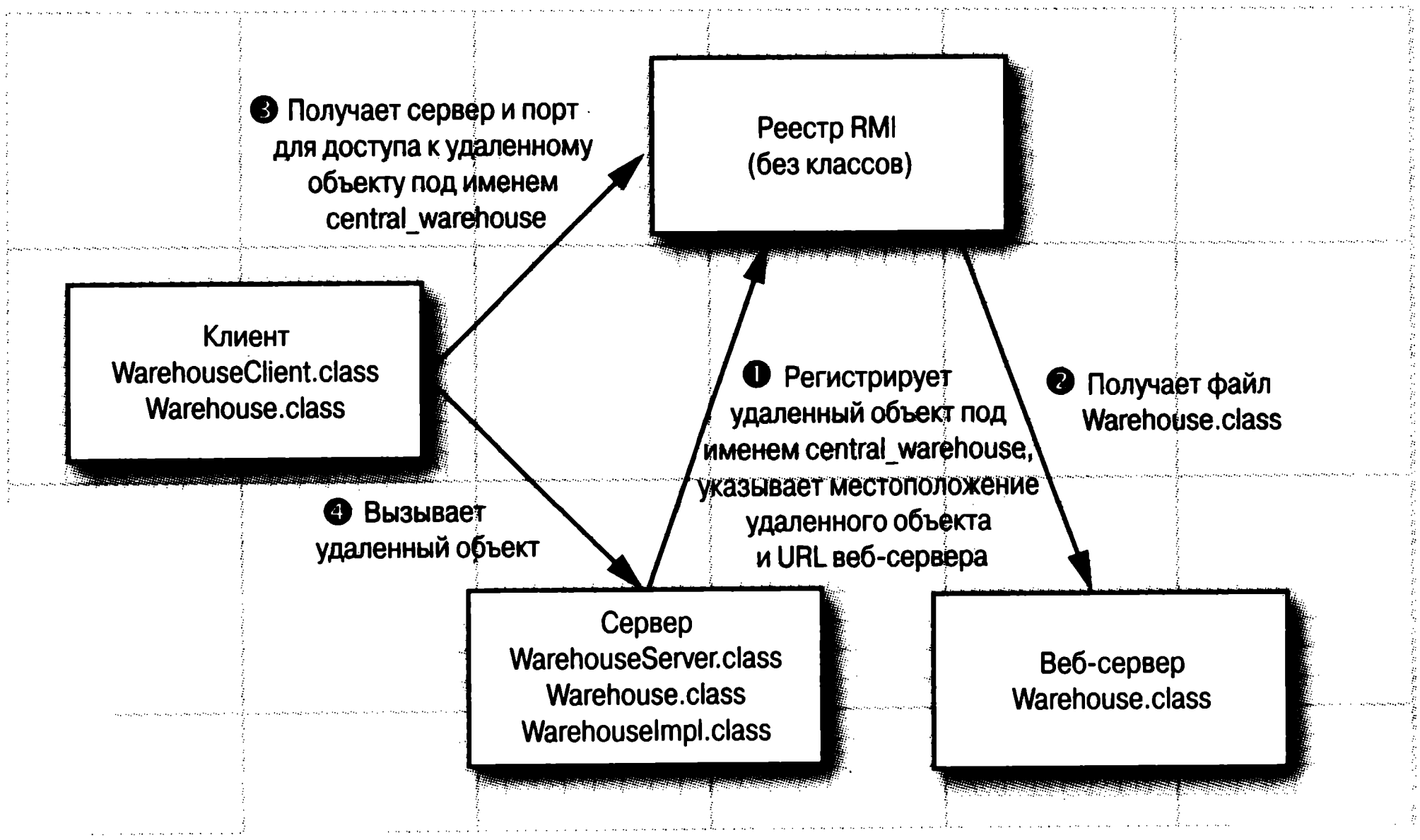


Рис. 11.5. Обращения к серверу в прикладной программе `Warehouse`

НА ЗАМЕТКУ! Из соображений безопасности служба `rmiregistry`, входящая в состав JDK, позволяет делать вызовы для привязки удаленных объектов только с одного и того же хоста. Это означает, что процессы сервера и службы `rmiregistry` должны выполняться на одном и том же компьютере. Тем не менее архитектура RMI допускает и более общую реализацию реестра RMI для поддержки многих серверов.

Для того чтобы протестировать прикладную программу, рассматриваемую здесь в качестве примера, воспользуйтесь веб-сервером `NanoHTTPD`, доступным по адресу `https://github.com/NanoHttpd/nanohttpd`. Этот крошечный веб-сервер реализован в виде единственного файла с исходным кодом на Java. Откройте новое окно консоли, перейдите в каталог `download` и скопируйте в него исходный файл `NanoHTTPD.java`. Далее скомпилируйте этот исходный файл и запустите веб-сервер по следующей команде:

```
java NanoHTTPD 8080
```

В качестве аргумента командной строки следует указать номер порта. Если порт **8080** уже используется на вашем компьютере, укажите номер любого другого доступного порта. После этого откройте еще одно окно консоли, перейдите в каталог, где *отсутствуют* файлы классов, и запустите службу реестра RMI по команде

```
rmiregistry
```

ВНИМАНИЕ! Прежде чем запускать реестр RMI, проверьте, что переменной среды `CLASSPATH` не присвоено никакого значения и что в текущем каталоге действительно отсутствуют файлы классов. В противном случае реестр RMI может обнаружить какие-нибудь поддельные файлы классов, что способно ввести его в заблуждение, когда потребуется загрузить дополнительные классы из другого источника. Подобное поведение имеет свои причины, подробнее поясняемые по адресу `http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/codebase.html`. А по существу, у каждого объекта заглушки имеется запись *кодовой базы*, указывающая место, откуда он был загружен. Эта кодовая база служит для загрузки всех зависимых классов. Если же класс обнаруживается реестром RMI локально, кодовая база будет установлена неправильно.

Итак, все готово к запуску сервера. Откройте третье окно консоли, перейдите в каталог `server` и выполните следующую команду:

```
java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseServer
```

Свойство `java.rmi.server.codebase` указывает на URL, который должен использоваться для обслуживания файлов классов. Серверная программа передает этот URL реестру RMI. Перейдите на мгновение в то окно консоли, где выполняется веб-сервер `NanoHTTPD`. Там вы обнаружите сообщение о том, что файл `Warehouse.class` был передан на обслуживание реестру RMI.

ВНИМАНИЕ! Непременно проверьте, чтобы URL кодовой базы оканчивался знаком *косой черты* (/).

Следует иметь в виду, что рассматриваемая здесь серверная программа не завершается. На первый взгляд это кажется не совсем обычным. Ведь эта программа только создает объект типа `WarehouseImpl` и регистрирует его. На самом деле ее метод `main()` завершается сразу же после регистрации удаленного объекта,

как и следовало ожидать. Но при создании объекта класса, расширяющего класс `UnicastRemoteObject`, запускается отдельный поток исполнения, удерживающий серверную программу в активном состоянии до бесконечности. Благодаря этому серверная программа остается и далее доступной, позволяя клиентам связываться с ней.

И наконец, откройте еще одно, четвертое окно консоли, перейдите в каталог `client` и выполните команду

```
java WarehouseClient
```

В итоге появится короткое сообщение о том, что удаленный метод был успешно вызван (рис. 11.6).

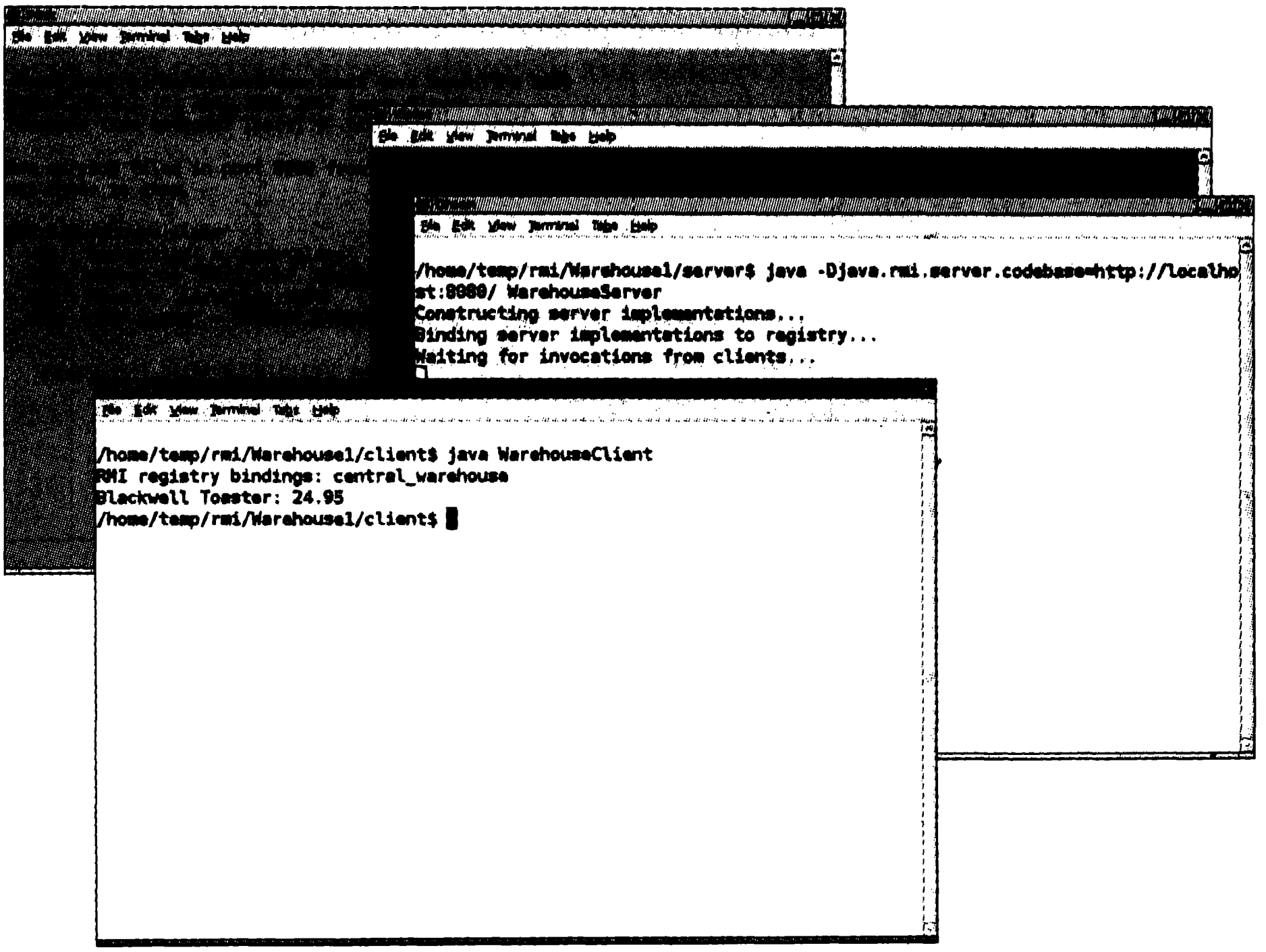


Рис. 11.6. Тестирование приложения RMI

НА ЗАМЕТКУ! Если вы хотите протестировать только основную логику рассматриваемой здесь программы, разместите файлы классов клиента и сервера в одном и том же каталоге и запустите реестр RMI, сервер и клиент в этом каталоге. В связи с тем что загрузка класса RMI вызывает немало затруднений и недоразумений, было решено, что лучше сразу продемонстрировать правильную процедуру настройки динамической загрузки классов.

Протоколирование действий RMI

Если запустить сервер с параметром командной строки, выделенным ниже полужирным, он будет отображать все вызовы удаленных методов в своем окне консоли. Попробуйте сделать это, чтобы получить ясное представление о сетевом трафике RMI.

```
-Djava.rmi.server.logCalls=true WarehouseServer &
```

При желании увидеть все остальные протокольные сообщения вам придется настроить соответствующим образом регистраторы RMI, используя стандартный прикладной интерфейс API для протоколирования в Java. (Подробнее о протоколировании см. в главе 11 первого тома данной книги.) С этой целью создайте файл `logging.properties` со следующим содержимым:

```
handlers=java.util.logging.ConsoleHandler
.level=FINE
java.util.logging.ConsoleHandler.level=FINE
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```

Вместо того чтобы устанавливать глобальный уровень протоколирования `FINE`, вы можете выполнить более точную настройку и установить отдельный уровень для каждого регистратора. Все доступные для настройки регистраторы RMI перечислены в табл. 11.1. Например, для отслеживания действий по загрузке классов укажите следующий параметр настройки:

```
sun.rmi.loader.level=FINE
```

Далее запустите реестр RMI с таким параметром командной строки:

`-J-Djava.util.logging.config.file=`*`каталог`*`/logging.properties`

И наконец, запустите клиент со следующим параметром командной строки:

`-Djava.util.logging.config.file=`*`каталог`*`/logging.properties`

Таблица 11.1. Регистраторы RMI

Имя регистратора	Регистрируемые действия
`sun.rmi.server.call`	Удаленные вызовы на стороне сервера
`sun.rmi.server.ref`	Удаленные ссылки на стороне сервера
`sun.rmi.client.call`	Удаленные вызовы на стороне клиента
`sun.rmi.client.ref`	Удаленные ссылки на стороне клиента
`sun.rmi.dgc`	Распределенная "сборка мусора"
`sun.rmi.loader`	Загрузка классов средствами класса `RMIClassLoader`
`sun.rmi.transport.misc`	Действия на транспортном уровне
`sun.rmi.transport.tcp`	Привязка и подключение по протоколу TCP
`sun.rmi.transport.proxy`	Туннелирование по протоколу TCP

Ниже приведен пример протокольного сообщения, уведомляющего о затруднениях при загрузке класса. В данном случае оно сообщает, что реестру RMI не удалось найти класс `Warehouse` вследствие остановки веб-сервера.

```
FINE: RMI TCP Connection(1)-127.0.1.1: (port 1099) op = 80
Oct 13, 2007 4:43:30 PM sun.rmi.server.LoaderHandler loadProxyClass
FINE: RMI TCP Connection(1)-127.0.1.1: interfaces = [java.rmi.Remote,
Warehouse], codebase =
"http://localhost:8080/"
Oct 13, 2007 4:43:30 PM sun.rmi.server.LoaderHandler loadProxyClass
FINE: RMI TCP Connection(1)-127.0.1.1: proxy class resolution failed
java.lang.ClassNotFoundException: Warehouse
```

Параметры и возвращаемые значения в удаленных методах

В начале вызова удаленного метода из виртуальной машины клиента виртуальной машине сервера должны быть переданы параметры, а по завершении вызова в обратном направлении должно быть соответственно передано возвращаемое значение. Передача значений от одной виртуальной машины к другой разделяется на две разновидности: передачу удаленных и неудаленных объектов. Допустим, клиент типа `WarehouseServer` может передавать ссылку из объекта типа `Warehouse` (т.е. заглушки, через которую вызывается удаленный объект товарного склада) на другой удаленный метод. Это служит примером передачи удаленного объекта. Но чаще всего в качестве параметров методам передаются обычные объекты Java, а не заглушки для доступа к удаленным объектам. Примером тому может служить параметр типа `String` метода `getPrice()` из рассмотренного ранее первого примера прикладной программы.

Передача удаленных объектов

При передаче ссылки на удаленный объект от одной виртуальной машины к другой отправитель и получатель удаленного объекта содержат ссылку на одну и ту же сущность. Эта ссылка не указывает на конкретную область оперативной памяти, что имело бы смысл только для единственной машины, а состоит из сетевого адреса и однозначного идентификатора удаленного объекта. Все эти сведения инкапсулируются в объекте заглушки.

По существу, передача удаленной ссылки очень похожа на передачу локальной ссылки в пределах виртуальной машины. Но не следует забывать, что вызов метода по удаленной ссылке выполняется намного медленнее и потенциально является менее надежным, чем вызов метода по локальной ссылке.

Передача неудаленных объектов

Рассмотрим в качестве примера параметр типа `String` метода `getPrice()`. Строковое значение должно копироваться из клиента на сервер. Представить, как именно копия символьной строки может передаваться по сети, совсем не трудно. Механизм RMI может создавать копии и более сложных объектов — главное, чтобы они были сериализируемыми. Для отправки объектов по сетевому соединению в RMI используется механизм сериализации, о котором более подробно рассказывалось в главе 1. Это означает, что в качестве параметров и возвращаемых типов могут быть указаны любые классы, которые реализуют интерфейс `Serializable`.

Передача параметров путем их сериализации отражается некоторым образом на семантике удаленных методов. В частности, при передаче объектов локальному методу на самом деле передаются *ссылки* на эти объекты. Если же в удаленном методе к обычному объекту параметра применяется модифицирующий метод, это обстоятельство не останется незамеченным в вызывающей части программы. А если удаленный метод модифицирует сериализированный параметр, то внесенные изменения затронут и копию, но вызывающая часть программы их даже не заметит.

Таким образом, для передачи значений между виртуальными машинами применяются два механизма.

- Объекты классов, реализующих интерфейс `Remote`, передаются в виде удаленных ссылок.
- Объекты классов, реализующих интерфейс `Serializable` (а не `Remote`), копируются с применением сериализации.

Все это происходит автоматически и не требует никакого вмешательства со стороны программиста. Но не следует забывать, что сериализация крупных объектов может выполняться очень медленно и что удаленный метод не в состоянии модифицировать сериализованные параметры. Этих осложнений можно, конечно, избежать, передавая только удаленные ссылки. Но и за это приходится расплачиваться, поскольку вызов методов по удаленным ссылкам обходится намного дороже с точки зрения потребляемых ресурсов, чем вызов локальных методов. Ясное понимание подобных затрат позволяет сделать осознанный выбор при разработке удаленных служб.

НА ЗАМЕТКУ! Удаленные объекты обрабатываются системой "сборки мусора" автоматически, т.е. таким же образом, как и локальные объекты. Совсем иначе дело обстоит с распределенной системой "сборки мусора". Когда локальная система "сборки мусора" обнаруживает, что удаленная ссылка не находит дальнейшего применения, она уведомляет распределенную систему "сборки мусора", что данный клиент больше не ссылается на сервер. А когда сервер больше не используется ни одним из клиентов, он помечается как "мусор".

Как именно передаются удаленные и сериализированные объекты, демонстрируется на примере следующей программы. С этой целью в интерфейс `Warehouse` внесены соответствующие изменения, как показано в листинге 11.5. Теперь при получении списка ключевых слов из удаленного склада возвращается наиболее подходящий товар в виде объекта типа `Product`.

Листинг 11.5. Исходный код из файла `warehouse2/Warehouse.java`

```
1  import java.rmi.*;
2  import java.util.*;
3
4  /**
5   * Удаленный интерфейс для простого товарного склада
6   * @version 1.0 2007-10-09
7   * @author Cay Horstmann
8  */
9  public interface Warehouse extends Remote
10 {
11   double getPrice(String description) throws RemoteException;
12   Product getProduct(List<String> keywords) throws RemoteException;
13 }
```

Параметр `getProduct` относится к типу `List<String>`. Значение этого параметра должно принадлежать классу, поддающемуся сериализации и реализующему интерфейс `List<String>`, как, например, обобщенный класс `ArrayList<String>`. (В данном примере клиент передает значение, получаемое в результате вызова метода `Arrays.asList()`. Правда, этот метод также гарантирует возврат списка сериализированных объектов.)

Возвращаемый объект типа `Product` инкапсулирует описание, цену и место нахождения товара, как следует из листинга 11.6. Необходимо также заметить, что класс

Product является сериализируемым. Сервер создает объект типа Product, а клиент получает его копию, как показано на рис. 11.7.

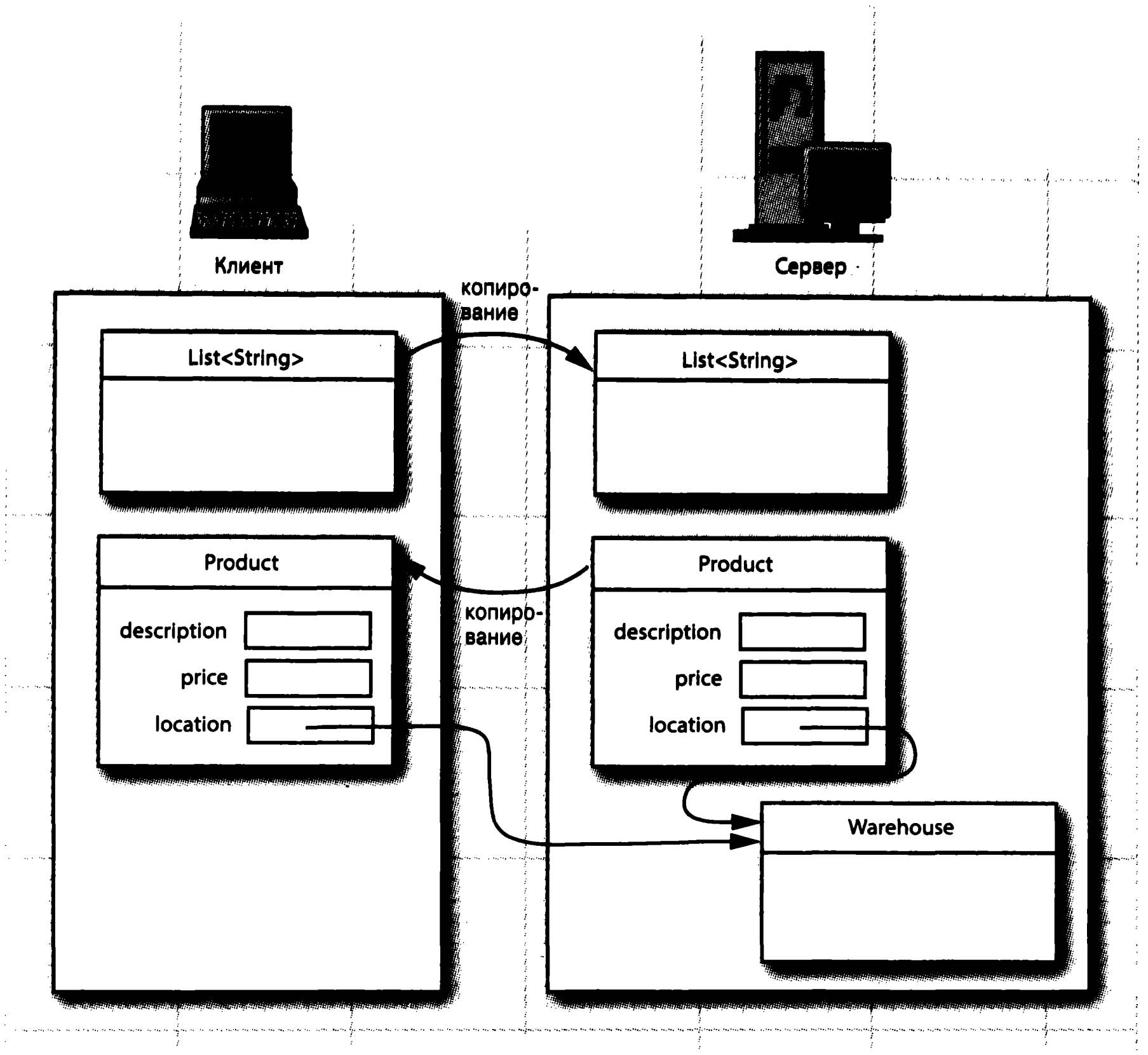


Рис. 11.7. Копирование локальных объектов параметров и результатов

Следует, однако, иметь в виду следующую особенность: у класса Product имеется поле экземпляра типа Warehouse, т.е. удаленного интерфейса. Объект типа Warehouse *не* сериализируется, а следовательно, может иметь огромное количество данных о своем состоянии. Поэтому вместо него клиент получает заглушку, ссылающуюся на удаленный объект типа Warehouse. Эта заглушка может отличаться от заглушки centralWarehouse, для которой вызывался метод getProduct(). В данной реализации имеются два вида товаров: тостеры и книги, находящиеся на разных товарных складах.

Листинг 11.6. Исходный код из файла warehouse2/Product.java

```
1  import java.io.*;
2
3  public class Product implements Serializable
4  {
5    private String description;
6    private double price;
```

```
7     private Warehouse location;
8
9     public Product(String description, double price)
10    {
11      this.description = description;
12      this.price = price;
13    }
14
15    public String getDescription()
16    {
17      return description;
18    }
19
20    public double getPrice()
21    {
22      return price;
23    }
24
25    public Warehouse getLocation()
26    {
27      return location;
28    }
29
30    public void setLocation(Warehouse location)
31    {
32      this.location = location;
33    }
34  }
```

Динамическая загрузка классов

В следующем примере программы демонстрируется еще одна особенность обращения с удаленными объектами. Список строк с ключевыми словами отправляется серверу, в результате чего из товарного склада возвращается товар в виде экземпляра класса `Product`. Безусловно, клиентской программе в этом случае потребуется файл класса `Product.class` для компиляции. Но на тот случай, если по ключевым словам не удастся отыскать подходящий товар, в рассматриваемом здесь примере серверной программы предусмотрен возврат объекта такого товара, который должен понравиться всем: данной книги по Java. Этот объект является экземпляром класса `Book`, производного от класса `Product`.

Во время компиляции клиент может и не знать о существовании класса `Book`. Но при запуске он должен быть способен выполнять методы из класса `Book`, которые переопределяют методы из класса `Product`. Этим демонстрируется, что у клиента должна быть возможность загружать дополнительные классы во время выполнения. И для этого в клиенте применяется тот же самый механизм, что и в реестре RMI. Классы обслуживаются веб-сервером, класс RMI-сервера сообщает клиенту их URL, а клиент использует эти URL для формирования запросов по протоколу HTTP, чтобы загрузить файлы соответствующих классов.

Всякий раз, когда программа загружает новый код из другого места в сети, возникает угроза безопасности. Вследствие этого во всех приложениях RMI с динамической загрузкой классов следует непременно применять диспетчер защиты. (Подробнее о диспетчерах защиты и загрузчиках классов см. в главе 9.) Программы, в которых применяется механизм RMI, должны устанавливать диспетчер защиты для контроля

над действиями динамически загружаемых классов. Для установки диспетчера защиты служит следующий оператор:

```
System.setSecurityManager(new SecurityManager());
```

На заметку! Если все классы доступны локально, то диспетчер защиты на самом деле не нужен. Так, если все файлы классов, требующиеся в программе, известны на этапе развертывания, то все они могут быть развернуты локально. Но нередко клиентская или серверная программа со временем усовершенствуется, и в нее вводятся новые классы. В подобных случаях динамическая загрузка классов оказывается очень выгодной. Но не следует забывать, что всякий раз, когда код загружается из другого источника, обязательно требуется диспетчер защиты.

По умолчанию диспетчер защиты типа `SecurityManager` накладывает на весь код программы ограничение на установление сетевых соединений. Но ведь программе требуется устанавливать сетевые соединения со следующими удаленными местами:

- веб-сервером, отвечающим за загрузку удаленных классов;
- реестром RMI;
- удаленными объектами.

С целью разрешить подобные сетевые операции создается файл правил защиты. (Подробнее о файлах правил защиты см. в главе 9.) Ниже приведено содержимое файла правил защиты, разрешающего прикладной программе устанавливать любые сетевые соединения через порт с номером, как минимум, **1024**. (В протоколе RMI по умолчанию выбирается порт **1099**, а удаленные объекты обычно запрашивают порты с номерами больше **1024**. Для динамической загрузки классов в данном примере используется порт **8080**.)

```
grant
{
  permission java.net.SocketPermission
     "*:1024-65535", "connect";
};
```

Диспетчеру защиты нужно указать, что он должен считывать файл правил защиты, установив в свойстве `java.security.policy` имя этого файла. С этой целью делается следующий вызов:

```
System.setProperty("java.security.policy", "rmi.policy");
```

С другой стороны, можно задать параметр системного свойства в командной строке, как показано ниже.

```
-Djava.security.policy=rmi.policy
```

Чтобы запустить на выполнение рассматриваемую здесь прикладную программу, сначала удалите реестр RMI, веб-сервер и серверную программу из предыдущего примера, а затем откройте четыре окна консоли и выполните следующие действия.

1. Скомпилируйте исходные файлы классов интерфейса, реализации, клиента и сервера по команде

   ```
   javac *.java
   ```

2. Создайте три каталога `client`, `server` и `download` и заполните их следующим образом:

```
client/
   WarehouseClient.class
   Warehouse.class
   Product.class
   client.policy
server/
   Warehouse.class
   Product.class
   Book.class
   WarehouseImpl.class
   WarehouseServer.class
   server.policy
download/
   Warehouse.class
   Product.class
   Book.class
```

3. Перейдите в первом окне консоли в тот каталог, где *отсутствуют* файлы классов, и запустите реестр RMI.
4. Перейдите во втором окне консоли в каталог `download` и запустите веб-сервер `NanoHTTPD`.
5. Перейдите в третьем окне консоли в каталог `server` и запустите сервер по команде

   ```
   java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseServer
   ```

6. Перейдите в четвертом окне консоли в каталог `client` и запустите клиент следующим образом:

   ```
   java WarehouseClient
   ```

В листинге 11.7 представлен исходный код класса `Book`. Следует иметь в виду, что метод `getDescription()` переопределяется таким образом, чтобы отображать номер ISBN. Благодаря этому клиентская программа показывает при запуске номер ISBN для данной книги по Java, а это подтверждает то обстоятельство, что класс `Book` загружен динамически. В листинге 11.8 приведен исходный код реализации товарного склада (удаленного интерфейса `Warehouse`). У этого склада имеется ссылка на резервный склад. Если товар не удается отыскать на основном товарном складе, поиск начинается на резервном складе. А в листинге 11.9 приведен исходный код серверной программы. Только центральный товарный склад введен в реестр RMI. Следует также отметить, что удаленная ссылка на резервный товарный склад может передаваться клиенту, несмотря на то, что он не введен в реестр RMI. Подобное происходит всякий раз, когда не удается отыскать ни одного товара по переданным ключевым словам, и тогда клиенту отправляется объект, представляющий книгу по Java (в поле `location` этого объекта содержится ссылка на резервный товарный склад).

Листинг 11.7. Исходный код из файла `warehouse2/Book.java`

```
1  /**
2   * Книга — это товар с номером ISBN
3   * @version 1.0 2007-10-09
4   * @author Cay Horstmann
5  */
6  public class Book extends Product
7  {
```

```
8     private String isbn;
9
10    public Book(String title, String isbn, double price)
11    {
12       super(title, price);
13       this.isbn = isbn;
14    }
15
16    public String getDescription()
17    {
18       return super.getDescription() + " " + isbn;
19    }
20 }
```

Листинг 11.8. Исходный код из файла `warehouse2/WarehouseImpl.java`

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  import java.util.*;
4
5  /**
6   * Этот класс реализует удаленный интерфейс товарного склада Warehouse
7   * @version 1.01 2012-01-26
8   * @author Cay Horstmann
9  */
10 public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
11 {
12   private Map<String, Product> products;
13   private Warehouse backup;
14
15   /**
16    * Создает реализацию товарного склада
17   */
18   public WarehouseImpl(Warehouse backup) throws RemoteException
19   {
20     products = new HashMap<>();
21     this.backup = backup;
22   }
23
24   public void add(String keyword, Product product)
25   {
26     product.setLocation(this);
27     products.put(keyword, product);
28   }
29
30   public double getPrice(String description) throws RemoteException
31   {
32     for (Product p : products.values())
33       if (p.getDescription().equals(description)) return p.getPrice();
34     if (backup == null) return 0;
35     else return backup.getPrice(description);
36   }
37
38   public Product getProduct(List<String> keywords) throws RemoteException
39   {
40     for (String keyword : keywords)
41     {
```

```
42        Product p = products.get(keyword);
43        if (p != null) return p;
44     }
45     if (backup != null)
46        return backup.getProduct(keywords);
47     else if (products.values().size() > 0)
48        return products.values().iterator().next();
49     else
50        return null;
51   }
52 }
```

Листинг 11.9. Исходный код из файла `warehouse2/WarehouseServer.java`

```
1  import java.rmi.*;
2  import javax.naming.*;
3
4  /**
5   * Эта серверная программа служит для получения экземпляра
6   * удаленного объекта товарного склада и его регистрации
7   * с помощью службы именования, ожидая от клиентов вызова
8   * удаленных методов
9   * @version 1.12 2007-10-09
10  * @author Cay Horstmann
11 */
12 public class WarehouseServer
13 {
14   public static void main(String[] args)
15      throws RemoteException, NamingException
16   {
17     System.setProperty("java.security.policy", "server.policy");
18     System.setSecurityManager(new SecurityManager());
19
20     System.out.println("Constructing server implementation...");
21     WarehouseImpl backupWarehouse = new WarehouseImpl(null);
22     WarehouseImpl centralWarehouse = new WarehouseImpl(backupWarehouse);
23
24     centralWarehouse.add("toaster", new Product(
25        "Blackwell Toaster", 23.95));
26     backupWarehouse.add("java", new Book(
27        "Core Java vol. 2", "0132354799", 44.95));
28
29     System.out.println("Binding server implementation to registry...");
30     Context namingContext = new InitialContext();
31     namingContext.bind("rmi:central_warehouse", centralWarehouse);
32
33     System.out.println("Waiting for invocations from clients...");
34   }
35 }
```

Удаленные ссылки на несколько интерфейсов

Удаленный класс может реализовать несколько интерфейсов. Рассмотрим в качестве примера приведенный ниже удаленный интерфейс `ServiceCenter`.

```
public interface ServiceCenter extends Remote
{
  int getReturnAuthorization(Product prod) throws RemoteException;
}
```

Допустим теперь, что класс `WarehouseImpl` реализует этот интерфейс таким же образом, как и интерфейс `Warehouse`. При передаче удаленной ссылки на такой центр обслуживания другой виртуальной машине получатель приобретает заглушку, имеющую доступ к удаленным методам как в интерфейсе `ServiceCenter`, так и в интерфейсе `Warehouse`. Для того чтобы выяснить, реализует ли такой интерфейс конкретный удаленный объект, можно воспользоваться операцией `instanceof`. Допустим, удаленный объект получается с помощью переменной типа `Warehouse` следующим образом:

```
Warehouse location = product.getLocation();
```

Этот удаленный объект может представлять или не представлять центр обслуживания. Выяснить это можно, выполнив следующую проверку:

```
if (location instanceof ServiceCenter)
```

Если проверка проходит успешно, переменную `location` можно привести к типу `ServiceCenter` и вызвать метод `getReturnAuthorization()`.

Удаленные объекты и методы `equals()`, `hashCode()` и `clone()`

Объекты, вводимые во множества, должны непременно переопределять метод `equals()`. А если используется хеш-множество или хеш-таблица, то должен быть определен и метод `hashCode()`. Но при попытке сравнить удаленные объекты возникает следующее затруднение. Для того чтобы выяснить, имеют ли два сравниваемых удаленных объекта одинаковое содержимое, при вызове метода `equals()` придется установить соединение с обслуживающими эти объекты серверами. Как и вызов любого удаленного метода, вызов этого метода может завершиться неудачно. Но метод `equals()` из класса `Object` не предусматривает в таком случае генерирование исключения типа `RemoteException`, а ведь все методы из удаленного интерфейса должны генерировать это исключение. Вследствие того что метод из подкласса не может генерировать больше исключений, чем заменяемый им метод из суперкласса, определить метод `equals()` в удаленном интерфейсе нельзя. Это же относится и к методу `hashCode()`.

Вместо этого методы `equals()` и `hashCode()` для объектов заглушек должны только проверять место расположения удаленных объектов. Если заглушки ссылаются на один и тот же удаленный объект, то они считаются в методе `equals()` равными. А если они ссылаются на разные удаленные объекты, то считаются неравными даже в том случае, если имеют одинаковое содержимое. Аналогичным образом хеш-код должен вычисляться только из идентификатора объекта.

По тем же техническим причинам в удаленных ссылках отсутствует метод `clone()`. Если бы удаленный вызов был сделан в методе `clone()`, чтобы дать серверу команду клонировать объект реализации, то методу `clone()` потребовалось бы сгенерировать исключение типа `RemoteException`. Но в суперклассе `Object` ему не разрешается генерировать никаких исключений, кроме `CloneNotSupportedException`.

Таким образом, пользоваться удаленными ссылками во множествах и хеш-таблицах можно, но не забывать, что при сравнении на равенство и хешировании содержимое удаленных объектов не учитывается. Да и клонировать удаленные ссылки просто не удастся.

Активизация удаленных объектов

В предыдущих примерах для создания экземпляров и регистрации объектов, которые клиенты могли бы вызывать удаленно, использовалась серверная программа. Но в некоторых случаях создание многих экземпляров удаленных объектов и ожидание соединения с ними, независимо от того, требуются ли они клиентским объектам или нет, может оказаться неэффективным. Механизм *активизации* позволяет откладывать создание объектов таким образом, чтобы удаленный объект конструировался только в том случае, если хотя бы один клиент вызовет для него удаленный метод. При активизации удаленных объектов клиентский код остается без изменений. Клиент просто запрашивает удаленную ссылку и делает по ней вызов.

Но серверная программа в данном случае заменяется программой активизации. Эта программа конструирует для объектов, которые должны быть созданы в дальнейшем, *дескрипторы активизации* и привязывает к ним получателей с помощью службы именования для вызова удаленных методов. При первом вызове сведения, содержащиеся в дескрипторе активизации, используются для создания соответствующего объекта.

Используемый подобным образом удаленный объект должен расширять класс `Activatable`, а не класс `UnicastRemoteObject`. И разумеется, его класс должен реализовывать один или более удаленный интерфейс, как показано в приведенном ниже примере.

```
class WarehouseImpl
  extends Activatable
  implements Warehouse
{
  . . .
}
```

В связи с тем что создание объекта откладывается на более позднее время, оно должно происходить в стандартизированной форме. Для этого нужно предоставить конструктор, принимающий два параметра.

- Идентификатор активизации, который просто передается конструктору суперкласса.
- Один объект, содержащий все сведения о построении объектов и заключенный в оболочку типа `MarshalledObject`.

Если для построения объектов потребуется несколько параметров, все они должны быть упакованы в один объект. Для этой цели всегда можно воспользоваться обычным массивом `Object[]` или же списочным массивом типа `ArrayList`. При построении дескриптора активизации объект типа `MarshalledObject` создается на основании сведений о построении:

```
MarshalledObject<T> param = new MarshalledObject<T>(constructionInfo);
```

Для получения десериализированных данных о построении в конструкторе объекта реализации необходимо вызвать метод `get()` из класса `MarshalledObject`, как показано ниже.

```
T constructionInfo = param.get();
```

Чтобы продемонстрировать принцип действия механизма активизации, внесем изменения в класс `WarehouseImpl` таким образом, чтобы получать сведения о построении из отображения, содержащего описания товаров на складе и цен на них.

Все эти сведения упаковываются в оболочку типа `MarshalledObject`, а затем извлекаются оттуда в следующем конструкторе:

```
public WarehouseImpl(
    ActivationID id, MarshalledObject<Map<String, Double>> param)
    throws RemoteException, ClassNotFoundException, IOException
{
  super(id, 0);
  prices = param.get();
  System.out.println("Warehouse implementation constructed.");
}
```

Если конструктору суперкласса в качестве второго параметра передается нулевое значение, это означает, что библиотека RMI должна назначить подходящий номер приемному порту. Этот конструктор предусматривает вывод на экран сообщения о том, что объект товарного склада активизирован по запросу.

НА ЗАМЕТКУ! На самом деле удаленные объекты совсем не обязательно должны расширять класс `Activatable`. Вместо этого достаточно ввести в конструктор класса сервера следующий вызов статического метода:

```
Activatable.exportObject(this, id, 0)
```

А теперь перейдем непосредственно к программе активизации. Прежде всего нужно определить группу активизации и описать в ней общие параметры для запуска виртуальной машины, содержащей удаленные объекты. Наиболее важным из этих параметров является правило защиты. Дескриптор группы активизации создается следующим образом:

```
Properties props = new Properties();
props.put("java.security.policy", "/path/to/server.policy");
ActivationGroupDesc group = new ActivationGroupDesc(props, null);
```

Второй параметр в этом конструкторе служит для описания специальных параметров командной строки. Но в рассматриваемом здесь примере такие параметры не нужны, поэтому в качестве второго параметра данному конструктору передается пустая ссылка `null`.

Далее нужно создать идентификатор группы, сделав следующий вызов:

```
ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);
```

После этого можно приступать непосредственно к созданию самих дескрипторов активизации. С этой целью для каждого объекта, который должен создаваться по запросу, нужно предоставить следующие сведения.

- Идентификатор группы активизации для виртуальной машины, где должен создавать данный объект.
- Имя класса (например, `"WarehouseImpl` или `"com.mycompany.MyClassImpl"`).
- Символьная строка с URL, по которому должны загружаться файлы классов (это должен быть базовый URL, не включающий в себя никаких путей к пакетам).
- Упакованные сведения о построении.

Ниже приведен пример кода для построения дескриптора активизации.

```
MarshalledObject param = new MarshalledObject(constructionInfo);
ActivationDesc desc = new ActivationDesc(
    id, "WarehouseImpl", "http://myserver.com/download/", param);
```

Этот дескриптор далее передается статическому методу `Activatable.register()`, который возвращает объект некоторого класса, реализующего удаленные интерфейсы класса реализации. Полученный в итоге объект можно затем привязать к имени с помощью службы именования:

```
Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

В отличие от серверных программ из предыдущих примеров, работа программы активизации завершается после регистрации и привязки получателей, а создание всех удаленных объектов происходит только при первом вызове удаленного метода. В листингах 11.10 и 11.11 представлен исходный код программы активизации и реализации активизируемого товарного склада. А исходный код удаленного интерфейса товарного склада и клиентской программы остается прежним.

Чтобы запустить эту серверную программу, выполните следующие действия.

1. Скомпилируйте все исходные файлы.
2. Распределите файлы классов по каталогам следующим образом:

```
client/
   WarehouseClient.class
   Warehouse.class
server/
   WarehouseActivator.class
   Warehouse.class
   WarehouseImpl.class
   server.policy
download/
   Warehouse.class
   WarehouseImpl.class
rmi/
   rmid.policy
```

3. Запустите реестр RMI из каталога `rmi`, где не должно быть никаких файлов классов.
4. Запустите демон активизации RMI из того же самого каталога `rmi` по следующей команде:

```
rmid -J-Djava.security.policy=rmid.policy
```

5. Программа `rmid`, выполняющая функции демона, перехватывает запросы на активизацию объектов и затем активизирует соответствующие объекты в отдельной виртуальной машине. Для запуска виртуальной машины программе `rmid` требуются определенные полномочия, которые задаются в файле правил защиты из листинга 11.12. Параметр командной строки **`-J`** служит для передачи параметров виртуальной машине, под управлением которой работает демон активизации.
6. Запустите веб-сервер `NanoHTTPD` из каталога `download`.
7. Запустите программу активизации из каталога `server`:

```
java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseActivator
```

8. Эта программа завершит свою работу после регистрации и привязки получателей запросов на активизацию с помощью службы именования. (У вас может возникнуть вопрос: зачем указывать кодовую базу, если она и так предоставляется в конструкторе дескриптора активизации? Дело в том, что эти данные обрабатываются только демоном активизации RMI. Поэтому кодовая база все равно потребуется реестру RMI для загрузки классов удаленных интерфейсов.)
9. Запустите клиентскую программу из каталога `client` по команде

   ```
   java WarehouseClient
   ```

10. Клиентская программа выведет на экран уже знакомое вам описание товара, найденного на складе. При первом запуске этой программы в окне оболочки демона активизации можно будет также увидеть все сообщения конструктора.

Листинг 11.10. Исходный код из файла `activation/WarehouseActivator.java`

```
1  import java.io.*;
2  import java.rmi.*;
3  import java.rmi.activation.*;
4  import java.util.*;
5  import javax.naming.*;
6
7  /**
8   * Эта серверная программа служит для получения экземпляра
9   * удаленного объекта товарного склада и его регистрации
10  * с помощью службы именования, ожидая от клиентов вызова
11  * удаленных методов
12  * @version 1.13 2012-01-26
13  * @author Cay Horstmann
14 */
15 public class WarehouseActivator
16 {
17   public static void main(String[] args) throws RemoteException,
18      NamingException, ActivationException, IOException
19   {
20     System.out.println("Constructing activation descriptors...");
21
22     Properties props = new Properties();
23     // использовать файл server.policy из текущего каталога
24     props.put("java.security.policy", new File(
25        "server.policy").getCanonicalPath());
26     ActivationGroupDesc group = new ActivationGroupDesc(props, null);
27     ActivationGroupID id = ActivationGroup.getSystem()
28        .registerGroup(group);
29
30     Map<String, Double> prices = new HashMap<>();
31     prices.put("Blackwell Toaster", 24.95);
32     prices.put("ZapXpress Microwave Oven", 49.95);
33     MarshalledObject<Map<String, Double>> param =
34        new MarshalledObject<Map<String, Double>>(prices);
35
36     String codebase = "http://localhost:8080/";
37
38     ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl",
39        codebase, param);
```

```
40
41      Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);
42
43      System.out.println("Binding activable implementation to registry...");
44      Context namingContext = new InitialContext();
45      namingContext.bind("rmi:central_warehouse", centralWarehouse);
46      System.out.println("Exiting...");
47    }
48 }
```

Листинг 11.11. Исходный код из файла `activation/WarehouseImpl.java`

```
1  import java.io.*;
2  import java.rmi.*;
3  import java.rmi.activation.*;
4  import java.util.*;
5
6  /**
7   * Этот класс реализует удаленный интерфейс товарного склада Warehouse
8   * @version 1.0 2007-10-20
9   * @author Cay Horstmann
10 */
11 public class WarehouseImpl extends Activatable implements Warehouse
12 {
13   private Map<String, Double> prices;
14
15   public WarehouseImpl(ActivationID id,
16      MarshalledObject<Map<String, Double>> param)
17      throws RemoteException, ClassNotFoundException, IOException
18   {
19     super(id, 0);
20     prices = param.get();
21     System.out.println("Warehouse implementation constructed.");
22   }
23
24   public double getPrice(String description) throws RemoteException
25   {
26     Double price = prices.get(description);
27     return price == null ? 0 : price;
28   }
29 }
```

Листинг 11.12. Исходный код из файла `activation/rmid.policy`

```
1 grant
2 {
3   permission com.sun.rmi.rmid.ExecPermission
4      "${java.home}${/}bin${/}java";
5   permission com.sun.rmi.rmid.ExecOptionPermission
6      "-Djava.security.policy=*";
7 };
```

java.rmi.activation.Activatable 1.2

- `protected Activatable(ActivationID id, int port)`

  Создает активизируемый объект и устанавливает приемник для указанного порта. Если в качестве параметра `port` указывается нулевое значение, порт назначается автоматически.

- `static Remote exportObject(Remote obj, ActivationID id, int port)`

  Превращает удаленный объект в активизируемый. Возвращает объект, доступный для удаленных вызывающих программ. Для того чтобы порт был назначен автоматически, нужно указать в качестве его номера нулевое значение.

- `static Remote register(ActivationDesc desc)`

  Регистрирует дескриптор для активизируемого объекта и готовит его для получения удаленных вызовов. Возвращает объект, который должен стать доступным для удаленных вызывающих программ.

java.rmi.MarshalledObject 1.2

- `MarshalledObject(Object obj)`
- Создает объект, содержащий сериализированные данные указанного объекта.
- `Object get()`
- Десериализирует хранящиеся данные объекта и возвращает сам объект.

java.rmi.activation.ActivationGroupDesc 1.2

- `ActivationGroupDesc(Properties props, ActivationGroupDesc.CommandEnvironment env)`

  Создает дескриптор группы активизации, определяющий свойства виртуальной машины, которая поддерживает активизируемые объекты. Параметр `env` указывает путь к исполняемому файлу виртуальной машины и параметры командной строки. Если же специальные установки не нужны, то указывается пустое значение `null`.

java.rmi.activation.ActivationGroup 1.2

- `static ActivationSystem getSystem()`

  Возвращает ссылку на систему активизации.

java.rmi.activation.ActivationSystem 1.2

- `ActivationGroupID registerGroup(ActivationGroupDesc group)`

  Регистрирует группу активизации и возвращает идентификатор группы.

java.rmi.activation.ActivationDesc 1.2

- `ActivationDesc(ActivationGroupID` *id*`, String` *className*`, String` *classFileURL*`, MarshalledObject` *data*`)`

  Создает дескриптор активизации.

В этой главе был рассмотрен механизм RMI, представляющий собой модель распределенных объектов программ на Java. А в следующей главе речь пойдет о другом аспекте программирования на Java: организации взаимодействия на той же самой машине с "собственным" (платформенно-ориентированным) кодом, написанном на другом языке программирования.

ГЛАВА 12

Собственные методы

В этой главе...

- Вызов функции на C из программы на Java
- Числовые параметры и возвращаемые значения
- Строковые параметры
- Доступ к полям
- Кодирование сигнатур
- Вызов методов на Java
- Доступ к элементам массивов
- Обработка ошибок
- Применение прикладного интерфейса API для вызовов
- Пример обращения к реестру Windows

Несмотря на все выгоды, которые дает решение применять только Java, возможны ситуации, когда просто необходимо создавать (или использовать) код, написанный на каком-то другом языке. Такой код обычно называется *собственным*, или *платформенно-ориентированным*.

В частности, когда язык Java только появился, многие считали, что для ускорения работы критических частей прикладных программ на Java было бы выгоднее использовать язык C или C++. Но на практике такой прием редко оказывался эффективным. Презентация Java, проведенная на конференции JavaOne в 1996 году, очень ясно это показала. Разработчики криптографической библиотеки из компании Sun Microsystems продемонстрировали, что скорость выполнения криптографических функций, реализованных только в коде Java, оказалась вполне удовлетворительной. Безусловно, она уступала скорости выполнения тех же самых функций, написанных на C, но отличие было не очень значительным. Дело в том, что реализация платформы Java оказалась

намного более быстродействующей, чем сетевой ввод-вывод. Именно это и послужило настоящей причиной более низкой производительности.

Разумеется, у перехода на собственный код имеются свои недостатки. Если часть прикладной программы написана на каком-нибудь другом языке, придется предоставить отдельную специализированную библиотеку для каждой платформы, которую планируется поддерживать. Код, который пишется на C или C++, не предусматривает защиты против некорректного обращения к памяти вследствие неправильно заданного указателя. Это означает, что можно очень легко написать такие собственные методы, которые будут нарушать работу программы или заражать операционную систему.

Таким образом, собственный код рекомендуется использовать только в тех случаях, когда это действительно необходимо. В частности, существуют три случая, когда выбор собственного кода может оказаться правильным.

- Прикладной программе требуется доступ к таким системным функциям или устройствам, которые недоступны на платформе Java.
- Имеется немалое количество проверенного или отлаженного кода на другом языке, и известны способы его переноса на все требующиеся целевые платформы.
- В результате сопоставительных испытаний обнаружено, что код, написанный на Java, выполняется много медленнее, чем эквивалентный ему код на другом языке.

В состав платформы Java входит специальный прикладной интерфейс API для организации взаимодействия с собственным кодом на C. Этот прикладной интерфейс называется JNI (Java Native Interface — собственный интерфейс Java). И в этой главе обсуждаются вопросы программирования в прикладном интерфейсе JNI.

НА ЗАМЕТКУ C++! Для написания собственных методов вместо C можно также пользоваться языком C++. Он имеет ряд преимуществ, обеспечивая, например, более строгий контроль соответствия типов и более удобный доступ к функциям JNI. Но в JNI не поддерживается взаимное преобразование классов Java и C++.

Вызов функции на C из программы на Java

Допустим, имеется некоторая функция, написанная на C, выполняющая что-нибудь полезное, но по той или иной причине ее нежелательно реализовывать заново на Java. Ради простоты предположим для начала, что это элементарная функция, написанная на C, для вывода приветственного сообщения.

В языке программирования Java для организации вызова собственного метода служит ключевое слово `native`, а сам метод, очевидно, вводится в класс. В листинге 12.1 показано, как именно это делается. Ключевое слово `native` предупреждает компилятор, что метод будет определяться внешним образом. Разумеется, собственные методы не должны содержать код, написанный на Java, поэтому объявление метода состоит только из одного заголовка, после которого сразу же следует завершающая точка с запятой. Благодаря этому объявления собственных методов становятся похожими на объявления абстрактных методов.

Листинг 12.1. Исходный код из файла `helloNative/HelloNative.java`

```
1 /**
2  * @version 1.11 2007-10-26
3  * @author Cay Horstmann
4 */
5 class HelloNative
6 {
7   public static native void greeting();
8 }
```


НА ЗАМЕТКУ! Как и в предыдущей главе, в примерах программ, приведенных в этой главе, пакеты не применяются.

В данном конкретном примере собственный метод объявляется так же, как статический (`static`). Собственные методы могут быть как статическими, так и нестатическими. Их рассмотрение было начато со статического метода, чтобы не усложнять дело передачей параметров.

Приведенный выше класс уже можно скомпилировать, но если употребить его в программе, то виртуальная машина Java уведомит, что ей неизвестно, как найти функцию `greeting()`, выдав исключение типа `UnsatisfiedLinkError`. Чтобы устранить этот недостаток, следует реализовать собственный код, т.е. написать соответствующую функцию на C. Имя этой функции должно *точно* соответствовать ожиданиям виртуальной машины Java. Для этого нужно соблюсти следующие правила.

1. Использовать полное имя метода из кода Java, т.е. `HelloNative.greeting`. Если класс находится в пакете, добавить перед именем метода имя этого пакета, например `com.horstmann.HelloNative.greeting`.
2. Заменить все точки знаками подчеркивания и присоединить префикс **`Java_`**, например `Java_HelloNative_greeting` или `Java_com_horstmann_HelloNative_greeting`.
3. Если в имени класса присутствуют символы, не являющиеся ни буквами, ни цифрами в коде ASCII (например, **`_`** или **`$`**), или же символы в уникоде свыше **`'\u007F'`**, заменить их последовательностью символов **`_0xxxx`**, указав вместо `xxxx` четырехзначное шестнадцатеричное значение заменяемого символа в уникоде.

НА ЗАМЕТКУ! Если собственные методы *перегружаются*, т.е. предоставляется несколько собственных методов с одинаковым именем, то в конце имени каждого из них следует присоединить два знака подчеркивания и закодированные типы аргументов. (Подробнее о кодировании типов аргументов речь пойдет далее в этой главе.) Так, если предоставляется один собственный метод `greeting()` и другой собственный метод `greeting(int repeat)`, первый из них должен именоваться как `Java_HelloNative_greeting__`, а другой — как `Java_HelloNative_greeting__I`.

Разумеется, вручную этого никто не делает, поскольку существует специальная утилита `javah`, способная автоматически формировать правильные имена собственных функций. Чтобы воспользоваться этой утилитой, нужно сначала скомпилировать исходный файл. Ниже показано, как это делается на примере исходного файла из листинга 12.1.

```
javac HelloNative.java
```

После этого можно вызывать утилиту `javah`, чтобы сформировать с ее помощью заголовочный файл на C из скомпилированного файла класса. Исполняемый файл утилиты `javah` находится в каталоге *jdk*`/bin`. Вызывать ее нужно по имени класса, т.е. таким же образом, как и компилятор Java. Так, в результате выполнения приведенной ниже команды будет создан заголовочный файл `HelloNative.h`, содержимое которого приведено в листинге 12.2.

```
javah HelloNative
```

Листинг 12.2. Исходный код из файла `helloNative/HelloNative.h`

```
1  /* ЭТОТ ФАЙЛ НЕ РЕДАКТИРУЕТСЯ - он формируется автоматически */
2  #include <jni.h>
3  /* Заголовочный файл для класса HelloNative */
4
5  #ifndef  Included HelloNative
6  #define _Included_HelloNative
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10 /*
11  * Класс: HelloNative
12  * Метод: greeting
13  * Сигнатура: ()V
14 */
15 JNIEXPORT void JNICALL Java_HelloNative_greeting
16   (JNIEnv *, jclass);
17
18 #ifdef __cplusplus
19 }
20 #endif
21 #endif
```

Как следует из листинга 12.2, данный файл содержит объявление функции `Java_HelloNative_greeting`. (Макрокоманды `JNIEXPORT` и `JNICALL` определяются в заголовочном файле `jni.h`. Они обозначают зависящие от компилятора спецификаторы для экспортируемых функций из динамически загружаемой библиотеки.)

Теперь остается скопировать прототип функции из заголовочного файла в файл исходного кода и написать для этой функции код реализации, как показано в листинге 12.3. Не обращайте пока что внимания на аргументы `env` и `cl` в этой простой функции. Подробнее об аргументах собственных функций и методов речь пойдет далее в этой главе.

Листинг 12.3. Исходный код из файла `helloNative/HelloNative.c`

```
1  /*
2   *@version 1.10 1997-07-01
3   * @author Cay Horstmann
4  */
5
6   #include "HelloNative.h"
7   #include <stdio.h>
```

```
8
9   JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
10 {
11   printf("Hello Native World!\n");
12 }
```

НА ЗАМЕТКУ C++! Для реализации собственных методов можно использовать и язык C++. Но в этом случае функции, реализующие эти собственные методы, следует непременно объявлять как `extern "C"`, как выделено полужирным в приведенном ниже примере кода.

```
extern "C"
JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass
cl)
{
  cout << "Hello, Native World!" << endl;
}
```

Это не позволит компилятору C++ "скорректировать" имена методов.

Далее собственный код на C требуется скомпилировать в динамически загружаемую библиотеку. Этот процесс зависит от используемого компилятора. Например, для вызова компилятора GNU C в ОС Linux соответствующая команда будет выглядеть следующим образом:

```
gcc -fPIC -I jdk/include -I jdk/include/linux -shared
    -o libHelloNative.so HelloNative.c
```

Для вызова компилятора C от компании Sun Microsystems в ОС Solaris данная команда будет выглядеть так:

```
cc -G -I jdk/include -I jdk/include/solaris -o libHelloNative.so HelloNative.c
```

А для вызова компилятора C от корпорации Microsoft в ОС Windows эта же команда примет следующий вид:

```
cl -I jdk\include -I jdk\include\win32 -LD HelloNative.c -FeHelloNative.dll
```

Здесь ***jdk*** обозначает каталог, в котором находится комплект JDK.

СОВЕТ. Если используется компилятор C от корпорации Microsoft, нужно сначала запустить из командной оболочки командный файл `vcvars32.bat` или `vsvars32.bat`. Этот командный файл автоматически установит путь и переменные окружения, требующиеся для компилятора. Найти этот файл можно в каталоге `c:\Program Files\Microsoft Visual Studio .NET 2003\Common7\tools`, `c:\Program Files\Microsoft Visual Studio 8\VC` или в каком-нибудь другом каталоге с аналогичным названием.

Кроме того, можно воспользоваться бесплатно распространяемой средой программирования Cygwin, доступной для загрузки по адресу `http://www.cygwin.com`. В ее состав входит компилятор GNU C, а также библиотеки для написания кода в стиле Unix под ОС Windows. Если используется среда Cygwin, то команда вызова компилятора C будет выглядеть следующим образом (ее нужно ввести неразрывно в одной строке):

```
gcc -mno-cygwin -D __int64="long long" -I jdk/include/
    -I jdk/include/win32 -shared -Wl,--add-stdcall-alias
    -o HelloNative.dll HelloNative.c
```

НА ЗАМЕТКУ! В версии заголовочного файла `jni_md.h` для Windows содержится следующее объявление типа, характерное для компилятора C от корпорации Microsoft:

```
typedef __int64 jlong;
```

Поэтому если используется компилятор GNU C, то может возникнуть потребность отредактировать данный файл, например, следующим образом:

```
#ifdef __GNUC__
  typedef long long jlong;
#else
  typedef __int64 jlong;
#endif
```

С другой стороны, можно указать параметр командной строки **`-D __int64="long long"`** при вызове компилятора.

И последнее, что нужно сделать, — ввести в прикладную программу вызов метода `System.loadLibrary()`. Для полной гарантии, что виртуальная машина будет загружать библиотеку перед первым использованием класса, следует организовать статический блок инициализации, как показано в листинге 12.4.

Листинг 12.4. Исходный код из файла `helloNative/HelloNativeTest.java`

```
1  /**
2   * @version 1.11 2007-10-26
3   * @author Cay Horstmann
4  */
5  class HelloNativeTest
6  {
7  public static void main(String[] args)
8    {
9      HelloNative.greeting();
10   }
11
12   static
13   {
14     System.loadLibrary("HelloNative");
15   }
16 }
```

На рис. 12.1 приведена общая схема обработки собственного кода. После компиляции и запуска рассматриваемой здесь программы на экране появится сообщение `"Hello, Native World!"` (Здравствуй, родной мир).

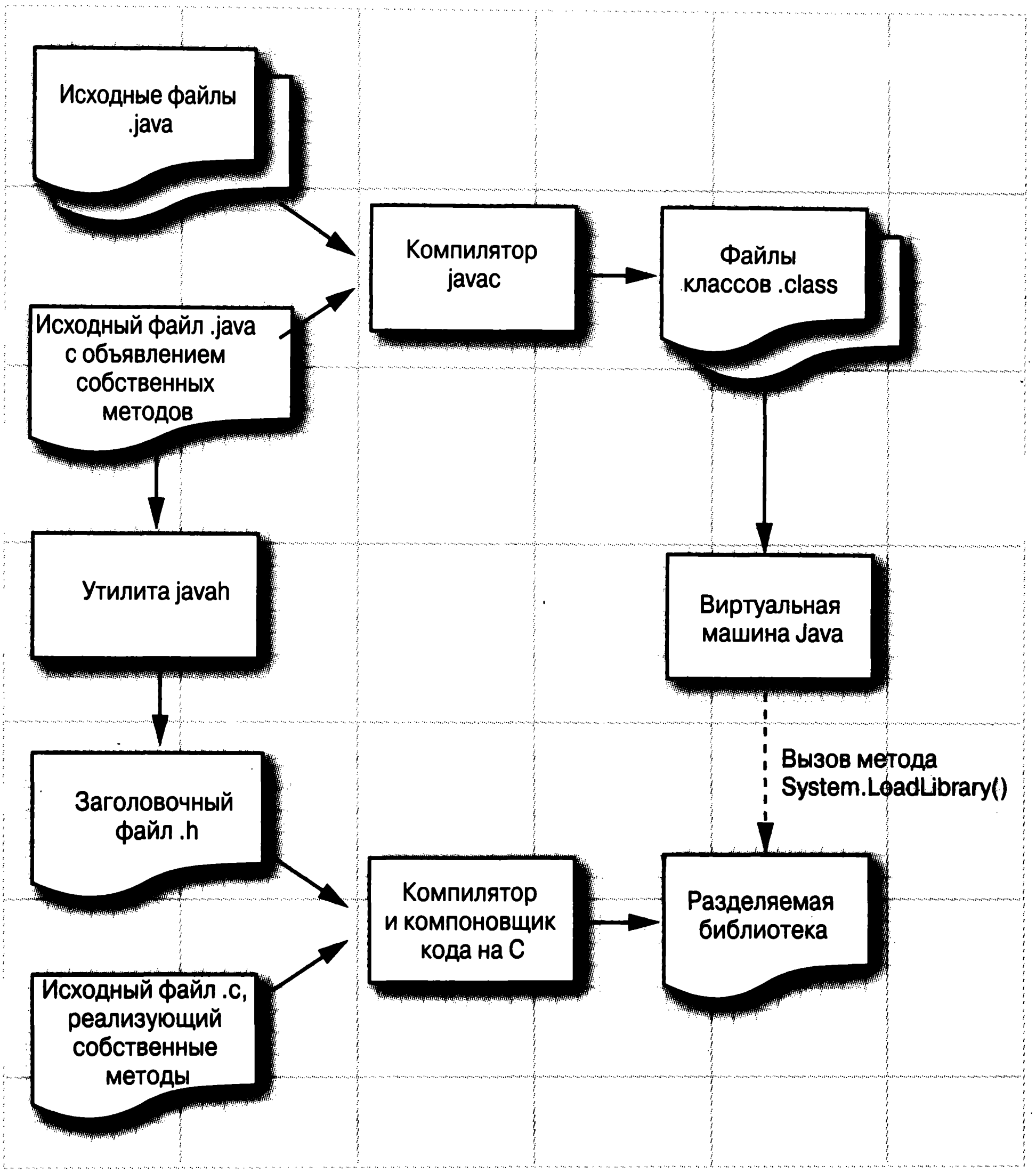


Рис. 12.1. Обработка собственного кода

НА ЗАМЕТКУ! Пользователям ОС Linux придется включить текущий каталог в путь к библиотеке. Для этого нужно установить переменную окружения `LD_LIBRARY_PATH` следующим образом:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

А с другой стороны, установить системное свойство `java.library.path`, как показано ниже.

```
java -Djava.library.path=. HelloNativeTest
```

Безусловно, результат не особенно впечатляет. Но если вспомнить, что выводимое сообщение формируется командой, написанной на C, а не на Java, то сразу становится очевидно, что это лишь первые шаги, которые удалось сделать на пути к преодолению пропасти между этими двумя языками программирования!

Таким образом, для связывания собственного метода с программой на Java необходимо выполнить следующие действия.

1. Объявить собственный метод в классе Java.
2. Запустить на выполнение утилиту `javah`, чтобы получить заголовочный файл с объявлением собственного метода на C.
3. Реализовать собственный метод на C.
4. Разместить полученный код в разделяемой библиотеке.
5. Загрузить разделяемую библиотеку в программу на Java.

java.lang.System 1.0

- `void loadLibrary(String libname)`

  Загружает библиотеку по указанному имени. Библиотека находится в том каталоге, который указан в пути поиска библиотек. Конкретный способ поиска библиотеки зависит от используемой операционной системы.

НА ЗАМЕТКУ! Некоторые разделяемые библиотеки с собственным кодом требуют выполнения кода инициализации. Любой код инициализации можно разместить в методе `JNI_OnLoad()`. А по завершении своей работы виртуальная машина Java будет аналогичным образом вызывать метод `JNI_OnUnload()`, если предоставить его. Ниже приведены прототипы этих методов.

```
jint JNI_OnLoad(JavaVM* vm, void* reserved);
void JNI_OnUnload(JavaVM* vm, void* reserved);
```

Метод `JNI_OnLoad()` должен возвратить самую раннюю версию виртуальной машины, которая требуется для нормальной работы библиотеки, например `JNI_VERSION_1_2`.

Числовые параметры и возвращаемые значения

Для обмена числовыми параметрами между кодами на C и Java следует ясно понимать, какие типы параметров соответствуют друг другу. Например, в C имеются типы данных `int` и `long`, но их реализация зависит от конкретной платформы. На одних платформах тип данных `int` представляет 16-разрядное целочисленное значение, а на других — 32-разрядное. А в Java тип данных `int` всегда представляет только 32-разрядное целочисленное значение. Именно поэтому в JNI и предоставляются такие типы данных, как `jint`, `jlong` и т.д. В табл. 12.1 приведено соответствие типов данных в Java и C.

В заголовочном файле `jni.h` эти типы данных объявляются с помощью операторов `typedef` в качестве эквивалентных типов целевой платформы. В этом же файле определяются также константы `JNI_FALSE = 0` и `JNI_TRUE = 1`.

Таблица 12.1. Типы данных в Java и C

Java	C	Количество байтов
boolean	jboolean	1
byte	jbyte	1
char	jchar	2
short	jshort	2
int	jint	4
long	jlong	8
float	jfloat	4
double	jdouble	8

Форматирование чисел с помощью функции `printf()`

До появления версии Java SE 5.0 в Java не было непосредственного аналога функции `printf()` на C. В приводимых ниже примерах предполагается, что по той или иной причине используется только старая версия JDK, и поэтому решено реализовать аналогичные функциональные возможности благодаря вызову функции `printf()` на C из собственного метода. В листинге 12.5 представлен исходный код класса `Printf1`, использующего собственный метод для вывода числового значения с плавающей точкой, указанной точностью и шириной поля.

Листинг 12.5. Исходный код из файла `printf1/Printf1.java`

```
1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4  */
5  class Printf1
6  {
7    public static native int print(int width, int precision, double x);
8
9   static
10  {
11    System.loadLibrary("Printf1");
12   }
13 }
```

Следует иметь в виду, что при реализации собственного метода на C все параметры типа `int` и `double` заменяются параметрами типа `jint` и `jdouble`, как показано в листинге 12.6.

Листинг 12.6. Исходный код из файла `printf1/Printf1.c`

```
1  /**
2   @version 1.10 1997-07-01
3   @author Cay Horstmann
4  */
5
6  #include "Printf1.h"
```

```
7  #include <stdio.h>
8
9  JNIEXPORT jint JNICALL Java_Printf1_print(JNIEnv* env, jclass cl,
10    jint width, jint precision, jdouble x)
11 {
12   char fmt[30];
13   jint ret;
14   sprintf(fmt, "%%%d.%df", width, precision);
15   ret = printf(fmt, x);
16   fflush(stdout);
17   return ret;
18 }
```

В функции из листинга 12.6 сначала компонуется форматирующая строка `"%w.pf"` с помощью переменной `fmt`, а затем вызывается стандартная функция `printf()`, после чего возвращается количество выведенных символов. В листинге 12.7 приведен исходный код примера программы, где демонстрируется применение класса `Printf1`.

Листинг 12.7. Исходный код из файла `printf1/Printf1Test.java`

```
1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4  */
5  class Printf1Test
6  {
7    public static void main(String[] args)
8    {
9      int count = Printf1.print(8, 4, 3.14);
10     count += Printf1.print(8, 4, count);
11     System.out.println();
12     for (int i = 0; i < count; i++)
13       System.out.print("-");
14     System.out.println();
15   }
16 }
```

Строковые параметры

А теперь рассмотрим, как обмениваться символьными строками с собственными методами. Как известно, в Java символьные строки представляют собой последовательности кодовых точек в кодировке UTF-16, а в C — оканчивающиеся нулевым символом последовательности байтов. Иными словами, способы представления символьных строк в этих двух языках программирования заметно отличаются. Поэтому в JNI предоставляются два ряда функций для манипулирования символьными строками: первый из них позволяет преобразовывать символьные строки Java в байтовые последовательности "модифицированного формата UTF-8", а второй — в массивы символьных значений в кодировке UTF-16, т.е. в массивы типа `jchar`. (Подробнее о кодировках UTF-8 и UTF-16, а также о "модифицированном формате UTF-8" см. в главе 1. Напомним, что в кодировке UTF-8 и "модифицированном формате UTF-8" символы кода ASCII оставляются без изменений, а все остальные символы уникода кодируются в виде многобайтовых последовательностей.)

НА ЗАМЕТКУ! Стандартный и модифицированный форматы UTF-8 отличаются только дополнительными символами с кодами свыше `0xFFFF`. В стандартной кодировке UTF-8 эти символы кодируются 4-байтовыми последовательностями. А в модифицированном формате символ сначала представляется так называемой "суррогатной парой" в кодировке UTF-16, а затем каждый суррогат кодируется в кодировке UTF-8. В итоге получается последовательность из 6 байт. Такое решение нельзя назвать изящным, но другого выхода нет, ведь спецификация виртуальной машины Java была написана в те времена, когда длина символа в уникоде ограничивалась 16 битами.

Если в прикладном коде на C уже применяется уникод, то лучше всего воспользоваться вторым рядом функций преобразования символьных строк. Если же все символьные строки содержат только символы в коде ASCII, то можно воспользоваться функциями преобразования символьных строк в "модифицированный формат UTF-8".

Собственный метод со строковым параметром типа `String` фактически получает значение типа `jstring`. А собственный метод с возвращаемым строковым значением типа `String` должен непременно возвращать значение типа `jstring`. Для чтения и создания объектов типа `jstring` применяются специальные функции JNI. Например, функция `NewStringUTF()` создает новый объект типа `jstring` из массива `char`, содержащего символы в коде ASCII или, что бывает намного чаще, байтовые последовательности, кодированные в "модифицированном формате UTF-8". Функции JNI вызываются не совсем обычно. Например, вызов функции `NewStringUTF()` выглядит следующим образом:

```
JNIEXPORT jstring JNICALL Java_HelloNative_getGreeting(JNIEnv* env, jclass cl)
{
  jstring jstr;
  char greeting[] = "Hello, Native World\n";
  jstr = (*env)->NewStringUTF(env, greeting);
  return jstr;
}
```

НА ЗАМЕТКУ! Весь код здесь и далее в этой главе приводится на C, если не указано иное.\

Во всех вызовах функций JNI используется указатель `env`, который является первым параметром каждого собственного метода. Он ссылается на таблицу с указателями функций (рис. 12.2). Вследствие этого каждый вызов функции JNI следует предварять префиксом **`(*env)->`** для разыменования конкретного указателя функции. Более того, указатель `env` является также первым параметром каждой функции JNI.

НА ЗАМЕТКУ C++! В C++ доступ к функциям JNI организуется проще. В версии класса `JNIEnv` на C++ имеются встраиваемые функции-члены, способные автоматически выполнять поиск указателей функций. Например, вызвать функцию `NewStringUTF()` в C++ можно следующим образом:

```
jstr = env->NewStringUTF(greeting);
```

Обратите внимание на отсутствие указателя `JNIEnv` в списке параметров вызываемого метода.

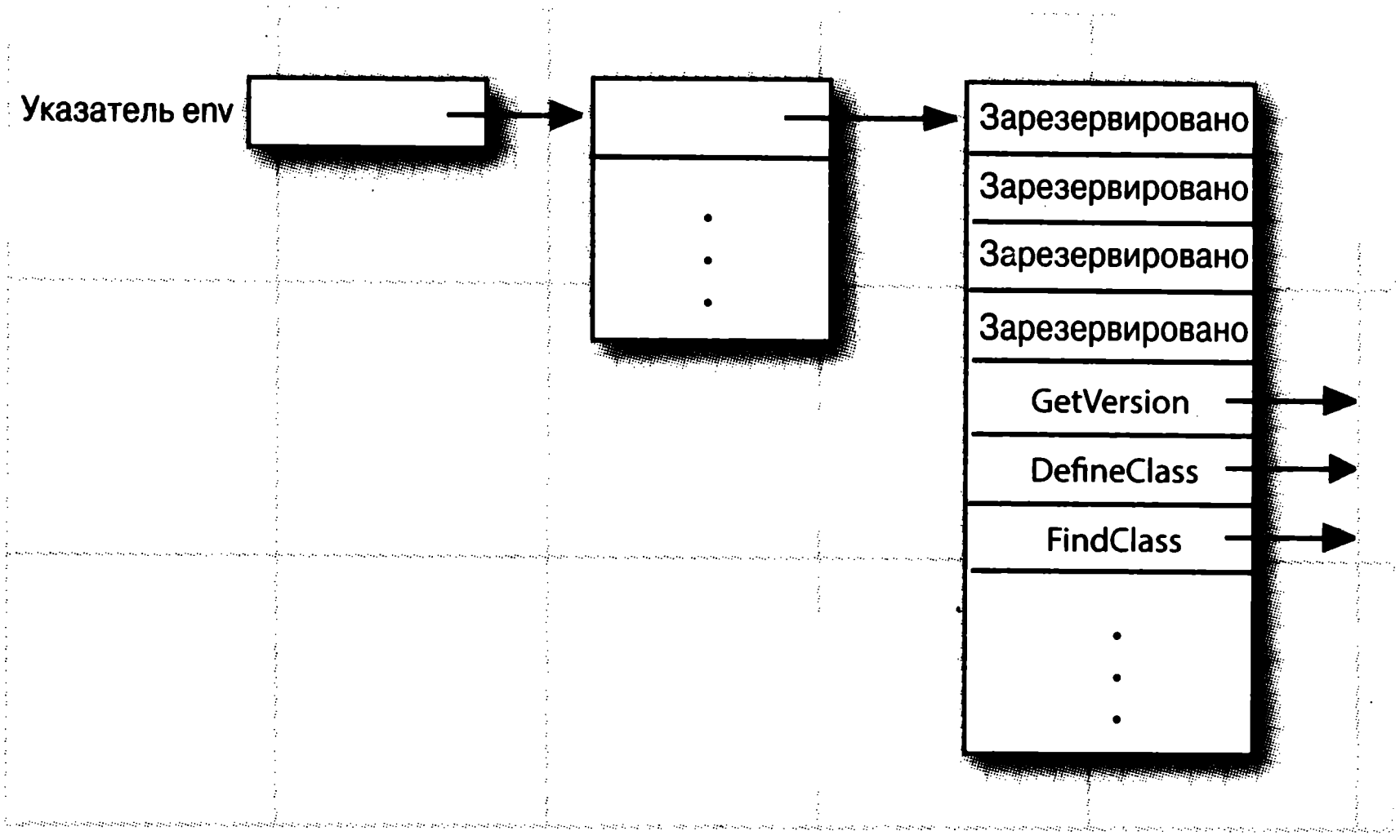


Рис. 12.2. Указатель `env`

Функция `NewStringUTF()` позволяет создавать новый объект типа `jstring`. Для считывания содержимого уже существующего объекта типа `jstring` применяется функция `GetStringUTFChars()`. Эта функция возвращает указатель `const jbyte*` на описывающие строку символы в модифицированном формате UTF-8. Следует иметь в виду, что некоторые виртуальные машины Java могут использовать данный формат для внутреннего представления символьных строк, из-за чего возможно получение указателя на конкретную символьную строку в Java. А поскольку символьные строки в Java считаются неизменяемыми, то очень важно со *всей* серьезностью отнестись к обозначению `const` и не пытаться ничего записывать в такой символьный массив. С другой стороны, если виртуальная машина применяет кодировку UTF-16 или UTF-32 для внутреннего представления строк, то вызов функции `GetStringUTFChars()` будет приводить к выделению нового блока памяти и его заполнению эквивалентными символами в модифицированном формате UTF-8.

По завершении манипулирования символьной строкой следует вызвать функцию `ReleaseStringUTFChars()`, чтобы сообщить об этом виртуальной машине, которая должна освободить память, занимаемую данной строкой. Как известно, процесс "сборки мусора" происходит в отдельном потоке исполнения и может прерывать выполнение собственных методов. Именно поэтому требуется вызвать функцию `ReleaseStringUTFChars()`.

С другой стороны, можно предоставить свой буфер для хранения символов строки, вызвав функции `GetStringRegion()` или `GetStringUTFRegion()`. И наконец, функция `GetStringUTFLength()` возвращает количество символов, необходимых для представления строки в модифицированном формате UTF-8.

НА ЗАМЕТКУ! Описание прикладного интерфейса JNI API можно найти по адресу `http://docs.oracle.com/javase/7/docs/technotes/guides/jni`.

Доступ к символьным строкам в Java из кода на C

- `jstring NewStringUTF(JNIEnv* env, const char bytes[])`

  Возвращает новый объект символьной строки в Java из оканчивающейся нулевым байтом последовательности байтов в модифицированном формате UTF-8, или значение `NULL`, если сконструировать символьную строку не удается.

- `jsize GetStringUTFLength(JNIEnv* env, jstring string)`

  Возвращает количество байтов, требующееся для кодирования символьной строки в модифицированном формате UTF-8 (без учета завершающего нулевого байта).

- `const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean* isCopy)`

  Возвращает указатель на символьную строку в модифицированном формате UTF-8 или значение `NULL`, если создать символьный массив в таком формате не удается. Этот указатель остается действительным до тех пор, пока не будет вызвана функция `ReleaseStringUTFChars()`. Параметр *isCopy* принимает значение `NULL` или указатель на переменную типа *jboolean*, которая заполняется значением `JNI_TRUE`, если создается копия символьной строки, а иначе — значение `JNI_FALSE`.

- `void ReleaseStringUTFChars(JNIEnv* env, jstring string, const jbyte bytes[])`

  Извещает виртуальную машину, что из собственного кода больше не требуется доступ к символьной строке Java через массив *bytes* по указателю, возвращаемому функцией `GetStringUTFChars()`.

- `void GetStringRegion(JNIEnv* env, jstring string, jsize start, jsize length, jchar* buffer)`

  Копирует последовательность двойных байтов в кодировке UTF-16 из символьной строки в предоставляемый пользователем буфер размером, как минимум, **2×*length***.

- `void GetStringUTFRegion(JNIEnv* env, jstring string, jsize start, jsize length, jbyte* buffer)`

  Копирует последовательность байтов в модифицированном формате UTF-8 из символьной строки в буфер, предоставляемый пользователем. Емкости буфера должно хватить, чтобы вместить все байты. В худшем случае копируется только **3×*length*** байтов.

- `jstring NewString(JNIEnv* env, const jchar chars[], jsize length)`

  Возвращает новый объект символьной строки Java из символьной строки в уникоде или значение `NULL`, если создать символьную строку Java не удается.

Параметры:		
	env	Указатель прикладного интерфейса JNI
	chars	Символьная строка в кодировке UTF-16, оканчивающаяся нулевым символом
	length	Количество символов в строке

- `jsize GetStringLength(JNIEnv *env, jstring string)`
  Возвращает количество символов в строке.
- `const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean* isCopy)`
  Возвращает указатель на последовательность символов в уникоде или значение `NULL`, если создать символьный массив в таком формате не удается. Указатель действует до тех пор, пока не будет вызвана функция `ReleaseStringUTFChars()`. Параметр *isCopy* принимает значение `NULL` или указатель на переменную типа *jboolean*, которая заполняется значением `JNI_TRUE`, если создается копия символьной строки, а иначе — значением `JNI_FALSE`.
- `void ReleaseStringChars(JNIEnv* env, jstring string, const jchar chars[])`
  Извещает виртуальную машину, что из собственного кода больше не требуется доступ к символьной строке Java через массив *chars* по указателю, возвращаемому функцией `GetStringChars()`.

А теперь опробуем упомянутые выше функции на практике, разработав класс, способный вызывать функцию `sprintf()` на C. Желательно, чтобы эта функция вызывалась так, как показано в листинге 12.8.

Листинг 12.8. Исходный код из файла `printf2/Printf2Test.java`

```
1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4  */
5  class Printf2Test
6  {
7    public static void main(String[] args)
8    {
9      double price = 44.95;
10     double tax = 7.75;
11     double amountDue = price * (1 + tax / 100);
12
13     String s = Printf2.sprint("Amount due = %8.2f", amountDue);
14     System.out.println(s);
15   }
16 }
```

В листинге 12.9 представлен исходный код класса с собственным методом `sprint()`.

Листинг 12.9. Исходный код из файла `printf2/Printf2.java`

```
1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4  */
5  class Printf2
6  {
7    public static native String sprint(String format, double x);
8
```

```
9     static
10    {
11      System.loadLibrary("Printf2");
12    }
13 }
```

Таким образом, функция на C, с помощью которой форматируется число с плавающей точкой, имеет следующий прототип:

```
JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* env, jclass cl,
    jstring format, jdouble x)
```

Код, реализующий эту функцию на C, приведен в листинге 12.10. Следует иметь в виду, что для считывания параметра форматирования `format` вызывается функция `GetStringUTFChars()`, для генерирования возвращаемого значения — функция `NewStringUTF()`, а для извещения виртуальной машины, что доступ к символьной строке больше не требуется, — функция `ReleaseStringUTFChars()`.

Листинг 12.10. Исходный код из файла `printf2/Printf2.c`

```
1  /**
2   @version 1.10 1997-07-01
3   @author Cay Horstmann
4  */
5
6  #include "Printf2.h"
7  #include <string.h>
8  #include <stdlib.h>
9  #include <float.h>
10
11  /**
12   * @param format Символьная строка со спецификатором формата для
13   * функции printf(), например, "%8.2f". Подстроки "%%" пропускаются
14   * @return Возвращает указатель на спецификатор формата, пропуская
15   * ·       символ '%', или значение NULL в отсутствие однозначного
16   *         спецификатора формата
17  */
18  char* find_format(const char format[])
19  {
20    char* p;
21    char* q;
22
23    p = strchr(format, '%');
24    while (p != NULL && *(p + 1) == '%') /* пропустить подстроку "%%" */
25      p = strchr(p + 2, '%');
26    if (p == NULL) return NULL;
27    /* проверить единственность символа "%" */
28    p++;
29    q = strchr(p, '%');
30    while (q != NULL && *(q + 1) == '%') /* пропустить подстроку "%%" */
31      q = strchr(q + 2, '%');
32    if (q != NULL) return NULL; /* символ "%" не единственный */
33    q = p + strspn(p, " -0+#"); /* пропустить признаки */
34    q += strspn(q, "0123456789"); /* пропустить ширину поля */
35    if (*q == '.') { q++; q += strspn(q, "0123456789"); }
36      /* пропустить точность */
37    if (strchr("eEfFgG", *q) == NULL) return NULL;
38      /* это не формат с плавающей точкой */
```

```
39    return p;
40  }
41
42  JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* env, jclass cl,
43     jstring format, jdouble x)
44  {
45    const char* cformat;
46    char* fmt;
47    jstring ret;
48
49    cformat = (*env)->GetStringUTFChars(env, format, NULL);
50    fmt = find_format(cformat);
51    if (fmt == NULL)
52      ret = format;
53    else
54    {
55      char* cret;
56      int width = atoi(fmt);
57      if (width == 0) width = DBL_DIG + 10;
58      cret = (char*) malloc(strlen(cformat) + width);
59      sprintf(cret, cformat, x);
60      ret = (*env)->NewStringUTF(env, cret);
61      free(cret);
62    }
63    (*env)->ReleaseStringUTFChars(env, format, cformat);
64    return ret;
65  }
```

В данной функции сохранен простой механизм обработки ошибок. Если код форматирования, отвечающий за вывод числового значения с плавающей точкой, не соответствует форме **%w.pc** (где **c** — один из символов `e`, `E`, `f`, `g`, `G`), тогда форматирование числового значения просто не выполняется. Далее в этой главе будет показано, как организовать в собственном методе генерирование исключения.

Доступ к полям

Все рассматривавшиеся до сих пор собственные методы были статическими с числовыми и строковыми параметрами. А теперь речь пойдет о методах, способных обращаться с объектами произвольного типа. Для примера попробуем реализовать как собственный метод из класса `Employee`, упоминавшийся в главе 4 первого тома данной книги. И хотя это несколько отвлеченный пример, тем не менее, он наглядно показывает, как получается доступ к полям из собственного метода.

Доступ к полям экземпляра

Чтобы выяснить, как получить доступ к полям экземпляра из собственного метода, реализуем снова метод `raiseSalary()`. Исходный код этого метода на Java выглядит следующим образом:

```
public void raiseSalary(double byPercent)
{
  salary *= 1 + byPercent / 100;
}
```

Перепишем этот метод как собственный. В отличие от собственных методов из предыдущих примеров, данный метод не является статическим. Поэтому в результате выполнения утилиты `javah` получается следующий прототип данного метода:

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv *, jobject, jdouble);
```

Обратите внимание на второй параметр, который теперь имеет тип `jobject`, а не `jclass`. По существу, он является эквивалентом ссылки `this`. Статические методы получают ссылку на класс, а нестатические — на конкретный объект.

Итак, нам требуется доступ к полю `salary` объекта, представленного неявным параметром. В первоначальном варианте связывания кода Java и C в версии Java 1.0 подобная задача решалась очень просто. Программисты могли получить прямой доступ к полям данных объекта. Но для прямого доступа всем виртуальным машинам придется раскрыть внутреннюю структуру данных. Поэтому для получения и установки значений в полях программистам приходится обращаться к специальным функциям JNI.

В рассматриваемом здесь примере поле `salary` относится к типу `double`, поэтому для доступа к нему воспользуемся функциями `GetDoubleField()` и `SetDoubleField()`. Для доступа к полям данных других типов предусмотрены функции `GetIntField()` и `SetIntField()`, `GetObjectField()` и `SetObjectField()` и т.д. При обращении с ними применяется следующий синтаксис:

```
x = (*env)->GetXxxField(env, this_obj, fieldID);
(*env)->SetXxxField(env, this_obj, fieldID, x);
```

где параметр `fieldID` представляет значение специального типа, которое обозначает поле структуры, а суффикс *Xxx* — тип данных Java (например, `Object`, `Boolean`, `Byte` и т.д.). Для того чтобы получить значение `fieldID`, нужно сначала получить значение, представляющее класс, используя функцию `GetObjectClass()` или `FindClass()`. В частности, функция `GetObjectClass()` возвращает класс произвольного объекта, как показано ниже.

```
jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
```

А функция `FindClass()` позволяет указать имя класса в виде символьной строки (вместо точки для разделения пакетов следует использовать знак /), как выделено ниже полужирным.

```
jclass class_String = (*env)->FindClass(env, "java/lang/String");
```

Для получения параметра `fieldID` служит функция `GetFieldID()`, при вызове которой указывается *имя* поля и его *сигнатура*, т.е. кодированное представление его типа. Например, в приведенной ниже строке кода получается идентификатор поля `salary`.

```
jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");
```

Символьная строка `"D"` обозначает здесь тип `double`. Более подробно правила кодирования сигнатур полей рассматриваются в следующем разделе.

На первый взгляд, такой порядок доступа к полям данных может показаться слишком сложным. Но не следует забывать, что разработчики прикладного интерфейса JNI не стремились раскрывать поля данных напрямую, и поэтому им пришлось предоставить функции для установки и получения значений полей. Для сокращения издержек, связанных с применением этих функций, вычисление идентификатора из имени поля выделено в отдельную задачу, поскольку она требует наибольших издержек вычислительных ресурсов. Следовательно, при неоднократном обращении к полю с целью установить или получить его значение идентификатор этого поля вычисляется только один раз.

Итак, принимая во внимание все сказанное выше, реализуем метод `raiseSalary()` как собственный. Ниже приведен исходный код этого метода, написанный на C.

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env,
    jobject this_obj, jdouble byPercent)
{
  /* получить класс */
  jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
  /* получить идентификатор поля */
  jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");
  /* получить значение поля */
  jdouble salary = (*env)->GetDoubleField(env, this_obj, id_salary);
  salary *= 1 + byPercent / 100;
  /* установить значение поля */
  (*env)->SetDoubleField(env, this_obj, id_salary, salary);
}
```

ВНИМАНИЕ! Ссылки на классы действительны только до завершения собственного метода. Поэтому кешировать значения, возвращаемые функцией `GetObjectClass()`, не удастся. Не пытайтесь сберечь ссылку на класс для последующего использования при вызове собственного метода. Всякий раз, когда используется собственный метод, нужно вызывать функцию `GetObjectClass()`. Если же это неприемлемо, ссылку можно зафиксировать с помощью функции `NewGlobalRef()`, например, следующим образом:

```
static jclass class_X = 0;
static jfieldID id_a;
. . .
if (class_X == 0)
{
  jclass cx = (*env)->GetObjectClass(env, obj);
  class_X = (*env)->NewGlobalRef(env, cx);
  id_a = (*env)->GetFieldID(env, cls, "a", ". . .");
}
```

Теперь зафиксированную ссылку на класс и идентификаторы полей можно использовать при последующих вызовах собственного метода. По завершении всех необходимых операций с классом, доступным по данной ссылке, ее следует удалить с помощью функции `DeleteGlobalRef()`:

```
(*env)->DeleteGlobalRef(env, class_X);
```

В листинге 12.11 приведен исходный код примера программы на Java, а в листинге 12.12 — исходный код класса `Employee`. Что же касается исходного кода собственного метода `raiseSalary()` на C, то он представлен в листинге 12.13.

Листинг 12.11. Исходный код из файла `employee/EmployeeTest.java`

```
1  /**
2   * @version 1.10 1999-11-13
3   * @author Cay Horstmann
4  */
5
6  public class EmployeeTest
7  {
8    public static void main(String[] args)
9    {
```

```
10      Employee[] staff = new Employee[3];
11
12      staff[0] = new Employee("Harry Hacker", 35000);
13      staff[1] = new Employee("Carl Cracker", 75000);
14      staff[2] = new Employee("Tony Tester", 38000);
15
16      for (Employee e : staff)
17        e.raiseSalary(5);
18      for (Employee e : staff)
19        e.print();
20    }
21 }
```

Листинг 12.12. Исходный код из файла `employee/Employee.java`

```
1  /**
2   * @version 1.10 1999-11-13
3   * @author Cay Horstmann
4  */
5
6  public class Employee
7  {
8    private String name;
9    private double salary;
10
11   public native void raiseSalary(double byPercent);
12
13   public Employee(String n, double s)
14   {
15     name = n;
16     salary = s;
17   }
18
19   public void print()
20   {
21     System.out.println(name + " " + salary);
22   }
23
24   static
25   {
26     System.loadLibrary("Employee");
28   }
29 }
```

Листинг 12.13. Исходный код из файла `employee/Employee.c`

```
1  /**
2   * @version 1.10 1999-11-13
3   * @author Cay Horstmann
4  */
5
6  #include "Employee.h"
7
8  #include <stdio.h>
9
10 JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env,
```

```
11    jobject this_obj, jdouble byPercent)
12 {
13    /* получить класс */
14    jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
15
16    /* получить идентификатор поля */
17    jfieldID id_salary = (*env)->GetFieldID(
18       env, class_Employee, "salary", "D");
19
20    /* получить значение поля */
21    jdouble salary = (*env)->GetDoubleField(env, this_obj, id_salary);
22
23    salary *= 1 + byPercent / 100;
24
25    /* установить значение поля */
26    (*env)->SetDoubleField(env, this_obj, id_salary, salary);
27 }
```

Доступ к статическим полям

Доступ к статическим полям осуществляется аналогично доступу к нестатическим полям. Для этой цели служат функции `GetStaticFieldID()` и `GetStatic`*`Xxx`*`Field()`/`SetStatic`*`Xxx`*`Field()`, которые действуют практически так же, как и их нестатические аналоги, но имеют два отличия.

- Вследствие того что объект отсутствует, для получения ссылки на класс следует вызвать функцию `FindClass()` вместо функции `GetObjectClass()`.
- Для доступа к статическому полю следует предоставить класс, а не объект.

В качестве примера ниже показано, как получить ссылку на стандартный поток вывода `System.out`.

```
/* получить класс */
jclass class_System = (*env)->FindClass(env, "java/lang/System");

/* получить идентификатор поля */
jfieldID id_out = (*env)->GetStaticFieldID(env, class_System, "out",
   "Ljava/io/PrintStream;");

/* получить значение поля */
jobject obj_out = (*env)->GetStaticObjectField(env, class_System, id_out);
```

Доступ к полям

- `jfieldID GetFieldID(JNIEnv* `*`env`*`, jclass `*`cl`*`, const char `*`name[]`*`, const char `*`fieldSignature[]`*`)`

  Возвращает идентификатор поля в классе.

- `Xxx GetXxxField(JNIEnv* `*`env`*`, jobject `*`obj`*`, jfieldID `*`id`*`)`

  Возвращает значение поля. В качестве типа *Xxx* поля может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.

- `void SetXxxField(JNIEnv* env, jobject obj, jfieldID id, Xxx value)`

  Устанавливает новое значение в поле. В качестве типа *Xxx* поля может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.

- `jfieldID GetStaticFieldID(JNIEnv* env, jclass cl, const char name[], const char fieldSignature[])`

  Возвращает идентификатор статического поля в классе.

- *Xxx* `GetStaticXxxField(JNIEnv* env, jclass cl, jfieldID id)`

  Возвращает значение статического поля. В качестве типа *Xxx* поля может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.

- `void SetStaticXxxField(JNIEnv* env, jclass cl, jfieldID id, Xxx value)`

  Устанавливает новое значение в статическом поле. В качестве типа *Xxx* поля может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.

Кодирование сигнатур

Для доступа к полям экземпляра и вызова методов, определенных в Java, необходимо знать правила корректирования имен типов данных и сигнатур методов. (Напомним, что сигнатура метода описывает параметры и возвращаемое значение.) Для этой цели используется приведенная ниже схема кодирования.

`B`	`byte`
`C`	`char`
`D`	`double`
`F`	`float`
`I`	`int`
`J`	`long`
`Lимя_класса;`	тип класса
`S`	`short`
`V`	`void`
`Z`	`boolean`

Для описания типа массива служит знак `[`. Например, массив символьных строк описывается следующим образом:

```
[Ljava/lang/String;
```

А двумерный массив `float[][]` кодируется так:

```
[[F
```

Для полного обозначения сигнатуры метода сначала перечисляются все типы параметров в круглых скобках, а затем указывается возвращаемый тип. Например, сигнатура метода, получающего два целочисленных значения и возвращающего одно целочисленное значение, кодируется следующим образом:

```
(II)I
```

Метод `print()`, использовавшийся в предыдущем примере, имеет приведенную ниже скорректированную сигнатуру. Эта сигнатура означает, что данный метод получает символьную строку, но ничего не возвращает, поскольку возвращаемое значение имеет тип `void`.

```
(Ljava/lang/String;)V
```

Обратите внимание на то, что точка с запятой в конце выражения L*имя_класса*; служит признаком окончания определяющего тип выражения, а не разделителем параметров. Например, следующий конструктор:

```
Employee(java.lang.String, double, java.util.Date)
```

имеет такую сигнатуру:

```
"(Ljava/lang/String;DLjava/util/Date;)V"
```

Обратите внимание на отсутствие разделителя между обозначениями `D` и `Ljava/util/Date;`, а также на то, что в данной схеме кодирования для разделения имен пакетов и классов требуется указывать знак косой черты (/), а не точки (.). Обозначение **V** в конце сигнатуры означает, что данный конструктор ничего не возвращает, поскольку возвращаемое значение имеет тип `void`. И хотя возвращаемый тип для конструкторов в Java не указывается, обозначение **V** все равно должно добавляться в конце сигнатуры специально для виртуальной машины.

СОВЕТ. Для автоматического генерирования сигнатур методов из файлов классов достаточно выполнить команду `javap` с параметром командной строки **-s**, например, следующим образом:

```
javap -s -private Employee
```

В итоге на экран будут выведены сигнатуры всех полей и методов указанного класса, как показано ниже.

```
Compiled from "Employee.java"
public class Employee extends java.lang.Object{
private java.lang.String name;
  Signature: Ljava/lang/String;
private double salary;
  Signature: D
public Employee(java.lang.String, double);
  Signature: (Ljava/lang/String;D)V
public native void raiseSalary(double);
  Signature: (D)V
public void print();
  Signature: ()V
static {};
  Signature: ()V
}
```

НА ЗАМЕТКУ! Не существует никакого рационального объяснения тому, что программисты вынуждены применять именно такую скорректированную схему кодирования сигнатур. Разработчики механизма вызова собственного кода с таким же успехом могли бы написать функцию, позволяющую сначала считывать сигнатуры в стиле языка программирования Java, например `void(int, java.lang.String)`, а затем кодировать их в любое удобное внутреннее представление. Но, с другой стороны, применение скорректированных сигнатур позволяет приобщиться к таинствам программирования на уровне, близком к виртуальной машине.

Вызов методов на Java

Итак, вы знаете, как вызывать функции, написанные на C, из прикладного кода, написанного на Java. Именно для этого и служат собственные методы. А можно ли выполнить обратную операцию, вызвав методы на Java из функций на C, и зачем вообще такая операция может потребоваться? Оказывается, что нередко из собственного метода требуется запрашивать какую-нибудь службу из переданного ему объекта. Поэтому далее будет показано, как добиться этого сначала для методов экземпляра, а затем для статических методов.

Методы экземпляра

В качестве примера вызова метода экземпляра на Java из собственного кода усовершенствуем класс `Printf`, введя в него метод, действующий как функция `fprintf()` на C. Следовательно, этот метод должен быть в состоянии выводить отформатированную строку в произвольный поток в виде объекта типа `PrintWriter`. Ниже показано, каким образом этот метод определяется на Java.

```
class Printf3
{
  public native static void fprint(PrintWriter out, String s, double x);
  . . .
}
```

Сначала выводимая символьная строка составляется в объекте `str` типа `String`, как и в реализованном ранее методе `sprint()`. Затем из функции на C вызывается метод `print()` из класса `PrintWriter`. Любой метод на Java можно вызвать из функции на C в следующей форме:

```
(*env)->CallXxxMethod(env, неявный_параметр, идентификатор_метода,
    явные_параметры)
```

где *Xxx* обозначает возвращаемый тип, например `Void`, `Int`, `Obect` и т.д. Если для доступа к полю требуется его идентификатор, то для вызова метода следует указать его идентификатор. Для получения идентификатора вызываемого метода в прикладном интерфейсе JNI предусмотрена функция `GetMethodID()` с параметрами, которые задают класс, имя и сигнатуру метода.

В рассматриваемом здесь примере требуется получить идентификатор метода `print()` из класса `PrintWriter`. Как упоминалось в главе 12 первого тома данной книги, в классе `PrintWriter` имеются девять разных методов под одинаковым именем `print`. Поэтому для выбора конкретного варианта следует точно указать параметры и возвращаемое значение. Например, для вызова метода `void print(java.lang.String)` следует закодировать его скорректированную сигнатуру в символьной строке `"(Ljava/lang/String;)V"`, как пояснялось в предыдущем разделе.

Ниже приведен весь код, который выполняет следующие действия для вызова этого метода.

1. Получение класса из неявного параметра.
2. Получение идентификатора метода.
3. Вызов метода.

```
/* получить класс */
class_PrintWriter = (*env)->GetObjectClass(env, out);
```

```
/* получить идентификатор метода */
id_print = (*env)->GetMethodID(env, class_PrintWriter, "print",
   "(Ljava/lang/String;)V");

/* вызвать метод */
(*env)->CallVoidMethod(env, out, id_print, str);
```

В листинге 12.14 приведен исходный код примера программы на Java, а в листинге 12.15 — исходный код класса `Printf3`. Что же касается собственного метода `fprint()`, то его исходный код на C представлен в листинге 12.16.

НА ЗАМЕТКУ! Числовые идентификаторы методов и полей выполняют те же функции, что и объекты типа `Method` и `Field` из прикладного интерфейса API для рефлексии в Java. Для их взаимного преобразования служат следующие функции:

```
jobject ToReflectedMethod(JNIEnv* env, jclass class, jmethodID methodID);
  // возвращает объект типа Method
methodID FromReflectedMethod(JNIEnv* env, jobject method);
jobject ToReflectedField(JNIEnv* env, jclass class, jfieldID fieldID);
  // возвращает объект типа Field
fieldID FromReflectedField(JNIEnv* env, jobject field);
```

Листинг 12.14. Исходный код из файла `printf3/Printf3Test.java`

```
1  import java.io.*;
2  /**
3   * @version 1.10 1997-07-01
4   * @author Cay Horstmann
5  */
6  class Printf3Test
7  {
8    public static void main(String[] args)
9    {
10     double price = 44.95;
11     double tax = 7.75;
12     double amountDue = price * (1 + tax / 100);
13     PrintWriter out = new PrintWriter(System.out);
14     Printf3.fprint(out, "Amount due = %8.2f\n", amountDue);
15     out.flush();
16   }
17 }
```

Листинг 12.15. Исходный код из файла `printf3/Printf3.java`

```
1  import java.io.*;
2  /**
3   * @version 1.10 1997-07-01
4   * @author Cay Horstmann
5  */
6  class Printf3
7  {
8    public static native void fprint(PrintWriter out,
9       String format, double x);
10
11   static
```

```
12   {
13     System.loadLibrary("Printf3");
14   }
15 }
```

Листинг 12.16. Исходный код из файла `printf3/Printf3.c`

```
1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4  */
5  #include "Printf3.h"
6  #include <string.h>
7  #include <stdlib.h>
8  #include <float.h>
9  /**
10  * @param format Символьная строка со спецификатором формата для
11  * функции printf(), например "%8.2f". Подстроки "%%" пропускаются
12  * @return Возвращает указатель на спецификатор формата, пропуская
13  *         символ '%', или значение NULL в отсутствие однозначного
14  *         спецификатора формата
15 */
16 char* find_format(const char format[])
17 {
18   char* p;
19   char* q;
20
21   p = strchr(format, '%');
22   while (p != NULL && *(p + 1) == '%') /* пропустить подстроку "%%" */
23     p = strchr(p + 2, '%');
24   if (p == NULL) return NULL;
25   /* проверить единственность символа "%" */
26   p++;
27   q = strchr(p, '%');
28   while (q != NULL && *(q + 1) == '%') /* пропустить подстроку "%%" */
29     q = strchr(q + 2, '%');
30   if (q != NULL) return NULL; /* символ "%" не единственный */
31   q = p + strspn(p, " -0+#"); /* пропустить признаки */
32   q += strspn(q, "0123456789"); /* пропустить ширину поля */
33   if (*q == '.') { q++; q += strspn(q, "0123456789"); }
34     /* пропустить точность */
35   if (strchr("eEfFgG", *q) == NULL) return NULL;
36     /* это не формат с плавающей точкой */
37   return p;
38 }
39
40 JNIEXPORT void JNICALL Java_Printf3_fprint(JNIEnv* env, jclass cl,
41    jobject out, jstring format, jdouble x)
42 {
43   const char* cformat;
44   char* fmt;
45   jstring str;
46   jclass class_PrintWriter;
47   jmethodID id_print;
48   cformat = (*env)->GetStringUTFChars(env, format, NULL);
49   fmt = find_format(cformat);
50   if (fmt == NULL)
51     str = format;
52   else
```

```
53   {
54     char* cstr;
55     int width = atoi(fmt);
56     if (width == 0) width = DBL_DIG + 10;
57     cstr = (char*) malloc(strlen(cformat) + width);
58     sprintf(cstr, cformat, x);
59     str = (*env)->NewStringUTF(env, cstr);
60     free(cstr);
61   }
62   (*env)->ReleaseStringUTFChars(env, format, cformat);
63
64   /* вызвать метод ps.print(str) */
65
66   /* получить класс */
67   class_PrintWriter = (*env)->GetObjectClass(env, out);
68
69   /* получить идентификатор метода */
70   id_print = (*env)->GetMethodID(env, class_PrintWriter, "print",
71       "(Ljava/lang/String;)V");
72
73   /* вызвать метод */
74   (*env)->CallVoidMethod(env, out, id_print, str);
75 }
```

Статические методы

Вызов статических методов из собственного кода выполняется аналогично вызову методов экземпляра, кроме двух отличий.

- Для этой цели вызываются функции `GetStaticMethodID()` и `CallStatic`*Xxx*`Method()`.
- При вызове статического метода вместо неявного параметра указывается объект соответствующего класса.

В качестве примера рассмотрим следующий вызов статического метода `getProperty()` из собственного метода:

```
System.getProperty("java.class.path")
```

Из этого метода возвращается символьная строка, содержащая текущий путь к классу. Сначала нужно обнаружить используемый класс. В отсутствие объектов класса `System` придется вызвать функцию `FindClass()` вместо функции `GetObjectClass()`, как показано ниже.

```
jclass class_System = (*env)->FindClass(env, "java/lang/System");
```

Затем нужно получить идентификатор статического метода `getProperty()`. Параметр и возвращаемое значение этого метода представлены символьной строкой, поэтому его сигнатура кодируется следующим образом:

```
"(Ljava/lang/String;)Ljava/lang/String;"
```

Далее следует получить идентификатор данного метода:

```
jmethodID id_getProperty = (*env)->GetStaticMethodID(env, class_System,
    "getProperty", "(Ljava/lang/String;)Ljava/lang/String;");
```

И наконец, остается сделать вызов данного метода, как показано ниже. Следует иметь в виду, что функции `CallStaticObjectMethod()` передается объект, представляющий класс.

```
jobject obj_ret = (*env)->CallStaticObjectMethod(env, class_System,
    id_getProperty, (*env)->NewStringUTF(env, "java.class.path"));
```

Значение, возвращаемое из данного метода, относится к типу `jobject`. Поэтому для выполнения операций над символьными строками его придется привести к типу `jstring` следующим образом:

```
jstring str_ret = (jstring) obj_ret;
```

НА ЗАМЕТКУ C++! В языке C типы `jstring` и `jclass`, а также типы массивов, которые рассматриваются далее, равнозначны типу `jobject`, поэтому выполнять приведение типов в упомянутой выше строке кода на C необязательно. Но в языке C++ эти типы являются указателями на так называемые "фиктивные" классы, имеющие правильную иерархию наследования. Например, присваивание значения типа `jstring` переменной типа `jobject` допускается без приведения типов. Но в то же время приведение типов обязательно для присваивания значения типа `jobject` переменной типа `jstring`.

Конструкторы

В собственном методе можно создать новый объект Java, вызвав его конструктор с помощью функции `NewObject()` в следующей форме:

```
jobject obj_new = (*env)->NewObject(env, класс, идентификатор_метода,
    параметры_конструктора);
```

Для того чтобы получить идентификатор метода, следует вызвать функцию `GetMethodID()`, указав имя метода в символьной строке `"<init>"` и закодированную сигнатуру конструктора с возвращаемым типом `void`. В качестве примера ниже показано, каким образом в собственном методе создается объект типа `FileOutputStream` для потока вывода в файл.

```
const char[] fileName = ". . .";
jstring str_fileName = (*env)->NewStringUTF(env, fileName);
jclass class_FileOutputStream =
    (*env)->FindClass(env, "java/io/FileOutputStream");
jmethodID id_FileOutputStream =
    (*env)->GetMethodID(env, class_FileOutputStream, "<init>",
      "(Ljava/lang/String;)V");
jobject obj_stream =
    (*env)->NewObject(env, class_FileOutputStream, id_FileOutputStream,
      str_fileName);
```

Обратите внимание на то, что в сигнатуре конструктора описывается параметр типа `java.lang.String` и возвращаемое значение типа `void`.

Альтернативные вызовы методов

В прикладном интерфейсе JNI существует несколько вариантов функций, позволяющих вызывать методы на Java из собственного кода. И хотя эти функции играют не такую важную роль, как рассмотренные ранее функции, тем не менее, они могут иногда принести большую пользу.

В частности, функции `CallNonvirtual`*`Xxx`*`Method()` принимают неявный параметр, идентификатор метода, объект класса (который должен соответствовать суперклассу задаваемого неявным образом аргумента), а также явные параметры. Они вызывают версию метода в указанном классе в обход обычного механизма динамической диспетчеризации. Все подобные функции имеют варианты с суффиксом **A** или **V**. Они

получают явно заданные параметры в массиве или структуре типа va_list, определенной в заголовочном файле stdarg.h на C.

Вызов методов на Java

- jmethodID GetMethodID(JNIEnv* *env*, jclass *cl*, const char *name[]*, const char *methodSignature[]*)

  Возвращает идентификатор метода в классе.

- Xxx CallXxxMethod(JNIEnv* *env*, jobject *obj*, jmethodID *id*, *args*)
- Xxx CallXxxMethodA(JNIEnv* *env*, jobject *obj*, jmethodID *id*, jvalue *args[]*)
- Xxx CallXxxMethodV(JNIEnv* *env*, jobject *obj*, jmethodID *id*, va_list *args*)

  Вызывают метод. В качестве возвращаемого типа Xxx может быть указано одно из следующих обозначений: Object, Boolean, Byte, Char, Short, Int, Long, Float или Double. Первая функция принимает переменное количество аргументов, присоединяемых после идентификатора метода. Вторая функция получает параметры метода в виде массива значений типа jvalue, который имеет вид следующего объединения:

```
typedef union jvalue
{
  jboolean z;
  jbyte b;
  jchar c;
  jshort s;
  jint i;
  jlong j;
  jfloat f;
  jdouble d;
  jobject l;
} jvalue;
```

  Третья функция получает параметры метода в виде структуры va_list, определяемой в заголовочном файле stdarg.h на C.

- Xxx CallNonvirtual*Xxx*Method(JNIEnv* *env*, jobject *obj*, jclass *cl*, jmethodID *id*, *args*)
- Xxx CallNonvirtual*Xxx*MethodA(JNIEnv* *env*, jobject *obj*, jclass *cl*, jmethodID *id*, jvalue *args[]*)
- Xxx CallNonvirtual*Xxx*MethodV(JNIEnv* *env*, jobject *obj*, jclass *cl*, jmethodID *id*, va_list *args*)

  Вызывают метод в обход механизма динамической диспетчеризации. В качестве возвращаемого типа *Xxx* может быть указано одно из следующих обозначений: Object, Boolean, Byte, Char, Short, Int, Long, Float или Double. Первая функция имеет переменное количество аргументов, присоединяемых после идентификатора метода. Параметры метода передаются второй функции в массиве типа jvalue. А третья функция получает параметры в виде структуры va_list, определяемой в заголовочном файле stdarg.h на C.

- jmethodID GetStaticMethodID(JNIEnv* *env*, jclass *cl*, const char *name[]*, const char *methodSignature[]*)
- Возвращает идентификатор статического метода в классе.
- *Xxx* CallStatic*Xxx*Method(JNIEnv* *env*, jclass *cl*, jmethodID *id*, *args*)
- *Xxx* CallStatic*Xxx*MethodA(JNIEnv* *env*, jclass *cl*, jmethodID *id*, jvalue *args[]*)
- *Xxx* CallStatic*Xxx*MethodV(JNIEnv* *env*, jclass *cl*, jmethodID *id*, va_list *args*)

  Вызывают статический метод. В качестве возвращаемого типа Xxx может быть указано одно из следующих обозначений: Object, Boolean, Byte, Char, Short, Int, Long, Float или Double. Первая функция имеет переменное количество аргументов, присоединяемых после идентификатора метода. Вторая функция получает параметры метода в виде массива типа jvalue, а третья — в виде структуры va_list, определяемой в заголовочном файле stdarg.h на C.
- jobject NewObject(JNIEnv* *env*, jclass *cl*, jmethodID *id*, *args*)
- jobject NewObjectA(JNIEnv* *env*, jclass *cl*, jmethodID *id*, jvalue *args[]*)
- jobject NewObjectV(JNIEnv* *env*, jclass *cl*, jmethodID *id*, va_list *args*)

  Вызывают конструктор класса. Идентификатор метода получается в результате вызова функции GetMethodID() с именем метода в виде строки "<init>" и возвращаемым типом void. Первая функция имеет переменное количество аргументов, присоединяемых после идентификатора метода. Вторая функция получает параметры метода в виде массива типа jvalue, а третья — в виде структуры va_list, определяемой в заголовочном файле stdarg.h на C.

Доступ к элементам массивов

Все типы массивов в Java имеют соответствующие им типы массивов в C, перечисленные в табл. 12.2.

Таблица 12.2. Соответствие типов массивов в Java и C

Java	C
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
int[]	jintArray
short[]	jshortArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
Object[]	jobjectArray

НА ЗАМЕТКУ C++! В языке C все типы массивов, по существу, являются синонимами типа jobject. А в языке C++ они упорядочены в иерархическую структуру, как показано на рис. 12.3. Тип jarray обозначает обобщенный массив.

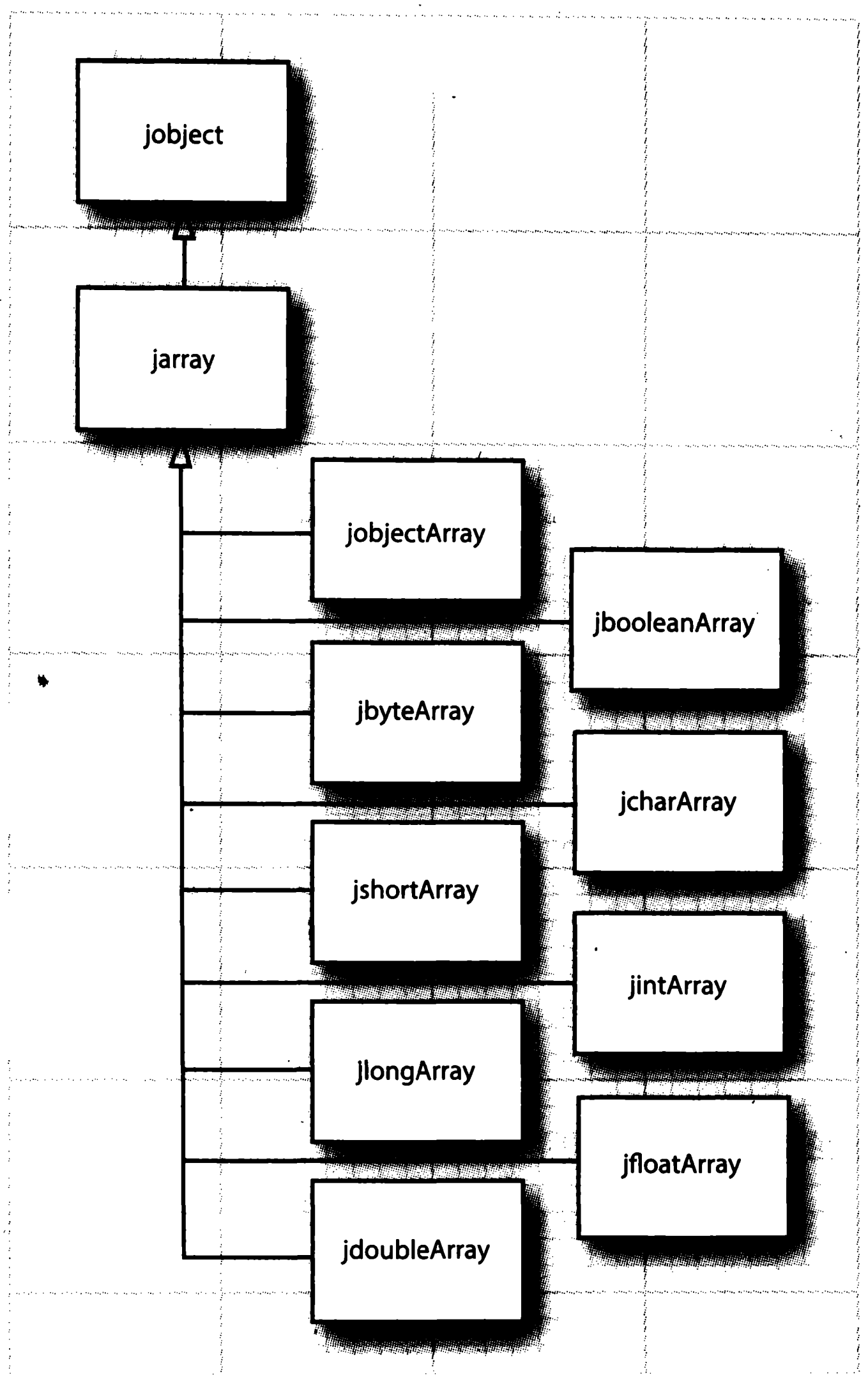


Рис. 12.3. Иерархия наследования типов массивов в C++

Функция `GetArrayLength()` возвращает длину массива, как показано ниже.

```
jarray array = . . .;
jsize length = (*env)->GetArrayLength(env, array);
```

Порядок доступа к элементам массива зависит от того, хранятся ли в нем объекты или значения примитивных типов (числовые, символьные или логические). Для доступа к элементам массива объектов вызываются функции `GetObjectArrayElement()` и `SetObjectArrayElement()`:

```
jobjectArray array = . . .;
int i, j;
jobject x = (*env)->GetObjectArrayElement(env, array, i);
(*env)->SetObjectArrayElement(env, array, j, x);
```

Несмотря на всю свою простоту, такой подход крайне неэффективен. Много лучше получить непосредственный доступ к элементам массива. И это особенно важно для выполнения операций с векторами или матрицами.

Функция `Get`*`Xxx`*`ArrayElements()` возвращает указатель C на начальный элемент массива. Как и при обращении с обычными символьными строками, если этот указатель больше не нужен, следует вызывать функцию `Release`*`Xxx`*`ArrayElements()`, где *Xxx* — примитивный тип данных, а не объект. Используя указатель C, можно непосредственно считывать и записывать значения элементов массива. Но такой указатель может указывать на *копию* массива, поэтому любые изменения будут отражены в исходном массиве только после вызова функции `Release`*`Xxx`*`ArrayElements()`.

НА ЗАМЕТКУ! Чтобы выяснить, с каким именно массивом приходится иметь дело: с оригиналом или копией, укажите в качестве третьего параметра функции `Get`*`Xxx`*`ArrayElements()` переменную типа `jboolean`. Если используется копия массива, то в результате вызова данной функции переменная будет содержать значение `JNI_TRUE`. Если же вас это мало интересует, укажите в качестве третьего параметра функции `Get`*`Xxx`*`ArrayElements()` значение `NULL`.

Ниже приведен фрагмент кода, в котором все элементы массива типа `double` умножаются на постоянное значение `scaleFactor` типа `double`. Доступ к отдельным элементам массива `a[i]` осуществляется по указателю C.

```
jdoubleArray array_a = . . .;
double scaleFactor = . . .;
double* a = (*env)->GetDoubleArrayElements(env, array_a, NULL);
for (i = 0; i < (*env)->GetArrayLength(env, array_a); i++)
  a[i] = a[i] * scaleFactor;
(*env)->ReleaseDoubleArrayElements(env, array_a, a, 0);
```

Будет ли виртуальная машина Java копировать этот массив, зависит от способа выделения памяти и организации "сборки мусора". Некоторые копирующие системы "сборки мусора" перемещают объекты и обновляют ссылки на объекты. Такая методика не совместима с "закреплением" за массивом определенного места в памяти, поскольку система "сборки мусора" не может обновлять значения указателей в собственном коде.

НА ЗАМЕТКУ! В виртуальной машине Sun JVM массив значений типа `boolean` представляется в виде массива, содержащего 32-разрядные значения. Функция `GetBooleanArrayElements()` распаковывает его и копирует в массив типа `jboolean`.

Для доступа лишь к нескольким элементам крупного массива служат функции `Get`*`Xxx`*`ArrayRegion()` и `Set`*`Xxx`*`ArrayRegion()`. Они копируют из массива Java в массив C и обратно элементы в указанных пределах изменения индексов.

Новые массивы Java в собственных методах создаются с помощью функции `New`*`Xxx`*`Array()`. Для создания нового массива объектов нужно указать его длину, тип элементов массива, а также исходное значение всех элементов (как правило, `NULL`). В приведенном ниже примере кода показано, как это делается.

```
jclass class_Employee = (*env)->FindClass(env, "Employee");
jobjectArray array_e = (*env)->NewObjectArray(env, 100, class_Employee, NULL);
```

Массивы элементов простых типов устроены проще. Поэтому для них достаточно указать длину массива. Так, в результате выполнения следующей строки кода массив заполняется нулями:

```
jdoubleArray array_d = (*env)->NewDoubleArray(env, 100);
```

НА ЗАМЕТКУ! Начиная с версии Java SE 1.4 в прикладном интерфейсе JNI появились следующие три функции:

```
jobject NewDirectByteBuffer(JNIEnv* env, void* address, jlong capacity)
void* GetDirectBufferAddress(JNIEnv* env, jobject buf)
jlong GetDirectBufferCapacity(JNIEnv* env, jobject buf)
```

Непосредственные буферы применяются в пакете `java.nio` для поддержки более эффективных операций ввода-вывода и сведения к минимуму числа операций копирования данных между собственным кодом и массивами в Java.

Манипулирование массивами Java

- `jsize GetArrayLength(JNIEnv* env, jarray array)`
  Возвращает количество элементов в массиве.
- `jobject GetObjectArrayElement(JNIEnv* env, jobjectArray array, jsize index)`
  Возвращает значение указанного элемента массива.
- `void SetObjectArrayElement(JNIEnv* env, jobjectArray array, jsize index, jobject value)`
  Устанавливает новое значение в элементе массива.
- *Xxx*`* Get`*Xxx*`ArrayElements(JNIEnv* env, jarray array, jboolean* isCopy)`
  Выдает указатель C на элементы массива Java. В качестве типа `Xxx` поля может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`. Если указатель больше не требуется, он должен быть передан функции `Release`*Xxx*`ArrayElements()`. Параметр *isCopy* принимает значение `NULL` или указатель на переменную типа *jboolean*, которая заполняется значением `JNI_TRUE`, если создается копия массива, а иначе — значение `JNI_FALSE`.
- `void Release`*Xxx*`ArrayElements(JNIEnv* env, jarray array, Xxx elems[], jint mode)`
  Уведомляет виртуальную машину Java, что указатель, полученный с помощью функции `Get`*Xxx*`ArrayElements()`, больше не требуется. Параметр *mode* может принимать одно из следующих значений: **0** (освободить буфер `elems` после обновления элементов массива), `JNI_COMMIT` (не освобождать буфер `elems` после обновления элементов массива) или `JNI_ABORT` (освободить буфер *elems* без обновления элементов массива).
- `void Get`*Xxx*`ArrayRegion(JNIEnv* env, jarray array, jint start, jint length, Xxx elems[])`
  Копирует элементы из массива Java в массив C. В качестве типа *Xxx* поля может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.
- `void SetXxxArrayRegion(JNIEnv* env, jarray array, jint start, jint length, Xxx elems[])`
  Копирует элементы из массива C в массив Java. В качестве типа *Xxx* поля может быть указано одно из следующих обозначений: `Object`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` или `Double`.

Обработка ошибок

Применение собственных методов представляет существенную угрозу нарушить безопасность программ на Java. В языке C не предусмотрена защита от ошибочного указания диапазона массивов, использования некорректных указателей и т.д. Для сохранения целостности программы и виртуальной машины программистам, создающим собственные методы, следует уделять особое внимание обработке подобных ошибок. Так, если собственный метод обнаруживает какую-нибудь ошибку, которую не в состоянии обработать, он должен сообщить о ней виртуальной машине Java, которая сгенерирует соответствующее исключение.

Для создания нового объекта исключения нужно вызвать функцию `Throw()` или `ThrowNew()`, генерирующую исключение в виртуальной машине Java по завершении собственного метода. Для того чтобы воспользоваться функцией `Throw()`, следует вызвать функцию `NewObject()` с целью создать экземпляр класса, производного от класса `Throwable`. Например, в приведенном ниже фрагменте кода создается объект типа `EOFException` и генерируется соответствующее исключение.

```
jclass class_EOFException = (*env)->FindClass(env, "java/io/EOFException");
jmethodID id_EOFException = (*env)->GetMethodID(env, class_EOFException,
"<init>", "()V");
    /* идентификатор конструктора без аргументов */
jthrowable obj_exc = (*env)->NewObject(env, class_EOFException,
id_EOFException);
(*env)->Throw(env, obj_exc);
```

Но на практике удобнее пользоваться функцией `ThrowNew()`. Она также создает объект исключения, исходя из заданного класса и байтовой последовательности в модифицированном формате UTF-8:

```
(*env)->ThrowNew(env, (*env)->FindClass(env, "java/io/EOFException"),
    "Unexpected end of file");
```

Обе функции, `Throw()` и `ThrowNew()`, только передают исключение, а не прерывают поток управления собственного метода. Поэтому виртуальная машина Java может генерировать исключение только по завершении собственного метода. А это означает, что после каждого вызова функции `Throw()` или `ThrowNew()` должен быть указан оператор `return`.

НА ЗАМЕТКУ C++! Если собственный метод реализуется на C++, в его теле нельзя генерировать объект исключения Java. Теоретически взаимное преобразование исключений между языками C++ и Java возможно, но в настоящее время оно не реализовано. Для генерирования объектов исключений Java в собственном коде на C++ придется использовать функции `Throw()` и `ThrowNew()`, а также принять меры, чтобы собственные методы не генерировали исключения C++.

Обычно собственный код не должен заниматься обработкой исключений Java, но подобная ситуация может возникнуть при вызове из собственной функции метода на Java. Например, функция `SetObjectArrayElement()` генерирует исключение типа `ArrayIndexOutOfBoundsException`, если значение индекса выходит за границы массива, а также исключение типа `ArrayStoreException`, если класс хранимого объекта не является производным от класса элемента массива. В подобных случаях собственный метод должен вызывать функцию `ExceptionOccured()`, чтобы определить факт генерирования исключения. В результате приведенного ниже вызова

возвращается ссылка на текущий объект исключения. Если же в очереди отсутствуют исключения, ожидающие обработки, функция возвращает значение NULL.

```
jboolean occurred = (*env)->ExceptionCheck(env);
```

Обычно при генерировании исключения собственный метод *лишь* возвращает управление. В итоге виртуальная машина Java может передать исключение соответствующему коду на Java. Но собственный метод *может* проанализировать объект исключения и определить, сумеет ли он обработать его самостоятельно. Если такая обработка возможна, необходимо отозвать сообщение об исключении с помощью следующей функции:

```
(*env)->ExceptionClear(env);
```

В рассматриваемом здесь примере программы реализуется собственный метод `fprintf()` с некоторыми чрезмерными, на первый взгляд, но весьма полезными на практике мерами по обработке следующих исключений:

- `NullPointerException`, если указатель на форматирующую строку принимает значение NULL.
- `IllegalArgumentException`, если форматирующая строка не содержит спецификатор %, необходимый для вывода значений типа `double`.
- `OutOfMemoryError`, если произошла ошибка при выполнении функции `malloc()`.

И наконец, для того чтобы продемонстрировать, каким образом выполняется проверка на наличие исключения при вызове метода на Java из собственного кода, строка направляется в поток вывода посимвольно, и после вывода каждого символа вызывается функция `ExceptionOccured()`. В листинге 12.17 представлен исходный код собственного метода, а в листинге 12.18 — исходный код класса, содержащего этот метод. Следует иметь в виду, что если при вызове метода `PrintWriter.print()` возникает исключение, то собственный метод завершает свою работу не сразу — сначала он освобождает буфер `cstr`. Когда же происходит возврат из собственного метода, виртуальная машина Java снова генерирует исключение. Тестовая программа из листинга 12.19 позволяет увидеть, как именно собственный метод генерирует исключение, если форматирующая строка оказывается недействительной.

Листинг 12.17. Исходный ход из файла `printf4/Printf4.c`

```
1   /**
2    * @version 1.10 1997-07-01
3    * @author Cay Horstmann
4   */
5
6   #include "Printf4.h"
7   #include <string.h>
8   #include <stdlib.h>
9   #include <float.h>
10
11  /**
12   * @param format Символьная строка со спецификатором формата для
13   * функции printf(), например, "%8.2f". Подстроки "%%" пропускаются
14   * @return Возвращает указатель на спецификатор формата, пропуская
15   *         символ '%', или значение NULL в отсутствие однозначного
```

```
16   *          спецификатора формата
17  */
18  char* find_format(const char format[])
19  {
20    char* p;
21    char* q;
22
23    p = strchr(format, '%');
24    while (p != NULL && *(p + 1) == '%') /* пропустить подстроку "%%" */
25      p = strchr(p + 2, '%');
26    if (p == NULL) return NULL;
27    /* проверить единственность символа "%" */
28    p++;
29    q = strchr(p, '%');
30    while (q != NULL && *(q + 1) == '%') /* пропустить подстроку "%%" */
31      q = strchr(q + 2, '%');
32    if (q != NULL) return NULL; /* символ "%" не единственный */
33    q = p + strspn(p, " -0+#"); /* пропустить признаки */
34    q += strspn(q, "0123456789"); /* пропустить ширину поля */
35    if (*q == '.') { q++; q += strspn(q, "0123456789"); }
36      /* пропустить точность */
37    if (strchr("eEfFgG", *q) == NULL) return NULL;
38      /* это не формат с плавающей точкой */
39    return p;
40  }
41
42  JNIEXPORT void JNICALL Java_Printf4_fprint(JNIEnv* env, jclass cl,
43     jobject out, jstring format, jdouble x)
44  {
45    const char* cformat;
46    char* fmt;
47    jclass class_PrintWriter;
48    jmethodID id_print;
49    char* cstr;
50    int width;
51    int i;
52
53    if (format == NULL)
54    {
55      (*env)->ThrowNew(env,
56        (*env)->FindClass(env,
57        "java/lang/NullPointerException"),
58        "Printf4.fprint: format is null");
59      return;
60    }
61
62    cformat = (*env)->GetStringUTFChars(env, format, NULL);
63    fmt = find_format(cformat);
64
65    if (fmt == NULL)
66    {
67      (*env)->ThrowNew(env,
68        (*env)->FindClass(env,
69        "java/lang/IllegalArgumentException"),
70        "Printf4.fprint: format is invalid");
71      return;
72    }
73    width = atoi(fmt);
```

```
74    if (width == 0) width = DBL_DIG + 10;
75    cstr = (char*)malloc(strlen(cformat) + width);
76
77    if (cstr == NULL)
78    {
79      (*env)->ThrowNew(env,
80        (*env)->FindClass(env, "java/lang/OutOfMemoryError"),
81         "Printf4.fprint: malloc failed");
82      return;
83    }
84
85    sprintf(cstr, cformat, x);
86
87    (*env)->ReleaseStringUTFChars(env, format, cformat);
88
89    /* вызвать функцию ps.print(str) */
90
91    /* получить класс */
92    class_PrintWriter = (*env)->GetObjectClass(env, out);
93
94    /* получить идентификатор метода */
95    id_print = (*env)->GetMethodID(env, class_PrintWriter, "print", "(C)V");
96
97    /* вызвать метод */
98    for (i = 0; cstr[i] != 0 && !(*env)->ExceptionOccurred(env); i++)
99      (*env)->CallVoidMethod(env, out, id_print, cstr[i]);
100
101   free(cstr);
102 }
```

Листинг 12.18. Исходный ход из файла `printf4/Printf4.java`

```
1  import java.io.*;
2
3  /**
4   * @version 1.10 1997-07-01
5   * @author Cay Horstmann
6  */
7  class Printf4
8  {
9    public static native void fprint(PrintWriter ps, String format, double x);
10
11   static
12   {
13     System.loadLibrary("Printf4");
14   }
15 }
```

Листинг 12.19. Исходный ход из файла `printf4/Printf4Test.java`

```
1  import java.io.*;
2
3  /**
4   * @version 1.10 1997-07-01
5   * @author Cay Horstmann
6  */
```

```
7  class Printf4Test
8  {
9    public static void main(String[] args)
10   {
11     double price = 44.95;
12     double tax = 7.75;
13     double amountDue = price * (1 + tax / 100);
14     PrintWriter out = new PrintWriter(System.out);
15     /* При этом вызове может быть сгенерировано исключение,
16      * если в форматирующей строке отсутствует подстрока "%%" */
17      Printf4.fprint(out, "Amount due = %%8.2f\n", amountDue);
18      out.flush();
19   }
20 }
```

Обработка исключений в Java

- `jint Throw(JNIEnv* env, jthrowable obj)`

  Подготавливает исключение, которое должно генерироваться после выхода из собственного кода. При удачном исходе возвращает нулевое значение, в противном случае — отрицательное значение.

- `jint ThrowNew(JNIEnv* env, jclass cl, const char msg[])`

  Подготавливает исключение типа *cl*, которое должно генерироваться при выходе из собственного кода. При удачном исходе возвращает нулевое значение, в противном случае — отрицательное значение. Параметр *msg* обозначает аргумент для построения объекта исключения типа `String` в виде последовательности байтов в "модифицированном формате UTF-8".

- `jthrowable ExceptionOccurred(JNIEnv* env)`

  Возвращает объект исключения, если исключение ожидает своей очереди, а иначе — значение `NULL`.

- `jboolean ExceptionCheck(JNIEnv* env)`

  Возвращает логическое значение `true`, если исключение все еще ожидает своей очереди.

- `void ExceptionClear(JNIEnv* env)`

  Удаляет все исключения, ожидающие своей очереди.

Применение прикладного интерфейса API для вызовов

До сих пор рассматривались только программы на Java, делавшие вызовы на C по двум наиболее вероятным причинам: код на C мог выполняться быстрее или требовался доступ к таким функциональным возможностям, которые были недоступны на платформе Java. А теперь рассмотрим обратную ситуацию, когда имеется программа на C или C++ и требуется организовать из нее несколько вызовов кода на Java. Внедрять виртуальную машину Java в программу на C или C++ позволяет прикладной интерфейс API, специально предназначенный *для вызовов*. Ниже приведен минимальный фрагмент кода, который требуется для инициализации виртуальной машины Java.

```
JavaVMOption options[1];
JavaVMInitArgs vm_args;
JavaVM *jvm;
```

```
JNIEnv *env;

options[0].optionString = "-Djava.class.path=.";

memset(&vm_args, 0, sizeof(vm_args));
vm_args.version = JNI_VERSION_1_2;
vm_args.nOptions = 1;
vm_args.options = options;

JNI_CreateJavaVM(&jvm, (void**) &env, &vm_args);
```

Вызов функции `JNI_CreateJavaVM()` приводит к созданию виртуальной машины и заполнению указателя `jvm` на нее, а также указателя `env` на среду выполнения. Количество параметров для виртуальной машины Java можно указывать любым, увеличивая размер массива `options` и значение в поле `vm_args.nOptions`. Например, выполнение следующей строки кода приведет к отключению динамического компилятора:

```
options[i].optionString = "-Djava.compiler=NONE";
```

СОВЕТ. Если возникают неполадки следующего характера: программа завершается аварийно, отказывается инициализировать виртуальную машину Java или не в состоянии загрузить классы, можно включить режим отладки. Это делается следующим образом:

```
options[i].optionString = "-verbose:jni";
```

В этом режиме выводятся сообщения с подробными сведениями о ходе инициализации JVM. Если же сообщения о загрузке классов отсутствуют, следует проверить установки как общего пути, так и пути к классам.

После настройки виртуальной машины Java можно приступать к вызову методов на Java способом, описанным в предыдущих разделах, т.е. с помощью указателя `env`. А указатель `jvm` следует использовать только в том случае, если требуется вызвать другие функции из прикладного интерфейса API. В настоящее время имеется лишь четыре такие функции. Наиболее важной из них является функция `DestroyJavaVM()`, вызываемая для прекращения работы виртуальной машины Java:

```
(*jvm)->DestroyJavaVM(jvm);
```

К сожалению, в Windows стало трудно осуществлять динамическое связывание с функцией `JNI_CreateJavaVM()` из библиотеки `jre/bin/client/jvm.dll` из-за того, что правила связывания в версии Vista изменились, а компания Oracle по-прежнему делает ставку на устаревшую версию библиотеки рабочих программ на C. В рассматриваемом здесь примере программы данное затруднение разрешается путем загрузки упомянутой библиотеки вручную. Такой же самый подход применяется и в утилите `java`. Убедиться в этом можно, проанализировав исходный код из файла `launcher/java_md.c`, находящегося в архивном файле `src.jar`, включаемом в состав JDK.

В листинге 12.20 представлен исходный код программы на C, в которой сначала инициализируется виртуальная машина Java, а затем вызывается метод `main()` из класса `Welcome`, который рассматривался в главе 2 первого тома данной книги. (Перед запуском этой программы следует скомпилировать исходный файл `Welcome.java`.)

Листинг 12.20. Исходный код из файла `invocation/InvocationTest.c`

```
/**
 * @version 1.20 2007-10-26
 * @author Cay Horstmann
*/

#include <jni.h>
#include <stdlib.h>

#ifdef _WINDOWS

#include <windows.h>
static HINSTANCE loadJVMLibrary(void);
typedef jint (JNICALL *CreateJavaVM_t)(JavaVM **, void **,
   JavaVMInitArgs *);

#endif

int main()
{
  JavaVMOption options[2];
  JavaVMInitArgs vm_args;
  JavaVM *jvm;
  JNIEnv *env;
  long status;

  jclass class_Welcome;
  jclass class_String;
  jobjectArray args;
  jmethodID id_main;

#ifdef _WINDOWS
  HINSTANCE hjvmlib;
  CreateJavaVM_t createJavaVM;
#endif

  options[0].optionString = "-Djava.class.path=.";
  memset(&vm_args, 0, sizeof(vm_args));
  vm_args.version = JNI_VERSION_1_2;
  vm_args.nOptions = 1;
  vm_args.options = options;

#ifdef _WINDOWS
  hjvmlib = loadJVMLibrary();
  createJavaVM = (CreateJavaVM_t) GetProcAddress(
     hjvmlib, "JNI_CreateJavaVM");
  status = (*createJavaVM)(&jvm, (void **) &env, &vm_args);
#else
  status = JNI_CreateJavaVM(&jvm, (void **) &env, &vm_args);
#endif

  if (status == JNI_ERR)
  {
    fprintf(stderr, "Error creating VM\n");
    return 1;
  }
```

```
56
57    class_Welcome = (*env)->FindClass(env, "Welcome");
58    id_main = (*env)->GetStaticMethodID(env, class_Welcome, "main",
59       "([Ljava/lang/String;)V");
60
61    class_String = (*env)->FindClass(env, "java/lang/String");
62    args = (*env)->NewObjectArray(env, 0, class_String, NULL);
63    (*env)->CallStaticVoidMethod(env, class_Welcome, id_main, args);
64
65    (*jvm)->DestroyJavaVM(jvm);
66
67    return 0;
68  }
69
70  #ifdef _WINDOWS
71
71  static int GetStringFromRegistry(HKEY key, const char *name,
72     char *buf, jint bufsize)
73  {
74    DWORD type, size;
75
76    return RegQueryValueEx(key, name, 0, &type, 0, &size) == 0
77       && type == REG_SZ
78       && size < (unsigned int) bufsize
79       && RegQueryValueEx(key, name, 0, 0, buf, &size) == 0;
80  }
81
82  static void GetPublicJREHome(char *buf, jint bufsize)
83  {
84    HKEY key, subkey;
85    char version[MAX_PATH];
86    /* найти текущую версию JRE */
87    char *JRE_KEY = "Software\\JavaSoft\\Java Runtime Environment";
88    if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, JRE_KEY, 0, KEY_READ, &key) != 0)
89    {
90      fprintf(stderr, "Error opening registry key '%s'\n", JRE_KEY);
91      exit(1);
92    }
93
94    if (!GetStringFromRegistry(key, "CurrentVersion", version,
95       sizeof(version)))
96    {
97      fprintf(stderr, "Failed reading value of registry key:
98         \n\t%s\\CurrentVersion\n", JRE_KEY);
99      RegCloseKey(key);
100     exit(1);
101   }
102
103   /* Найти каталог, где установлена текущая версия */
104   if (RegOpenKeyEx(key, version, 0, KEY_READ, &subkey) != 0)
105   {
106     fprintf(stderr, "Error opening registry key '%s\\%s'\n",
107        JRE_KEY, version);
108     RegCloseKey(key);
109     exit(1);
110   }
111
112   if (!GetStringFromRegistry(subkey, "JavaHome", buf, bufsize))
```

```
113   {
114     fprintf(stderr, "Failed reading value of registry key:
115         \n\t%s\\%s\\JavaHome\n", 110 JRE_KEY, version);
116     RegCloseKey(key);
117     RegCloseKey(subkey);
118     exit(1);
119   }
120
121   RegCloseKey(key);
122   RegCloseKey(subkey);
123 }
124
125 static HINSTANCE loadJVMLibrary(void)
126 {
127   HINSTANCE h1, h2;
128   char msvcdll[MAX_PATH];
129   char javadll[MAX_PATH];
130   GetPublicJREHome(msvcdll, MAX_PATH);
131   strcpy(javadll, msvcdll);
132   strncat(msvcdll, "\\bin\\msvcr71.dll", MAX_PATH - strlen(msvcdll));
133   msvcdll[MAX_PATH - 1] = '\0';
134   strncat(javadll, "\\bin\\client\\jvm.dll", MAX_PATH - strlen(javadll));
135   javadll[MAX_PATH - 1] = '\0';
136
137   h1 = LoadLibrary(msvcdll);
138   if (h1 == NULL)
139   {
140     fprintf(stderr, "Can't load library msvcr71.dll\n");
141     exit(1);
142   }
143
144   h2 = LoadLibrary(javadll);
145   if (h2 == NULL)
146   {
147     fprintf(stderr, "Can't load library jvm.dll\n");
148     exit(1);
149   }
150   return h2;
151 }
152
153 #endif
```

Чтобы скомпилировать эту программу в ОС Linux, выполните следующую команду:

```
gcc -I jdk/include -I jdk/include/linux -o InvocationTest
    -L jdk/jre/lib/i386/client -ljvm InvocationTest.c
```

В ОС Solaris выполните для этой цели такую команду:

```
cc -I jdk/include -I jdk/include/solaris -o InvocationTest
    -L jdk/jre/lib/sparc -ljvm InvocationTest.c
```

Если же требуется скомпилировать данную программу в ОС Windows компилятором от корпорации Microsoft, выполните команду

```
cl -D_WINDOWS -I jdk\include -I jdk\include\win32 InvocationTest.c
    jdk\lib\jvm.lib advapi32.lib
```

Непременно убедитесь, что в переменные окружения INCLUDE LIB включен путь к заголовочным файлам и файлам библиотек из прикладного интерфейса API для Windows. В среде Cygwin данную программу можно скомпилировать по команде

```
gcc -D_WINDOWS -mno-cygwin -I jdk\include -I jdk\include\win32 -D__int64="long
    long" -I c:\cygwin\usr\include\w32api -o InvocationTest
```

Перед запуском данной программы в ОС Linux/UNIX следует удостовериться, что в переменную окружения LD_LIBRARY_PATH включены каталоги для разделяемых библиотек. Так, если в ОС Linux используется командная оболочка bash, то для этой цели нужно выполнить следующую команду:

```
export LD_LIBRARY_PATH=jdk/jre/lib/i386/client:$LD_LIBRARY_PATH
```

Функции прикладного интерфейса API для вызовов

- `jint JNI_CreateJavaVM(JavaVM **p_jvm, void **p_env, JavaVMInitArgs *vm_args)`

  Инициализирует виртуальную машину Java. При удачном завершении возвращает нулевое значение, а иначе — значение JNI_ERR.

  Параметры:

p_jvm	Указатель на таблицу функций из прикладного интерфейса API для вызовов
p_env	Указатель на таблицу функций из прикладного интерфейса JNI
vm_args	Аргументы для виртуальной машины

- `jint DestroyJavaVM(JavaVM *jvm)`

  Удаляет виртуальную машину Java. При удачном завершении возвращает нулевое значение, в противном случае — отрицательное значение. Эту функцию следует вызывать с помощью указателя на виртуальную машину, например `(*jvm)->DestroyJavaVM(jvm)`

Пример обращения к реестру Windows

В этом разделе рассматривается пример использования всех перечисленных ранее в этой главе способов обращения с собственными методами для обработки символьных строк, массивов, объектов, вызова конструкторов и обработки исключений. Основная цель данного примера — продемонстрировать создание на платформе Java оболочки для подмножества функций прикладного интерфейса API, реализованных на C и предназначенных для работы с реестром ОС Windows. В связи с тем что в рассматриваемом здесь примере программы применяются функции для обращения с таким специфическим объектом, как реестр Windows, она, разумеется, не может быть перенесена на другие платформы. Именно поэтому в Java не предусмотрено никаких инструментальных средств для работы с реестром, а предполагается, что разработчики будут пользоваться для этой цели собственным кодом.

Общее представление о реестре Windows

Как известно, реестр представляет собой хранилище конфигурационных данных для операционной системы Windows и прикладных программ. Наличие такого хранилища очень удобно для администрирования, создания резервных копий и настройки прикладных программ. Недостаток реестра, однако, заключается в том, что он является также единой точкой отказа: при наличии ошибок в реестре вся операционная система будет работать со сбоями, а возможно, и вообще не загрузится!

В связи с этим использовать реестр для хранения конфигурационных параметров прикладных программ на Java не рекомендуется. С этой целью лучше воспользоваться прикладным интерфейсом API для сохранения глобальных параметров настройки (см. главу 10 первого тома данной книги). В данном случае реестр используется лишь для того, чтобы продемонстрировать, каким образом собственные функции из нетривиального прикладного интерфейса API заключаются в оболочку класса Java.

Основным инструментальным средством для работы с реестром является *редактор реестра*. Его применение чревато крупными неприятностями для начинающих пользователей, поэтому для его запуска не предусмотрено никаких визуальных элементов пользовательского интерфейса. Чтобы вызывать редактор реестра из командной строки или в диалоговом окне, которое появляется на экране после выбора пункта меню Пуск⇨Выполнить, следует ввести команду **`regedit`**. На рис. 12.4 показано, как выглядит окно редактора реестра.

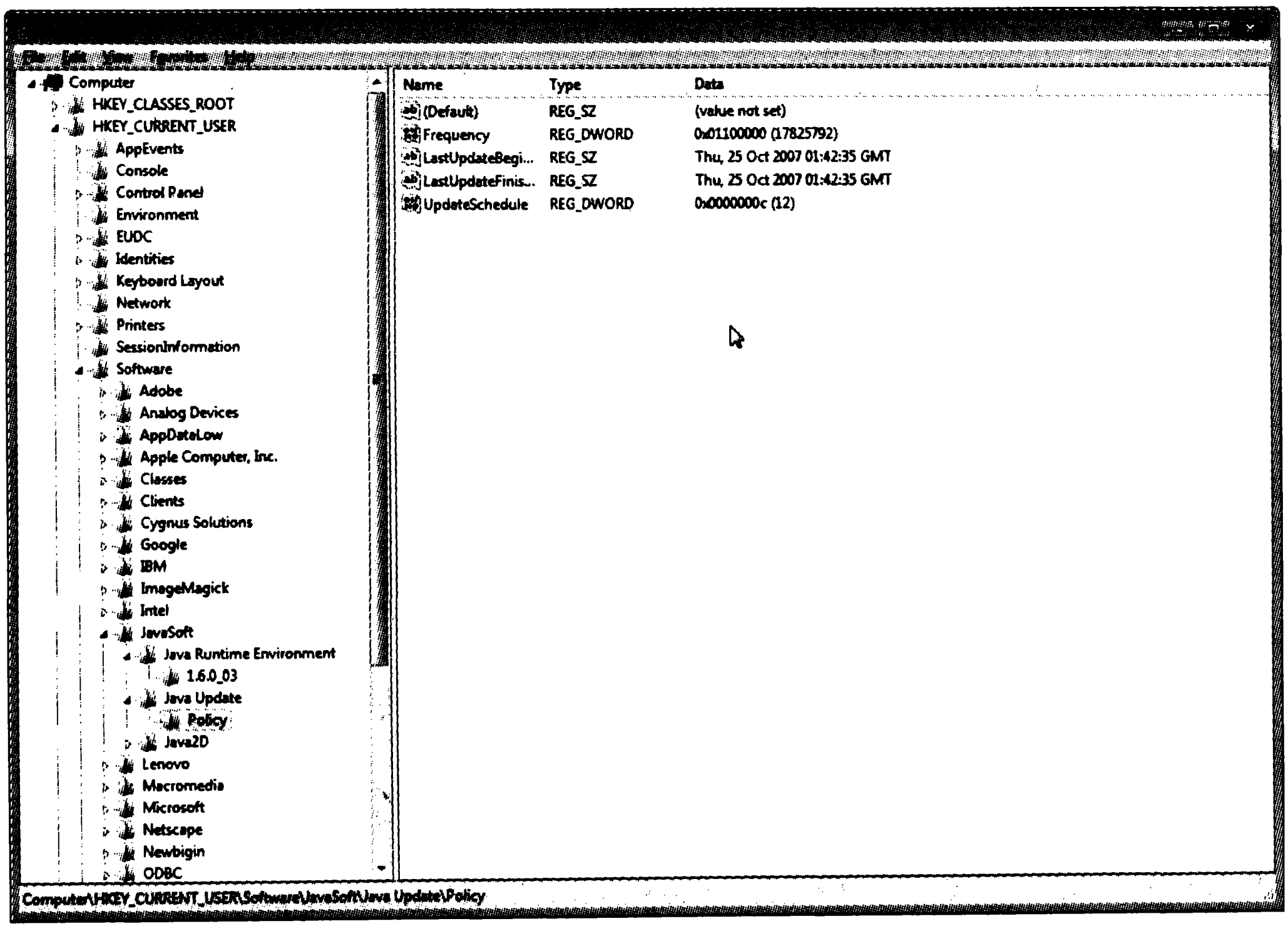


Рис. 12.4. Редактор реестра

В левой части окна редактора реестра представлены ключи, упорядоченные в виде иерархической древовидной структуры. Следует иметь в виду, что каждый ключ находится в поддереве, начинающемся в одном из перечисленных ниже узлов с префиксом **HKEY**.

```
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
. . .
```

В правой части окна редактора реестра отображаются пары "имя–значение", связанные с конкретным ключом. Так, если установить версию Java SE 7, то ключ

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Runtime Environment
```

будет содержать такую пару "имя–значение":

```
CurrentVersion="1.7.0_10"
```

В данном случае значение представлено в виде символьной строки. Но значения могут быть также представлены в виде целых чисел или массивов байтов.

Интерфейс на платформе Java для доступа к реестру

Рассмотрим создание простого интерфейса для организации доступа к реестру из прикладной программы на Java, а также его реализацию в собственном коде. Этот прикладной интерфейс позволяет выполнять лишь несколько операций с реестром. Ради сокращения объема разрабатываемого кода в данном прикладном интерфейсе не поддерживаются многие важные операции, в том числе ввод и удаление ключей. Но для их выполнения в данном прикладном интерфейсе нетрудно реализовать соответствующие функции.

Даже используя ограниченный набор функций, с реестром можно выполнять следующие действия:

- получать все имена, связанные с ключом;
- считывать значение, соответствующее имени;
- связывать с именем новое значение.

Для выполнения подобных действий служит класс Java, инкапсулирующий ключ реестра:

```
public class Win32RegKey
{
  public Win32RegKey(int theRoot, String thePath) { . . . }
  public Enumeration names() { . . . }
  public native Object getValue(String name);
  public native void setValue(String name, Object value);

  public static final int HKEY_CLASSES_ROOT = 0´80000000;
  public static final int HKEY_CURRENT_USER = 0´80000001;
  public static final int HKEY_LOCAL_MACHINE = 0´80000002;
  . . .
}
```

Метод `names()` возвращает объект типа `Enumeration`, который содержит все имена, связанные с данным ключом. Их можно получить с помощью методов `hasMoreElements()`/`nextElement()`. Метод `getValue()` возвращает объект,

который может быть символьной строкой, экземпляром класса `Integer` или байтовым массивом. Параметр `value` метода `setValue()` также должен относиться к одному из этих трех типов.

Реализация функций доступа к реестру в виде собственных методов

В рассматриваемом здесь примере необходимо реализовать следующие функциональные возможности:

- получение значения по имени;
- установка значения по имени;
- перечисление имен по ключу.

Правда, ранее уже были рассмотрены все основные инструментальные средства, требующиеся для взаимного преобразования символьных строк и массивов на Java и C. Кроме того, ранее было показано, каким образом генерируются и обрабатываются исключения при возникновении ошибок.

Но по сравнению с примерами из предыдущих разделов применение собственных методов усложняется следующими двумя обстоятельствами. Во-первых, методы `getValue()` и `setValue()` обращаются с объектами типа `Object`, которые могут относиться к одному из следующих конкретных типов: `String`, `Integer` или `byte[]`. Кроме того, объект перечисления должен сохранять сведения о состоянии между последовательными вызовами методов `hasMoreElements()`/`nextElement()`. Рассмотрим сначала метод `getValue()` (его исходный код представлен в листинге 12.22), который выполняет перечисленные ниже действия.

1. Открывает ключ в реестре. В прикладном интерфейсе API для доступа к реестру требуется, чтобы ключи были открыты для считывания значений.
2. Запрашивает тип и длину значения, связанного с данным именем.
3. Считывает данные значения в буфер.
4. Вызывает функцию `NewStringUTF()` для создания новой символьной строки с данными значения, если это значение относится к типу `REG_SZ` (т.е. является строкой).
5. Вызывает конструктор класса `Integer`, если это значение относится к типу `REG_DWORD` (т.е. является 32-разрядным целочисленным значением).
6. Взывает сначала функцию `NewByteArray()` для создания нового байтового массива, а затем функцию `SetByteArrayRegion()` для копирования данных значения в полученный массив, если это значение относится к типу `REG_BINARY` (т.е. является байтовым массивом).
7. Если же это значение не относится ни к одному из перечисленных типов или если при вызове функции из прикладного интерфейса API возникла какая-нибудь ошибка, генерирует исключение и освобождает все полученные до сих пор ресурсы.
8. Закрывает ключ в реестре и возвращает созданный объект (типа `String`, `Integer` или `byte[]`).

Как видите, рассматриваемый здесь пример наглядно показывает, каким образом формируются объекты разных типов в Java.

В данном собственном методе совсем не трудно организовать возврат обобщенного значения, поскольку ссылку типа `jstring`, `jint` или `byte[]` можно возвратить просто как ссылку типа `jobject`. Но метод `setValue()` получает ссылку на объект типа `Object` и должен точно определить его конкретный тип, чтобы сохранить его в виде символьной строки, целочисленного значения или байтового массива. Этот тип можно определить, запросив класс объекта `value` и получив ссылки на классы `java.lang.String`, `java.lang.Integer` или `java.lang.byte[]`, а затем сравнив их с помощью функции `IsAssignableFrom()`.

Если, например, `class1` и `class2` являются ссылками на два разных класса, то в результате вызова приведенного ниже метода возвратится значение `JNI_TRUE`, при условии, что классы `class1` и `class2` одинаковы или же класс `class1` является подклассом, производным от класса `class2`.

```
(*env)->IsAssignableFrom(env, class1, class2)
```

В любом случае ссылки на объекты класса `class1` могут быть приведены к типу `class2`. Так, если представленный ниже метод возвращает логическое значение `true`, это означает, что объект `value` является байтовым массивом.

```
(*env)->IsAssignableFrom(env, (*env)->GetObjectClass(env, value),
    (*env)->FindClass(env, "[B"))
```

А теперь рассмотрим метод `setValue()`, выполняющий действия, описанные ниже.

1. Открывает ключ в реестре для записи.
2. Определяет тип записываемого значения.
3. Вызывает функцию `GetStringUTFChars()` для получения указателя на символы, если значение относится к типу `String`.
4. Вызывает метод `intValue()` для извлечения целочисленного значения, хранящегося в объекте-оболочке, если значение относится к типу `Integer`.
5. Вызывает метод `GetByteArrayElements()` для получения указателя на байты, если значение относится к типу `byte[]`.
6. Передает данные и длину значения в реестр.
7. Закрывает ключ в реестре.
8. Освобождает указатель на данные, если речь идет о значении типа `String` или `byte[]`.

И наконец, рассмотрим собственные методы для перечисления имен по ключу. Как следует из листинга 12.21, эти методы входят в состав класса `Win32RegKeyNameEnumeration`. Для того чтобы начать процесс перечисления, следует сначала открыть ключ в реестре. На время существования объекта перечисления нужно сохранить дескриптор ключа. Этот дескриптор относится к типу `DWORD` (32-разрядное целочисленное значение), и поэтому он может быть сохранен в поле `hkey` класса перечисления. В самом начале процесса перечисления это поле инициализируется с помощью метода `SetIntField()`. А в дальнейшем значение поля `hkey` считывается с помощью метода `GetIntField()`.

В рассматриваемом здесь примере в объекте перечисления сохраняются также три других элемента данных. В самом начале процесса перечисления можно запросить в реестре количество пар "имя–значение", а также длину самого длинного имени. Эти сведения требуются для выделения символьного массива на C и сохраняются в полях `count` и `maxsize` объекта перечисления. И наконец, поле `index` этого объекта инициализируется значением **-1**, указывая на начало процесса перечисления. После инициализации других полей экземпляра в этом поле устанавливается нулевое значение, которое затем инкрементируется на каждом шаге перечисления.

Рассмотрим по очереди действия собственных методов, поддерживающих перечисление. Начнем с самого простого метода `hasMoreElements()`, который выполняет следующие действия.

1. Извлекает значения полей `index` и `count`.
2. Если в поле `index` оказывается значение **-1**, то вызывается функция `startNameEnumeration()`, которая открывает соответствующий ключ в реестре, запрашивает значение счетчика и максимальную длину, а затем инициализирует поля `hkey`, `count`, `maxsize` и `index`.
3. Если значение в поле `index` меньше значения в поле `count`, возвращает значение `JNI_TRUE`, в противном случае — значение `JNI_FALSE`.

Метод `nextElement()` более сложен и выполняет перечисленные ниже действия.

1. Извлекает значения полей `index` и `count`.
2. Если значение в поле `index` равно **-1**, вызывает функцию `startNameEnumeration()`, которая открывает соответствующий ключ в реестре, запрашивает значение счетчика и максимальную длину, а затем инициализирует поля `hkey`, `count`, `maxsize` и `index`.
3. Если значение в поле `index` равно значению в поле `count`, генерирует исключение типа `NoSuchElementException`.
4. Считывает следующее имя из реестра.
5. Увеличивает значение в поле `index`.
6. Инкрементирует значение в поле `index`.
7. Если значение в поле `index` снова оказывается равным значению в поле `count`, закрывает ключ в реестре.

Перед компиляцией не забудьте запустить на выполнение утилиту `javah` с обоими заголовочными файлами — `Win32RegKey` и `Win32RegKeyNameEnumeration`. Для компилятора от корпорации Microsoft соответствующая командная строка будет выглядеть так:

```
cl -I jdk\include -I jdk\include\win32 -LD Win32RegKey.c
    advapi32.lib -FeWin32RegKey.dll
```

В среде Cygwin аналогичная команда будет следующей:

```
gcc -mno-cygwin -D __int64="long long" -I jdk\include -I jdk\include\win32
    -I c:\cygwin\usr\include\w32api -shared -Wl,--add-stdcall-alias
    -o Win32RegKey.dll Win32RegKey.c
```

Но прикладной интерфейс API для доступа к реестру рассчитан исключительно на применение в ОС Windows, поэтому в других операционных системах рассматриваемая здесь программа работать не будет.

В листинге 12.23 приведен код программы, позволяющей протестировать новые функции, разработанные для обращения с реестром. Для проведения такого тестирования три пары "имя–значение" (символьная строка, целое число и байтовый массив) сначала добавляются в следующий ключ реестра:

```
HKEY_CURRENT_USER\Software\JavaSoft\Java Runtime Environment
```

Затем перечисляются все имена этого ключа и извлекаются их значения. Ниже приведен результат выполнения данной программы.

```
Default user=Harry Hacker
Lucky number=13
Small primes=2 3 5 7 11 13
```

Несмотря на то что добавление подобных пар имен и значений в данный ключ вряд ли сможет нанести какой-нибудь вред реестру, после выполнения данной программы их нетрудно удалить, если потребуется, с помощью редактора реестра.

Листинг 12.21. Исходный код из файла `win32reg/Win32RegKey.java`

```
1  import java.util.*;
2
3  /**
4   * Объект класса Win32RegKey служит для получения и установки
5   * значений в ключе реестра Windows
6   * @version 1.00 1997-07-01
7   * @author Cay Horstmann
8  */
9  public class Win32RegKey
10 {
11   public static final int HKEY_CLASSES_ROOT = 0x80000000;
12   public static final int HKEY_CURRENT_USER = 0x80000001;
13   public static final int HKEY_LOCAL_MACHINE = 0x80000002;
14   public static final int HKEY_USERS = 0x80000003;
15   public static final int HKEY_CURRENT_CONFIG = 0x80000005;
16   public static final int HKEY_DYN_DATA = 0x80000006;
17
18   private int root;
19   private String path;
20
21   /**
22    * Получает значение ключа в реестре
23    * @param name Имя ключа
24    * @return Возвращает значение, связанное с заданным ключом
25   */
26   public native Object getValue(String name);
27
28   /**
29    * Устанавливает значение ключа в реестре
30    * @param name Имя ключа
31    * @param value Новое значение
32   */
33   public native void setValue(String name, Object value);
34
35   /**
36    * Создает объект ключа в реестре
37    * @param theRoot Один из следующих ключей: HKEY_CLASSES_ROOT,
38    *                HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS,
```

```
39     *                    HKEY_CURRENT_CONFIG, HKEY_DYN_DATA
40     * @param thePath Путь к ключу в реестре
41    */
42    public Win32RegKey(int theRoot, String thePath)
43    {
44      root = theRoot;
45      path = thePath;
46    }
47
48    /**
49     * Перечисляет все имена ключей в реестре по пути,
50     * описываемому в данном объекте
51     * @return Возвращает список всех перечисляемых имен
52    */
53    public Enumeration<String> names()
54    {
55      return new Win32RegKeyNameEnumeration(root, path);
56    }
57    static
58    {
59      System.loadLibrary("Win32RegKey");
60    }
61  }
62
63  class Win32RegKeyNameEnumeration implements Enumeration<String>
64  {
65    public native String nextElement();
66    public native boolean hasMoreElements();
67    private int root;
68    private String path;
69    private int index = -1;
70    private int hkey = 0;
71    private int maxsize;
72    private int count;
73
74    Win32RegKeyNameEnumeration(int theRoot, String thePath)
75    {
76      root = theRoot;
77      path = thePath;
78    }
79  }
80
81  class Win32RegKeyException extends RuntimeException
82  {
83    public Win32RegKeyException()
84    {
85    }
86
87    public Win32RegKeyException(String why)
88    {
89      super(why);
90    }
91  }
```

Листинг 12.22. Исходный код из файла `win32reg/Win32RegKey.c`

```
1    /**
2     * @version 1.00 1997-07-01
```

```
3     * @author Cay Horstmann
4    */
5
6    #include "Win32RegKey.h"
7    #include "Win32RegKeyNameEnumeration.h"
8    #include <string.h>
9    #include <stdlib.h>
10   #include <windows.h>
11   JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue(
12      JNIEnv* env, jobject this_obj, jobject name)
13   {
14     const char* cname;
15     jstring path;
16     const char* cpath;
17     HKEY hkey;
18     DWORD type;
19     DWORD size;
20     jclass this_class;
21     jfieldID id_root;
22     jfieldID id_path;
23     HKEY root;
24     jobject ret;
25     char* cret;
26
27     /* получить класс */
28     this_class = (*env)->GetObjectClass(env, this_obj);
29
30     /* получить идентификаторы полей */
31     id_root = (*env)->GetFieldID(env, this_class, "root", "I");
32     id_path = (*env)->GetFieldID(env, this_class, "path",
33        "Ljava/lang/String;");
34
35     /* получить поля */
36     root = (HKEY) (*env)->GetIntField(env, this_obj, id_root);
37     path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
38     cpath = (*env)->GetStringUTFChars(env, path, NULL);
39
40     /* открыть ключ в реестре */
41     if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) != ERROR_SUCCESS)
42     {
43       (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
44          "Open key failed");
45       (*env)->ReleaseStringUTFChars(env, path, cpath);
46       return NULL;
47     }
48
49     (*env)->ReleaseStringUTFChars(env, path, cpath);
50    cname = (*env)->GetStringUTFChars(env, name, NULL);
51
52     /* обнаружить тип и длину значения */
53     if (RegQueryValueEx(hkey, cname, NULL, &type, NULL, &size)
54        != ERROR_SUCCESS)
55     {
56       (*env)->ThrowNew(env, (*env)->FindClass(
57          env, "Win32RegKeyException"), "Query value key failed");
58       RegCloseKey(hkey);
59       (*env)->ReleaseStringUTFChars(env, name, cname);
60       return NULL;
```

```
61    }
62    /* выделить область памяти для хранения конкретного значения */
63    cret = (char*)malloc(size);
64
65    /* прочитать значение */
66    if (RegQueryValueEx(hkey, cname, NULL, &type, cret, &size)
67       != ERROR_SUCCESS)
68    {
69      (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
70         "Query value key failed");
71      free(cret);
72      RegCloseKey(hkey);
73      (*env)->ReleaseStringUTFChars(env, name, cname);
74      return NULL;
75    }
76
77    /* сохранить значение как символьную строку, целое число или
78     * байтовый массив в зависимости от его типа */
79    if (type == REG_SZ)
80    {
81      ret = (*env)->NewStringUTF(env, cret);
82    }
83    else if (type == REG_DWORD)
84    {
85      jclass class_Integer = (*env)->FindClass(env, "java/lang/Integer");
86      /* получить идентификатор метода для конструктора */
87      jmethodID id_Integer = (*env)->GetMethodID(
88         env, class_Integer, "<init>", "(I)V");
89      int value = *(int*) cret;
90      /* вызвать конструктор */
91      ret = (*env)->NewObject(env, class_Integer, id_Integer, value);
92    }
93    else if (type == REG_BINARY)
94    {
95      ret = (*env)->NewByteArray(env, size);
96      (*env)->SetByteArrayRegion(env, (jarray) ret, 0, size, cret);
97    }
98    else
99    {
100     (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
101        "Unsupported value type");
102     ret = NULL;
103   }
104
105   free(cret);
106   RegCloseKey(hkey);
107   (*env)->ReleaseStringUTFChars(env, name, cname);
108
109   return ret;
110 }
111 JNIEXPORT void JNICALL Java_Win32RegKey_setValue(JNIEnv* env,
112    jobject this_obj, jstring name, jobject value)
113 {
114   const char* cname;
115   jstring path;
116   const char* cpath;
117   HKEY hkey;
118   DWORD type;
```

```
119   DWORD size;
120   jclass this_class;
121   jclass class_value;
122   jclass class_Integer;
123   jfieldID id_root;
124   jfieldID id_path;
125   HKEY root;
126   const char* cvalue;
127   int ivalue;
128
129   /* получить класс */
130   this_class = (*env)->GetObjectClass(env, this_obj);
131
132   /* получить идентификаторы полей */
133   id_root = (*env)->GetFieldID(env, this_class, "root", "I");
134   id_path = (*env)->GetFieldID(env, this_class, "path",
135      "Ljava/lang/String;");
136
137   /* получить поля */
138   root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
139   path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
140   cpath = (*env)->GetStringUTFChars(env, path, NULL);
141
142   /* открыть ключ в реестре */
143   if (RegOpenKeyEx(root, cpath, 0, KEY_WRITE, &hkey) != ERROR_SUCCESS)
144   {
145     (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
146        "Open key failed");
147     (*env)->ReleaseStringUTFChars(env, path, cpath);
148     return;
149   }
150
151   (*env)->ReleaseStringUTFChars(env, path, cpath);
152   cname = (*env)->GetStringUTFChars(env, name, NULL);
153
154   class_value = (*env)->GetObjectClass(env, value);
155   class_Integer = (*env)->FindClass(env, "java/lang/Integer");
156   /* определить тип объекта, представляющего значение */
157   if ((*env)->IsAssignableFrom(env, class_value, (*env)->FindClass(
158      env, "java/lang/String")))
159   {
160     /* это строка, поэтому получить указатель на ее символы */
161     cvalue = (*env)->GetStringUTFChars(env, (jstring) value, NULL);
162     type = REG_SZ;
163     size = (*env)->GetStringLength(env, (jstring) value) + 1;
164   }
165   else if ((*env)->IsAssignableFrom(env, class_value, class_Integer))
166   {
167     /* это целое значение, поэтому вызвать метод intValue(),
168      * чтобы получить значение */
169     jmethodID id_intValue = (*env)->GetMethodID(
170        env, class_Integer, "intValue", "()I");
171     ivalue = (*env)->CallIntMethod(env, value, id_intValue);
172     type = REG_DWORD;
173     cvalue = (char*)&ivalue;
174     size = 4;
175   }
176   else if ((*env)->IsAssignableFrom(env, class_value,
```

```
177       (*env)->FindClass(env, "[B")))
178   {
179     /* это байтовый массив, поэтому получить указатель на байты */
180     type = REG_BINARY;
181     cvalue = (char*)(*env)->GetByteArrayElements(
182        env, (jarray) value, NULL);
183     size = (*env)->GetArrayLength(env, (jarray) value);
184   }
185   else
186   {
187     /* неизвестно, как обрабатывать этот тип данных */
188     (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
189        "Unsupported value type");
190     RegCloseKey(hkey);
191     (*env)->ReleaseStringUTFChars(env, name, cname);
192     return;
193   }
194
195   /* установить значение */
196   if (RegSetValueEx(hkey, cname, 0, type, cvalue, size) != ERROR_SUCCESS)
197   {
198     (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
199        "Set value failed");
200   }
201
202   RegCloseKey(hkey);
203   (*env)->ReleaseStringUTFChars(env, name, cname);
204
205   /* если значение оказалось символьной строкой или байтовым,
206    * массивом освободить указатель */
207   if (type == REG_SZ)
208   {
209     (*env)->ReleaseStringUTFChars(env, (jstring) value, cvalue);
210   }
211   else if (type == REG_BINARY)
212   {
213     (*env)->ReleaseByteArrayElements(env, (jarray) value,
214        (jbyte*) cvalue, 0);
215   }
216 }
217
218 /* Вспомогательная функция, начинающая процесс перечисления имен */
219 static int startNameEnumeration(JNIEnv* env, jobject this_obj,
220    jclass this_class)
221 {
222   jfieldID id_index;
223   jfieldID id_count;
224   jfieldID id_root;
225   jfieldID id_path;
226   jfieldID id_hkey;
227   jfieldID id_maxsize;
228
229   HKEY root;
230   jstring path;
231   const char* cpath;
232   HKEY hkey;
233   DWORD maxsize = 0;
234   DWORD count = 0;
```

```
235
236   /* получить идентификаторы полей */
237   id_root = (*env)->GetFieldID(env, this_class, "root", "I");
238   id_path = (*env)->GetFieldID(env, this_class, "path",
239      "Ljava/lang/String;");
240   id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
241   id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize", "I");
242   id_index = (*env)->GetFieldID(env, this_class, "index", "I");
243   id_count = (*env)->GetFieldID(env, this_class, "count", "I");
244
245   /* получить значения из полей */
246   root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
247   path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
248   cpath = (*env)->GetStringUTFChars(env, path, NULL);
249
250   /* открыть ключ в реестре */
251   if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) != ERROR_SUCCESS)
252   {
253     (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
254        "Open key failed");
255     (*env)->ReleaseStringUTFChars(env, path, cpath);
256     return -1;
257   }
258   (*env)->ReleaseStringUTFChars(env, path, cpath);
259   /* запросить число и максимальную длину имен */
260   if (RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL, NULL, NULL,
261      &count, &maxsize, NULL, NULL, NULL) != ERROR_SUCCESS)
262   {
263     (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
264        "Query info key failed");
265     RegCloseKey(hkey);
266     return -1;
267   }
268
269   /* установить значения в полях */
270   (*env)->SetIntField(env, this_obj, id_hkey, (DWORD) hkey);
271   (*env)->SetIntField(env, this_obj, id_maxsize, maxsize + 1);
272   (*env)->SetIntField(env, this_obj, id_index, 0);
273   (*env)->SetIntField(env, this_obj, id_count, count);
274   return count;
275 }
276
277 JNIEXPORT jboolean JNICALL Java_Win32RegKeyNameEnumeration_hasMoreElements(
278    JNIEnv* env, jobject this_obj)
279 { jclass this_class;
280   jfieldID id_index;
281   jfieldID id_count;
282   int index;
283   int count;
284   /* получить класс */
285   this_class = (*env)->GetObjectClass(env, this_obj);
286
287   /* получить идентификаторы полей */
288   id_index = (*env)->GetFieldID(env, this_class, "index", "I");
289   id_count = (*env)->GetFieldID(env, this_class, "count", "I");
290
291   index = (*env)->GetIntField(env, this_obj, id_index);
292   if (index == -1) /* в первый раз */
```

```
293   {
294     count = startNameEnumeration(env, this_obj, this_class);
295     index = 0;
296   }
297   else
298   count = (*env)->GetIntField(env, this_obj, id_count);
299   return index < count;
300 }
301
302 JNIEXPORT jobject JNICALL Java_Win32RegKeyNameEnumeration_nextElement(
303    JNIEnv* env, jobject this_obj)
304 {
305   jclass this_class;
306   jfieldID id_index;
307   jfieldID id_hkey;
308   jfieldID id_count;
309   jfieldID id_maxsize;
310
311   HKEY hkey;
312   int index;
313   int count;
314   DWORD maxsize;
315
316   char* cret;
317   jstring ret;
318
319   /* получить класс */
320   this_class = (*env)->GetObjectClass(env, this_obj);
321
322   /* получить идентификаторы полей */
323   id_index = (*env)->GetFieldID(env, this_class, "index", "I");
322   id_count = (*env)->GetFieldID(env, this_class, "count", "I");
323   id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
324   id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize", "I");
325
326   index = (*env)->GetIntField(env, this_obj, id_index);
327   if (index == -1) /* first time */
328   {
329     count = startNameEnumeration(env, this_obj, this_class);
330     index = 0;
331   }
332   else
333     count = (*env)->GetIntField(env, this_obj, id_count);
334
335   if (index >= count) /* already at end */
336   {
337     (*env)->ThrowNew(env, (*env)->FindClass(
338        env, "java/util/NoSuchElementException"),
339        "past end of enumeration");
340     return NULL;
341   }
342
343   maxsize = (*env)->GetIntField(env, this_obj, id_maxsize);
344   hkey = (HKEY)(*env)->GetIntField(env, this_obj, id_hkey);
345   cret = (char*)malloc(maxsize);
346
347   /* обнаружить следующее имя */
348   if (RegEnumValue(hkey, index, cret, &maxsize, NULL, NULL, NULL, NULL)
```

```
349       != ERROR_SUCCESS)
350    {
351       (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
352          "Enum value failed");
353       free(cret);
354       RegCloseKey(hkey);
355       (*env)->SetIntField(env, this_obj, id_index, count);
356       return NULL;
357    }
358
359    ret = (*env)->NewStringUTF(env, cret);
360    free(cret);
361
362    /* инкрементировать индекс */
363    index++;
364    (*env)->SetIntField(env, this_obj, id_index, index);
365
366    if (index == count) /* at end */
367    {
368       RegCloseKey(hkey);
369    }
370
371    return ret;
372 }
```

Листинг 12.13. Исходный код из файла `win32reg/Win32RegKeyTest.java`

```
1  import java.util.*;
2
3  /**
4   * @version 1.02 2007-10-26
5   * @author Cay Horstmann
6  */
7  public class Win32RegKeyTest
8  {
9     public static void main(String[] args)
10    {
11       Win32RegKey key = new Win32RegKey(
12       Win32RegKey.HKEY_CURRENT_USER,
13          "Software\\JavaSoft\\Java Runtime Environment");
14
15       key.setValue("Default user", "Harry Hacker");
16       key.setValue("Lucky number", new Integer(13));
17       key.setValue("Small primes", new byte[] { 2, 3, 5, 7, 11 });
18
19       Enumeration<String> e = key.names();
20
21       while (e.hasMoreElements())
22       {
23          String name = e.nextElement();
24          System.out.print(name + "=");
25
26          Object value = key.getValue(name);
27          if (value instanceof byte[])
28             for (byte b : (byte[]) value) System.out.print((b & 0xFF) + " ");
29          else
30             System.out.print(value);
```

```
31
32          System.out.println();
33        }
34      }
35  }
```

Функции запроса типа данных

- `jboolean IsAssignableFrom(JNIEnv* env, jclass cl1, jclass cl2)`

  Возвращает значение `JNI_TRUE`, если объекты одного класса (*cl1*) могут быть присвоены объектам другого класса (*cl2*), в противном случае — значение `JNI_FALSE`. Проверяет, являются ли классы *cl1* и *cl2* одинаковыми, является ли класс *cl1* производным от класса *cl2* или представляет ли класс *cl2* интерфейс, реализуемый классом `cl1` или же каким-нибудь из его суперклассов.

- `jclass GetSuperclass(JNIEnv* env, jclass cl)`

  Возвращает суперкласс указанного класса. Если же *cl* представляет класс `Object` или интерфейс, то возвращает значение `NULL`.

Вот и подошел к концу второй том данной книги, завершающий долгий экскурс в Java. В этом томе у вас была возможность ознакомиться со многими расширенными средствами программирования на Java и дополнительными прикладными интерфейсами API. В начале этого тома были рассмотрены такие важные для всякого программирующего на Java темы, как потоки ввода-вывода, XML, сетевые подключения, базы данных и интернационализация. Затем последовали три длинные главы, посвященные программированию графики и ГПИ. А в завершающих этот том главах обсуждались специальные вопросы безопасности, вызова удаленных методов, обработки аннотаций и применения собственных методов. Надеемся, что этот экскурс в расширенные средства программирования на Java и прикладные интерфейсы API оказался для вас занимательным и полезным, и теперь вы сможете применить приобретенные знания и навыки в своих проектах.

Предметный указатель

D

DTD
  задаваемые правила, 127
  определения
    сущностей, 130
    типов документов, 125
  связывание определений с XML-документами, 126

J

JDBC
  активизация трассировки, 248
  драйверы
    JAR-файлы, 245
    типы, 236
  место в трехуровневой модели приложений, 238
  основные цели, 237
  пул соединений, организация, 297
  синтаксис записи URL, 245

S

SQL
  большие объекты, разновидности, 266
  встроенные функции, 243
  исключения
    анализ, 253
    типы, 254
  команды
    CREATE TABLE, 243
    DELETE, 243
    INSERT, 243
    SELECT, 242
    UPDATE, 243
    порядок выполнения, 249
  метаданные
    определение, 282
    типы, 282
  назначение, 239
  переходы
    назначение, 268
    синтаксис, 268
  предупреждения, организация, 255
  типы данных
    основные, 244
    расширенные, 295

X

XML
  вывод документов
    в виде древовидной структуры DOM, 168
    средствами StAX, 172
  допустимые типы атрибутов, 128
  инструкции разметки и обработки, 112
  определения DTD
    задаваемые правила, 127
    способы предоставления, 126
  поиск информации средствами XPath, 149
  практическое применение документов, 135
  преобразование документов средствами XSLT, 180
  проверка достоверности документов
    на соответствие определениям DTD, 130
    особенности и средства, 125
    по схеме типа XML Schema, 135
  пространства имен
    механизм, 155
    обозначение, 155
    префикс, 156
    режим управления, 156
  синтаксический анализ
    дерева DOM, 115
    документов, 112
  смешанное содержимое, 111
  структура документов, особенности, 110
  сущности, определения и ссылки, 130
  схемы документов типа XML Schema
    назначение, 133
    простые и сложные типы элементов разметки, 133

сходство и отличие от HTML, 109
формирование документов
без пространств имен, 167
с пространствами имен, 168
чтение документов, 113
элементы разметки и атрибуты, употребление, 111
XML Schema
назначение, 125
особенности, 126
схемы XML-документов, 133
типы элементов разметки
анонимное определение, 134
простые и сложные, 133
XPath
назначение, 149
операции и выражения, 149
описание узлов дерева DOM, 149
функции, 150
XSLT
назначение, 180
преобразование XML-документов, 180
спецификация, 181
таблицы стилей
создание, 180
шаблоны преобразования, 181

Z

ZIP-архивы
назначение, 49
чтение и запись данных, 49

А

Алгоритмы
безопасного хеширования
SHA, 59
SHA-1, 808
интерполяции, 597
кодирования
Base64, 218
знаком %, 676
обработки компоновки печатаемых страниц, 615
определения краев изображения, 599
регистрации, 794
шифрования
AES, 829
DES, 829
DSA, 811; 812
MD5, 809
RSA, 812; 837
SHA-1, 809
открытым ключом, 836
симметричные, 836
тайнопись Юлия Цезаря, 766
Аннотации
для компиляции, 876
для управления ресурсами, 877
инструментальные средства, 866
маркерные и однозначные, 872
места ввода в код, 874
мета-аннотации, 877
обозначение в коде, 866
обработка
во время выполнения, 868
на уровне
байт-кода, 868
исходного кода, 868; 880
обработчики
выполнение 882
присоединение к компилятору, 882
событий, 867
объявление, 872
определение, 865
применение, 865
стандартные, 875
элементы
допустимые типы, 873
объявление, 871
определение, 866
Аплеты
дополнительные полномочия, 808
уровни защиты, 808
Аутентификация
механизмы, 793
модули правил регистрации, 794
пользователей, 793
проблема и разрешение, 816
ролевая, механизм, 799
субъект и принципалы, 794

Б

Базы данных
Apache Derby, 244
выполнение запросов, 260
групповые обновления, 293

заполнение, 256
запуск сервера, порядок действий, 245
извлечение автоматически генерируемых ключей, 270
как набор таблиц, 239
метаданные, применение, 283
множественные результаты, порядок получения, 269
подготовка запросов, 260
пул соединений, организация, 297
разновидности СУБД, 244
скалярные функции, 268
соединение таблиц
внешнее, 268
преимущества, 241
составление запросов
в текстовом виде на SQL, 241
по образцу, 241
схема и каталог, 290
транзакции для сохранение целостности, 291
хранимые процедуры, 268
чтение и запись больших объектов, 266
экспериментальные, создание, 244
Безопасность
верификация байт-кода, 771
диспетчер защиты
виды проверок, 775
назначение, 760
установка, 781
источники кода
кодовая база, 776
наборы сертификатов, 776
механизмы обеспечения, 759
полномочия
задание, 782
назначение, 776
пользовательские, 785
порядок предположения, 786
проверка, 778
правила защиты
диспетчер, назначение, 779
назначение, 776
применение, 775; 780
Библиотеки
AWT
всплывающие меню, 681
расширенные средства, 525
BCEL, конструирование байт-кодов, 887
Swing
всплывающие меню, 681
компоненты
JEditorPane, 473–477
JList, 354–366
JProgressBar, 479
JSpinner, 465
JTable, 370–400
JTree, 407–36
вспомогательные, 492; 519
текстовые, 443
расширенные средства, 353
Буфер обмена
механизмы
вырезания и вставки, возможности, 637
передачи данных, возможности, 638
перетаскивания, поддержка, 653
передача
изображений, 645
объектов, 649
текста, 639
реализация, 638
Буферы данных
определение, 91
принцип действия, 92
свойства, 91

В

Ввод-вывод
байтов, 20
двоичных данных, 42
разновидности, 24
текста, 31
Верификаторы
виды проверок, 770
назначение, 770
Внутренние фреймы
диалоговые окна, создание, 509
контроль над установкой свойств, 508
перетаскивание контуров, 510
расположение
каскадное, 505
мозаичное, 506
создание, 502
состояния, 506

Вызов удаленных методов
для заглушек, 901
передача объектов
неудаленных , 913
удаленных, 913
принцип действия, 900
процесс, 906
с помощью прокси-объектов, 899
упаковка параметров, 901

Д

Двухмерная графика
внутреннее представление координат, 530
возможности рисования в прикладном интерфейсе Java 2D API, 526
вывод на печать, порядок действий, 605
дуги
построение, 530
расчет углов скашивания, 533
кривые второго и третьего порядка, построение, 534
многоугольники, построение, 535
обводка
по умолчанию, 546
стили линий, 546
отсечение, назначение, 562
правила композиции, 565
преобразования
аффинные, 558
координат, разновидности, 556
составные, порядок выполнения, 558
прозрачность, описание в альфа-канале, 565
прямоугольники, построение, 530
указания по воспроизведению
задание, 573
ключи и значения, 573
сглаживание, 574
участки, построение, операции, 544
фигуры
воспроизведение в конвейере визуализации, 528
отсечения, 562
порядок действий при рисовании, 526
раскраска, разновидности, 554
Деревья
воспроизведение узлов, 425
модели
назначение и построение, 408
по умолчанию, 408
специальные, 438
назначение, 407
обозначение узлов, 413
обработка событий, 429
обход узлов, способы, 423
основные составляющие и их обозначение, 407
поиск узлов по пути к дереву, 416
пользовательские объекты в узлах, 409
простые, построение, 408
редактирование, 415
редакторы ячеек, реализация, 419
сворачивание и разворачивание, 412
Доступ
к FTP-файлу, защищенному паролем, особенности, 218
к базам данных средствами SQL, 239
к веб-ресурсам, порядок действий, 215
к веб-странице, защищенной паролем, порядок действий, 217

З

Загрузчики классов
в качестве пространств имен, применение, 764
иерархия, 761
инверсия, 763
контекста, назначение, 763
разновидности, 761
специальные, создание, 764
Запуск настольных приложений, поддержка, 675

И

Идентификаторы ресурсов, унифицированные
абсолютные и относительные, 214
иерархическая структура, 214
определение, 214
преобразование, 215
прозрачные и непрозрачные, 214
Изображения
выборочные значения цвета пикселя, 590
выбор средств чтения и записи, 580
вывод на печать, 605

размытие, 599
растровые, формирование, 590
фильтрация, 597
цветовые профили IIC, применение, 591
чтение и запись нескольких изображений, 581
особенности, 579
Имена ресурсов, унифицированные, 214
Индикаторы выполнения
назначение, 480
создание, 480
Интегрированные среды разработки
импорт компонентов JavaBeans, 695
инспекторы свойств, назначение, 699
настройщики
ввод в компонент, 730
вызов, 732
назначение, 730
создание классов, 732
редакторы свойств
графические, 726
назначение, 719
предоставление, 719
создание, 723
строковые, 723
составление приложений из готовых компонентов, 695
Интернационализация
идентификаторы валют, 311
кодирование символов, 332
коды языков и стран, 301
комплекты ресурсов, назначение, 333
назначение, 299
прикладной программы, пример, 337
региональные настройки, 300
сортировка, особенности, 320
файлы свойств, 335
форматирование
даты и времени, 312
денежных сумм, 311
сообщений, 327
чисел, 305
Интерфейсы
AnnotatedElement, реализация и методы, 867
Annotation, расширение и методы, 873
BeanInfo
методы, 716
назначение и реализация, 715
BufferedImageOp, назначение и реализация, 590; 597
Compilable, назначение и реализация, 848
Composite, назначение и реализация, 566
ContentHandler, методы, 158; 162
Customizer, реалиализация и методы, 732
DatabaseMetaData
методы, 272; 283
назначение, 283
DataInput, реализация и методы, 43
DataOutput, реализация и методы, 42
DataSource, назначение, 296
DiagnosticListener, назначение и реализация, 855
Document, назначение и реализация, 444
Doc, назначение и реализация, 627
EntityResolver, реализация и методы, 127; 132
ErrorHandler, реализация и методы, 131
FileVisitor, назначение, реализация и методы, 81
HyperlinkListener, назначение, 476
Invocable, назначение и реализация, 846
JavaFileManager, назначение, 855
ListCellRenderer, назначение и методы, 366
ListModel, назначение и методы, 360
LSSerializer, применение и реализация, 169
Node
иерархия наследования и реализация, 113
методы, 157
Paint, назначение и реализация, 554
Printable, реализация и метод, 606
PrintRequestAttributeSet, назначение и реализация, 607
PropertyEditor
методы, 726
реализация, 723
Remote, назначение и методы, 903
ResultSetMetaData, назначение, 283

Result, реализация, 185
RowSet
  методы, 280
  назначение, 278
  расширение, 278
Runnable, реализация, 203
Serializable, назначение
    и реализация, 54
Shape, назначение и реализация, 529
Source, реализация, 184
StyledDocument, назначение
    и реализация, 444
TableCellEditor, назначение
    и методы, 398
TableCellRenderer, реализация
    и метод, 394
TreeCellRenderer, реализация
    и метод, 426
TreeModelListener, назначение
    и методы, 436
TreeModel, назначение и реализация,
    436
TreeSelectionListener, реализация
    и метод, 429
XMLReader, реализация, 184
аннотаций
  назначение, 866
  особенности, 873
атрибутов печати, 631
ввода-вывода
  иерархия, 25
  методы, 25
  реализация, 25
  деревьев, иерархия наследования, 409
  для передачи данных через буфер
      обмена, 638
  наборов атрибутов, 631
  удаленных объектов, назначение, 903

К

Классы
  AbstractListModel, назначение
      и методы, 361
  AbstractSpinnerModel, назначение
      и методы, 467
  AffineTransform, назначение
      и методы, 558
  AlphaComposite, назначение
      и методы, 566
  Area, назначение и методы, 545
  Attributes, методы, 163
  Banner, назначение и методы, 616
  BasicStroke
    конструкторы и методы, 553; 554
    назначение, 546
  Book, назначение и методы, 615
  BufferedImage, назначение
      и методы, 590
  BufferedReader, назначение
      и методы, 34
  Buffer, назначение, подклассы
      и методы, 91
  ByteLookupTable
    назначение, 598
  Charset, назначение и методы, 38
  Cipher, назначение и методы, 828
  ClassLoader, назначение и методы, 765
  Collator, назначение и методы, 321
  ColorConvertOp, назначение, 598
  ColorModel, назначение и методы, 592
  Color, назначение и методы, 592
  CompilationTask, применение, 856
  ConvolveOp
    конструкторы, 605
    назначение, 599
  Currency, назначение и методы, 311
  DataInputStream
    методы, 28
    назначение, 24
  DataOutputStream, назначение, 24
  DateFormat, назначение
      и методы, 313; 453
  DefaultListModel, назначение
      и методы, 365
  DefaultMutableTreeNode
    методы, 413; 423
    назначение, 409
  DefaultPersistenceDelegate, назначение
      и методы, 745
  DefaultTreeCellRenderer, назначение
      и методы, 425
  DefaultTreeModel
    методы, 417
    назначение, 408; 417
  DocumentBuilderFactory, методы, 123;
      132; 157
  DocumentBuilder, методы, 123; 131

DocumentFilter, назначение
и методы, 451
DriverManager
регистрация драйверов JDBC, 246
FileChannel, назначение и методы, 93
FileInputStream, назначение, 27
FileOutputStream, назначение, 27
Files
методы, 75
назначение, 72
Graphics
методы, 528
назначение, 525
Graphics2D
методы, 526; 528; 546; 558
назначение, 526
HashPrintRequestAttributeSet,
назначение и методы, 607
ImageIO, назначение и методы, 579; 581
ImageViewerBean, назначение и
методы, 690
InetAddress, назначение и методы, 197
InputStream
методы, 20
назначение, 20
InputStreamReader, назначение, 31
Introspector, назначение и методы , 716
JavaCompiler, назначение и методы, 855
JDesktopPane, методы, 506
JEditorPane
методы, 479
назначение, 475
JFormattedTextField
методы, 452
назначение, 448
JInternalFrame
методы, 508; 517
применение, 510
установка свойств, контроль, 508
JLabel, назначение и методы, 368
JLayer
конструкторы и методы, 522
назначение, 519
JList
методы, 359; 364
шаблон проектирования MVC,
применение, 360
JProgressBar
методы, 490
назначение, 479
JSpinner
конструкторы, 471
методы, 471
JSplitPane, конструкторы и методы, 494
JTabbedPane, конструкторы и методы,
496; 500
JTable
методы, 381
назначение, 370
JTree
конструкторы, 414
методы, 414; 417; 422; 434
назначение, 407
KeyPairGenerator, применение, 837
LayerUI
конструкторы, 519
методы, 520; 522
ListResourceBundle, назначение и
производные, 336
Locale
методы, 303
назначение, 300
LoginContext
методы, 798
назначение, 793
LookupOp, применение, 598
LookupTable, назначение, 598
MaskFormatter, назначение и методы,
454
Matcher, назначение и методы, 102
MessageDigest, назначение и методы,
809
MessageFormat, назначение и методы,
327
NumberFormat, назначение и методы,
305; 453
ObjectOutputStream, назначение и
методы, 54
OutputStream
методы, 20
назначение, 20
OutputStreamWriter, назначение, 31
PageFormat, назначение и методы, 608
Path
методы, 73
назначение, 72
Pattern, назначение и методы, 104
PrinterJob, назначение и методы, 606

PrintPanel, назначение и методы, 609
PrintServiceLookup, назначение и
методы, 627
PrintService, назначение и методы, 627
PrintWriter, назначение
и методы, 31; 951
ProgressMonitor
методы, 484
назначение, 483
ProgressMonitorInputStream
конструктор, 487
назначение, 486
PropertyChangeSupport, назначение
и методы, 707
PropertyDescriptor, назначение
и методы, 719
PushbackInputStream, назначение, 28
RandomAccessFile, назначение
и методы, 45
Raster, назначение и методы, 592
Reader
методы, 24
назначение, 20
RenderingHints, назначение, 573
RescaleOp
конструкторы, 604
назначение, 597
ResourceBundle
методы, 337
назначение и производные, 335
ResultSet
методы, 250; 272
обработка результирующих
наборов, 250
RoundRectangle2D, конструкторы, 530
RowFilter, назначение и методы, 384
SAXParserFactory, методы, 161
SAXParser, методы, 162
Scanner, назначение, 34
SecretKeyFactory, применение, 830
SecureRandom, назначение
и методы, 831
SecurityManager, назначение
и методы, 778
ServerSocket, методы, 202
ShortLookupTable
конструкторы, 604
назначение, 598
SimpleBeanInfo, назначение
и методы, 715
SimpleDoc, назначение
и конструктор, 627
Socket
методы, 196; 206
назначение, 196
SocketChannel
методы, 213
назначение, 207
SQLException, назначение и методы,
254
Statement, назначение и методы, 250
StreamPrintServiceFactory,
назначение, 630
StreamPrintService, назначение, 630
SystemTray, назначение и методы, 681
TableColumn, назначение и методы, 380
ThreadedEchoHandler, назначение
и методы, 203
TrayIcon, назначение и методы, 681
TreePath, назначение и методы, 416
UnicastRemoteObject, наследование, 904
URI, назначение и методы, 215
URL
методы, 213; 222
назначение, 213
URLClassLoader, экземпляры и
конструкторы, 762
URLConnection
методы, 216; 223
назначение, 213
VetoableChangeSupport, назначение
и методы, 708
Writer
методы, 24
назначение, 20
XMLEncoder
назначение, 739
поддерживаемые типы данных, 742
XMLInputFactory, методы, 165
XMLStreamReader, методы, 166
XPathFactory, методы, 154
XPath, методы, 155
ZipInputStream, назначение
и методы, 49
ZipOutputStream, назначение
и методы, 50
атрибутов печати, 631

деревьев, иерархия наследования, 409
драйверов JDBC, регистрация, 246
загрузка
  динамическая, 916
  механизм, 760
  разрешение класса, 760
информации о компонентах JavaBeans
  автоматическое генерирование, 881
  применение, 703
комплектов ресурсов, 335
полномочий
  домен защиты, 778
  иерархия, 777
  реализация, 785
  целевые объекты и действия, 782
потоков
  ввода-вывода, иерархия, 23
  шифрования, применение, 835
раскраски, 554
рисования
  кривых второго и третьего порядков, 529
  фигур, 529
сериализация, особенности, 64
таблиц и столбцов, иерархия, 379
типа BeanInfo
  назначение, 715
  создание, 715
Кодировки символов
  UTF-8 и UTF-16, 42
  назначение, 31
  применяемые в Java, 39
  разновидности, 38
Компилятор Java
  вызов
    из приложений, потребность, 854
    на компиляцию, простой, 854
  выполнение заданий
    на компиляцию, 854
  динамическое генерирование кода, пример, 860
Комплекты ресурсов
  иерархия, 334
  назначение, 333
  обнаружение, 334
Компоненты
  JavaBeans
    CalendarBean, 690
    ChartBean
      настройщик, 730
      свойства, 720
    FileNameBean, применение, 693; 709
    ImageViewerBean, применение, 690; 693; 706
    изменение свойств, 699
    импорт в ИСР, 694
    назначение, 689
    определение, 687
    свойства
      индексируемые, 706
      ограниченные, 708
      простые, 705
      связанные, 706
    создание, особенности, 690
    составление приложений в ИСР, 695
    сохраняемость
      делегат, создание и установка, 743
      долговременная, применение, 742
      механизм, 738
      устранение избыточности, 739
      характеристики архивов, 749
    упаковка в JAR-файл, 693
    шаблоны именования
      свойств, применение, 703
      событий, применение, 704
  Swing
    JEditorPane, назначение и применение, 473
    JList, назначение и применение, 354–366
    JProgressBar, назначение и применение, 480
    JSpinner, назначение и применение, 465
    JTable, назначение и применение, 370–400
    JTree, назначение и применение, 407–36
    вспомогательные, назначение и применение, 492–519
    источники перетаскивания, реализация, 659
    поддержка перетаскивания, 655
    приемники перетаскивания, реализация, 662
    режимы опускания перетаскиваемых объектов, 663
    текстовые, назначение и применение, 444
Конструирование байт-кодов

во время загрузки, 893
порядок действий, 888
Контекст
воспроизведения шрифтов, применение, 562
графический, установка, 528
печатающего устройства, графический, 608

М

Межсетевые адреса
и имена хостов, взаимное преобразование, 197
назначение, 197
по протоколу IPv6, 198
Многодокументный интерфейс, особенности, 502
Множество Мандельброта, п остроение, 592
Модели
баз данных, реляционные, 241
выбора, назначение и применение, 381
деревьев, назначение и построение, 408
доверительных отношений, 817
печати, назначение, 605
приложений
клиент-сервер, особенности, 237
трехуровневые, особенности, 238
программирования RMI, назначение, 902
списков, назначение и реализация, 362
счетчиков, назначение и построение, 466
таблиц, назначение и реализация, 375
Мониторы текущего состояния
назначение, 483
обработка запросов, 484
потоков ввода, организация, 487
создание, 483

Н

Наборы
символов
кодирование и декодирование, 40
назначение, 37
псевдонимы, 38
строк
кешируемые, 279
применение, 278
Настольные панели
назначение, 492
создание, 503
Начальные экраны
индикация хода загрузки, способы, 671
назначение, 670
отображение, 670

О

Обработка данных из формы по сценарию, порядок действий, 225
Отправка электронной почты
по протоколу SMTP, порядок действий, 231
средствами JavaMail API, 232

П

Панели с вкладками
компоновка, 496
назначение, 495
создание, 496
Передача данных на веб-сервер
команды и параметры, 226
порядок действий, 227
Перетаскивание
выполняемые операции, разновидности, 654
инициирование операции, 653
источник и приемник, 653
механизм, 653
поддержка в Swing, 656
Печать
атрибуты
категории, 633
наборы, 631
разновидности, 630
двухмерной графики, 605
многостраничная, 615
поддерживаемые типы данных, 625
полосовой способ, 608
предварительный просмотр, 617
принцип WYSIWIG, 609
разновидности документов
определение, 626
указание, 626
формат бумаги в пунктах, 608
Подписание
JAR-файлов, процедура, 822
аплетов, 823

кода, возможные сценарии, 821
Полузакрытие
назначение, 206
на стороне клиента, механизм, 206
Пользовательский интерфейс на основе фреймов, особенности, 502
Потоки
ввода
определение, 20
превращение в потоки чтения, 31
чтение с упреждением, 29
ввода-вывода
буферизация данных, 29
разделение обязанностей, 28
сериализируемых объектов, 53
создание и присоединение к файлам, 27
сочетание фильтров, 29
типы, 23
фильтрация, 487
вывода
определение, 20
превращение в потоки записи, 31
записи
автоматическая очистка, 32
вывод текста, 31
чтения
ввод текста, 34
объединение с источником данных, 34
шифрования, применение, 835
Прокси-объекты
технологии взаимодействия, 899
установка, 899
Протоколы
HTTP
запросы и ответы, 218
формат заголовка, 217
IIOP, назначение, 900
RMI
назначение, 900
протоколирование действий, 911
развертывание программ, 908
регистраторы RMI, 912
SMTP, назначение, 231
SOAP, применение, 900
TCP, особенности, 196
UDP, особенности, 196

Р

Разделяемые панели
компоновка, 492
назначение, 492
создание, 492
Разновидности данных
представление, 643
типы MIME, 643
указание, 643
Региональные настройки
описание, 304
определение, 300
предопределенные объекты, 302
Регулярные выражения
назначение, 96
признаки сопоставления с шаблоном, 100
применение, 100
синтаксические конструкции, 96
Реестр
RMI
динамическая загрузка файлов классов, 909
регистрация удаленных объектов, особенности, 905
Windows
организация доступа, 972
редактор, применение, 971
Результирующие наборы
анализ, 250
обновляемые, 273
прокручиваемые, 271
Роли клиента и сервера, 901

С

Свертка
операция, 599
сообщения
алгоритмы вычисления, 809
свойства, 808
ядро, 599
Серверы
обслуживание многих клиентов, 202
подключение через порт, особенности, 193
реализация, 199
типичный цикл работы, 200
Сериализация

объектов
клонирование, 69
контроль версий, 67
механизм, 55
определение, 53
формат файлов, 58
одноэлементных множеств, 66
переходных полей объектов, 64
типизированных перечислений, 66
Сертификаты
подписание, 819
применение, 815
разработчиков программного обеспечения, 826
Синтаксические анализаторы
XML-документов, разновидности, 112
древовидные
DOM-анализатор, применение, 113
особенности, 113
определение, 112
потоковые
SAX-анализатор, применение, 158
StAX-анализатор, применение, 163
особенности, 158
Синтаксический анализ
URI, особенности, 214
XML-документов, 112
смешанного содержимого, 128
Системная область
назначение, 680
Системы
ZIP-файлов
назначение, 82
применение, 83
распределенных объектов, 904
сборки мусора, локальные и распределенные, 914
Слои
добавление, 519
назначение, 519
установка масок событий, 520
Службы
JAAS
модули регистрации, назначение, 799
назначение, 793
применение, 799
JNDI, назначение, 296
печати
обычные, применение, 627
потоковые, применение, 630
разновидности документов, 626
реестра самозагрузки, назначение, 904
Собственные методы
вызов
методов экземпляра, 951
статических методов, 954
доступ
к полям экземпляра, 944
к реестру Windows, организация и реализация функций, 972
к статическим полям, 948
кодирование сигнатур, 949
обработка ошибок и исключений, 961
объявление, 930
организация вызова, 930
перегрузка, 931
связывание с программами на Java, 936
строковые параметры, 938
числовые параметры, 936
Собственный код
вызов методов на Java, 955; 966
инициализация виртуальной машины Java, 965
обработка, 934
правила написания, 931
прекращение работы виртуальной машины Java, 966
соответствие типов
массивов, 957
целесообразность применения, 930
Сокеты
блокирующие, 207
время ожидания, 196
закрытие, 200
определение, 195
открытие, 195
прерываемые, 207
Сортировка
ключи, 322
разложение на составляющие, режимы, 322
уровни избирательности, 321
формы нормализации в уникоде, 322
Списки
ввод и удаление значений, 365
воспроизведение ячеек, 366
выбор элементов, 355

значения прототипных ячеек, применение, 362
модели, назначение, 362
обработка событий, особенности, 356
очень длинные, составление, 361
простые
из символьных строк, составление, 354
порядок составления, 360
расположение элементов, варианты, 355
Сценарии
вызов, 843
компиляция, 848
механизмы
наиболее употребительные, 842
получение, 842
переадресация ввода-вывода, 845
привязки переменных, 844
создание, пример, 849
функции и методы, вызов, 846
Счетчики
назначение, 465

Т

Таблицы
особенности составления, 370
построение по модели, 375
простые, составление, 370
специальные редакторы ячеек, 398
столбцы
воспроизведение заголовков, 396
выбор, 381
доступ, 379
изменение размеров, 380
порядок воспроизведения, 378
сокрытие и показ, 385
строки
выбор, 381
изменение размеров, 381
сортировка, 383
фильтрация, 384
ячейки
воспроизведение, 394
выбор, 382
редактирование, 372; 396
Текстовые поля
ввод целочисленных значений, 448
проверка достоверности вводимых данных, 448
при потере фокуса ввода, 449
средства, 452
фильтры документов, применение, 450
форматирование
специальные средства, 455
стандартные средства, 453
Транзакции
автоматическая фиксация, 292
групповые обновления, 293
точки сохранения, 292
фиксация и откат, 291

У

Удаленные объекты
активизация
дескрипторы, 922
механизм, 922
вызов методов, 901
доступ, 906
одноадресный и многоадресный поиск, 904
регистрация в реестре RMI, 904
сравнение, 921
хеширование, 921
Указатели ресурсов, унифицированные, 214
Утилиты
crypt, применение, 831
jarsigner, назначение, 816
javah, назначение, 931
javap, применение, 772
keytool, назначение, 813
native2ascii, назначение, 332
policytool, назначение, 785
serialver, назначение, 67
telnet, применение, 191

Ф

Файлы
атрибуты
основные , 78
получение, 78
блокировка
механизм, 93
особенности, 94
разделяемая и исключительная, 94

изображений
поддерживаемые форматы, 579
чтение и запись, 579
обход по каталогам, 79
отображаемые в памяти
каналы, получение, 84
применение, 84
режимы отображения, 84
перемещение и удаление, 76
правил защиты
места установки, 780
содержимое, 779
создание и распространение, 822
пути
абсолютные и относительные, 72
разрешение, 73
составление, 72
свойств
назначение, 108
недостатки формата, 108
применение, 335
создание вместе с каталогами, 77
с произвольным доступом
открытие, 45
указатель файла, 45
чтение и запись данных, 45
фильтрация по глобальному шаблону, 80
чтение и запись данных, 75
Форматы
SVG, особенности, 169
XML
назначение, 108
преимущества, 109
выбора, назначение, 330
с обратным порядком следования байтов, 42
с прямым порядком следования байтов, 42
файлов для сериализации объектов, 58
Функции JNI
вызова
методов
альтернативные, 955
для генерирования исключений, 961
доступа
к массивам на Java, 958
к полям, 945
особенности вызова, 939

Ц

Цветовые модели
ARGB, 568
RGB, 564
SRGB, 591
назначение, 591
получение, 592
Центр сертификации
имитация, 819
создание средствами OpenSSL, 820
функции, 819
Цифровые подписи
алгоритмы, 808
верификация, 816
принцип действия, 811

Ш

Шифрование
генерирование ключей, 830
открытый и секретный ключи, 811
открытым ключом, 836
применение, 828
режимы свертывания и развертывания, 829
схемы заполнения, 830
хранилища ключей, 814

Я

Языки сценариев
доступные механизмы поддержки, 843
преимущества и недостатки, 842